15-210
PARALLEL AND SEQUENTIAL ALGORITHMS AND DATA STRUCTURES

LECTURE 1

OVERVIEW – THE GENOME SEQUENCING PROBLEM

MAJOR THEMES

- Defining precise problem and data abstractions,

- Designing and programming
  - correct and efficient algorithms and data structures
  - for given problems and data abstractions

	Abstraction	**Implementation**
Functions	Problem	Algorithm
Data	Abstract Data Type	Data Structure

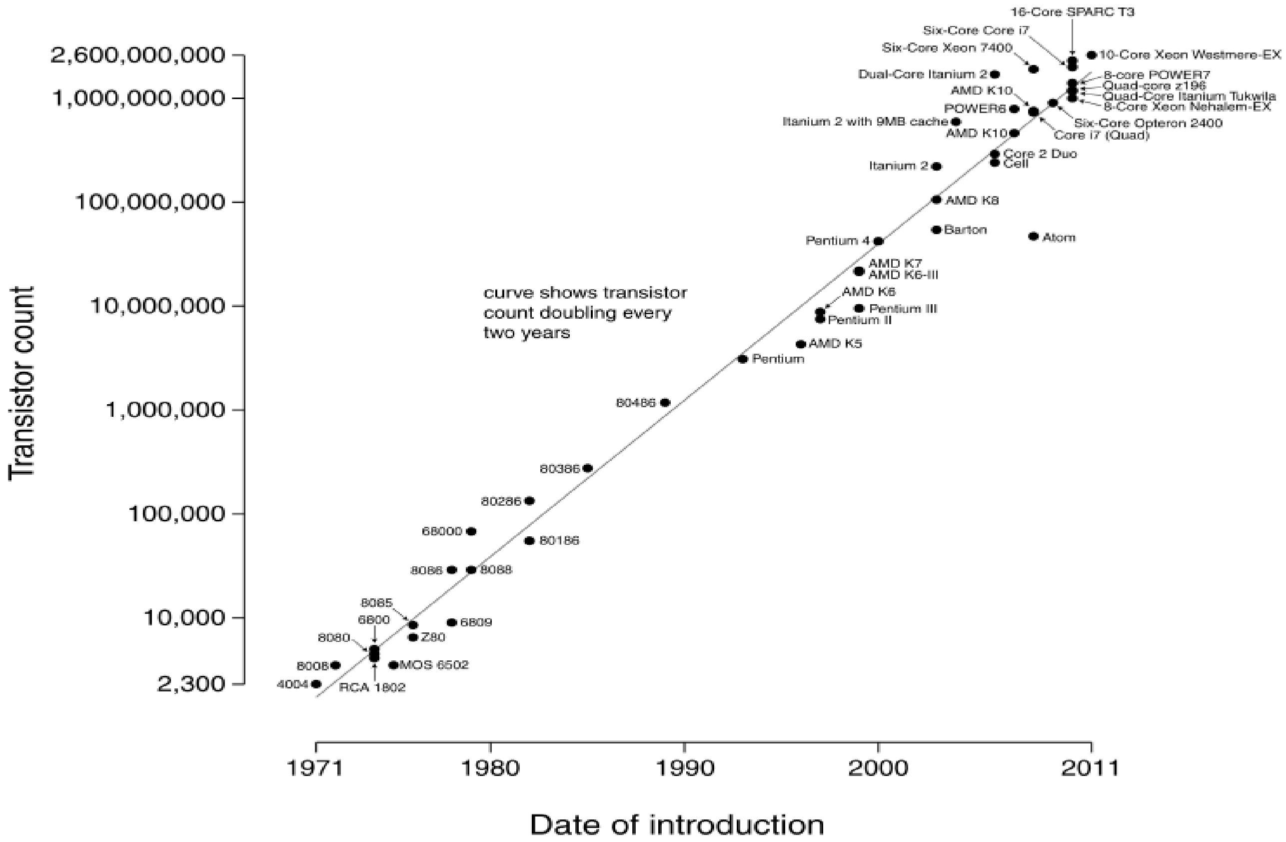
PROBLEM VS. ALGORITHM

- Sorting, string matching, finding shortest paths in graphs,. . . , are problems
    - **Input**: A sequence $[a_1, a_2, \cdots, a_n]$
    - **Output**: A permutation of the sequence $[a_{i_1}, a_{i_2}, \cdots, a_{i_n}]$ such that $\forall j, 1 \leq j < n, a_{i_j} \leq a_{i_{j+1}}$
- Quicksort, Mergesort, Insertion Sort , . . . , are algorithms for sorting.

ABSTRACT DATA TYPES VS. DATA STRUCTURES

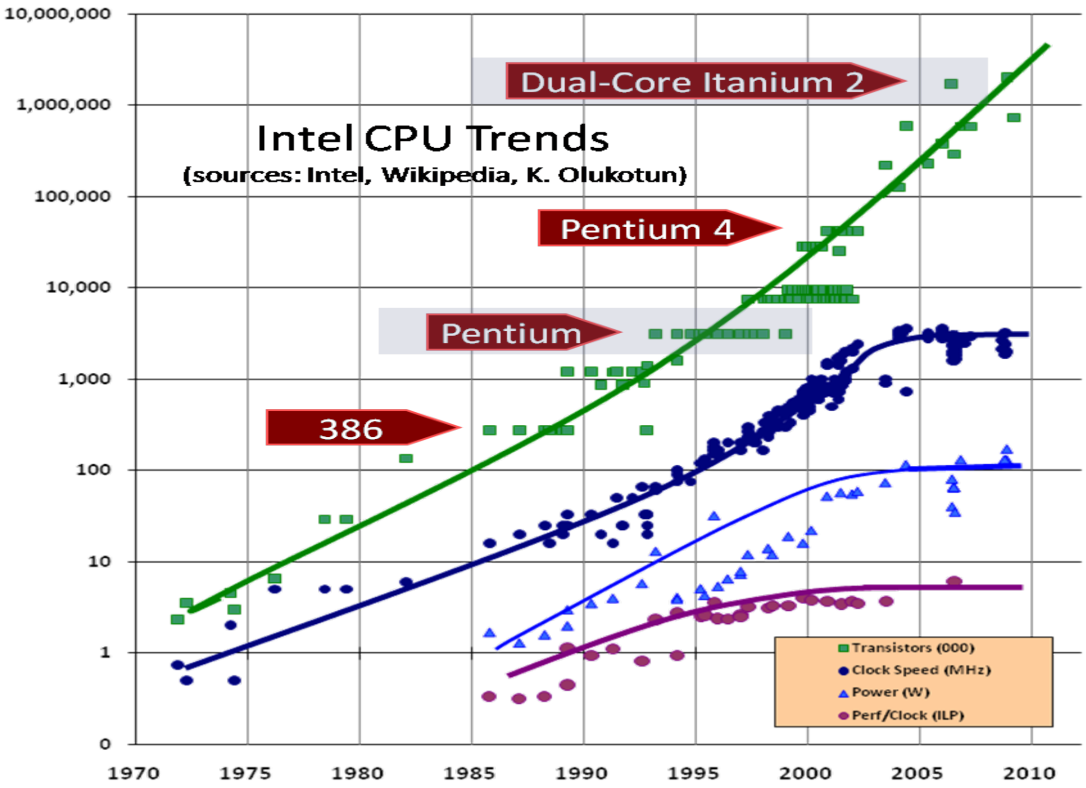
- A set is an abstract data type (ADT)
  - Test membership, intersect, union, difference, . . .
- Sequences, trees, hash-tables are examples of data structures.
- ADT's determine functionality, data structures determine costs.

TECHNOLOGY – MOORE'S LAW

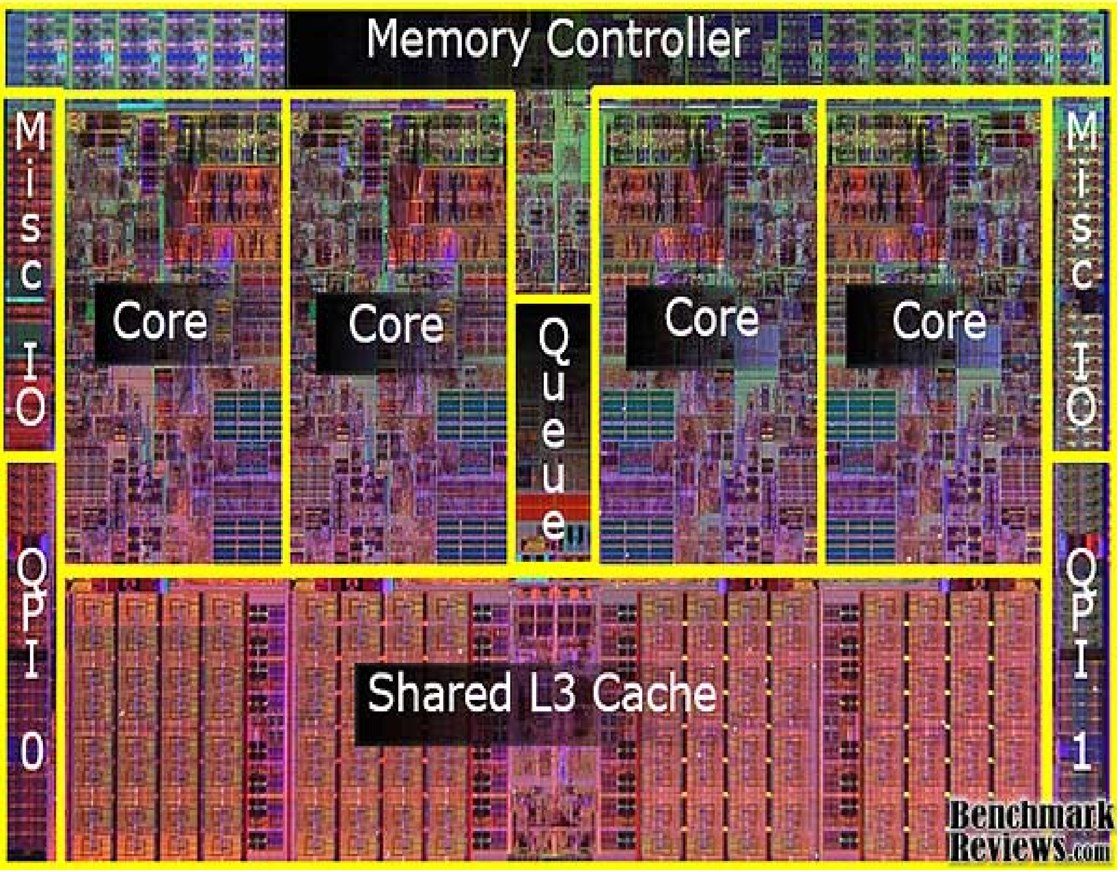


Source: Wikipedia

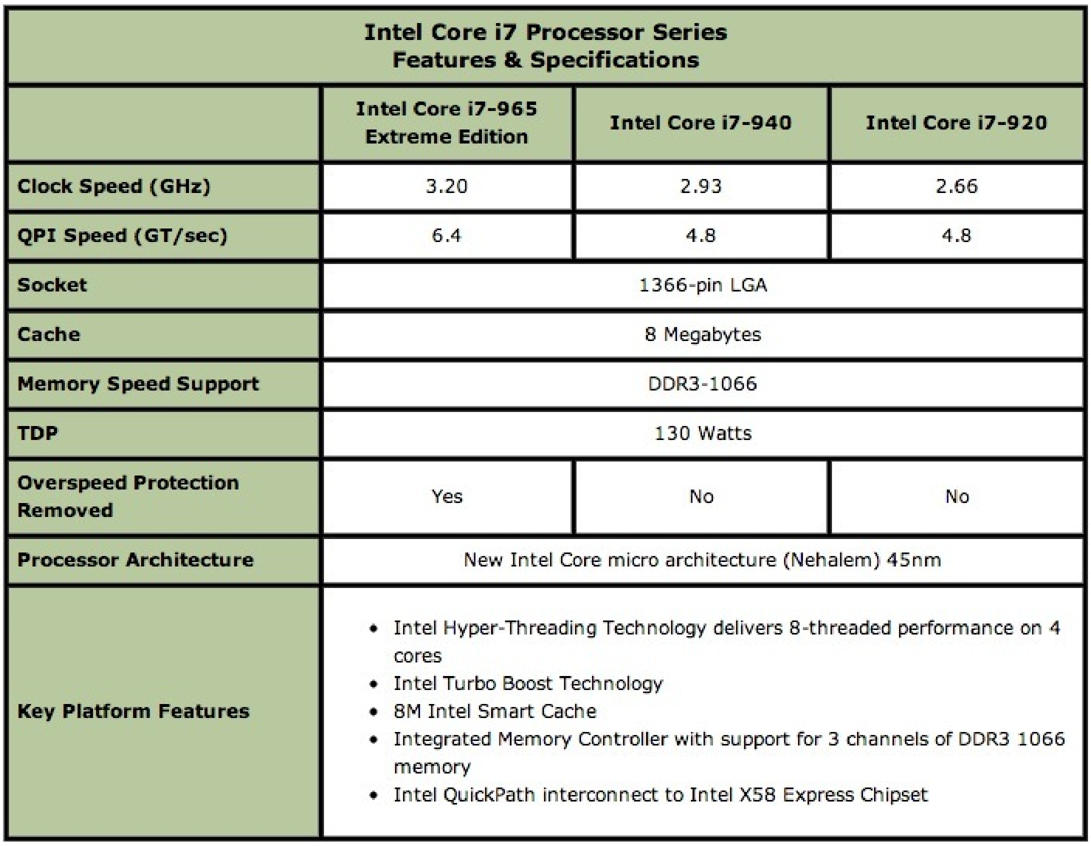
PROCESSOR TECHNOLOGY



MULTI-CORE CHIPS



MULTI-CORE CHIPS



Intel Core i7 Processor Series
Features & Specifications

	Intel Core i7-965 Extreme Edition	Intel Core i7-940	Intel Core i7-920
Clock Speed (GHz)	3.20	2.93	2.66
QPI Speed (GT/sec)	6.4	4.8	4.8
Socket	1366-pin LGA		
Cache	8 Megabytes		
Memory Speed Support	DDR3-1066		
TDP	130 Watts		
Overspeed Protection Removed	Yes	No	No
Processor Architecture	New Intel Core micro architecture (Nehalem) 45nm		
Key Platform Features	• Intel Hyper-Threading Technology delivers 8-threaded performance on 4 cores • Intel Turbo Boost Technology • 8M Intel Smart Cache • Integrated Memory Controller with support for 3 channels of DDR3 1066 memory • Intel QuickPath interconnect to Intel X58 Express Chipset		

PARALLEL ALGORITHMS

	Serial	**Parallel**		
		1-core	8-core	32h-core
Sorting 10M strings	2.90	2.90	0.40	.095 (30.5)
Remove dupl. 10M strings	0.66	1.00	0.14	.038 (17.4)
Min. span. tree 10M edges	1.60	2.50	0.42	.140 (11.4)
BFS 10M edges	0.82	1.20	0.20	.046 (17.8)

Running times in seconds

15-210 VS. A TRADITIONAL COURSE

- Emphasis on parallel thinking at a high level
  - Parallel algorithms and parallel data structures
- Purely functional model of computation
  - Safe for parallelism
  - Higher level of abstraction
- Ideas still relevant for imperative computation
  - Lot of overlap, but covered differently!

SYNOPSIS

- A real world problem: Gene sequencing.
- The computational problem.
- Algorithms

SEQUENCING THE GENOME

- The human DNA molecule encodes the complete set of genetic information using 4 bases
    - Adenine (A), Cytosine (C), Guanine (G) and Thymine (T)

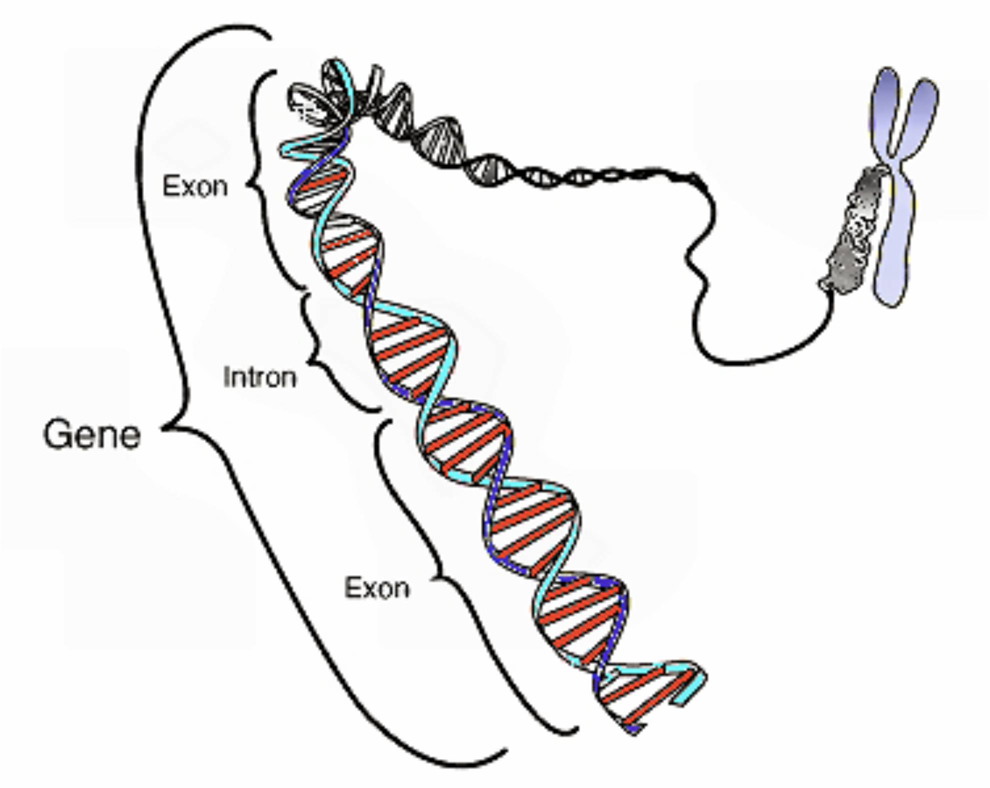
- A sequence of about
    - 3 billion base pairs
    - arranged into 46 chromosomes

  makes up the human genome.

SEQUENCING THE GENOME

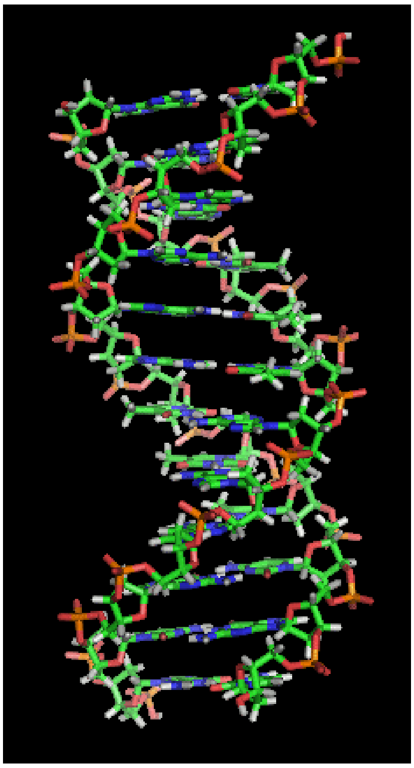
- A chromosome is a sequence of genes
- A gene is a sequence of the base pairs
  - But there seem to be a lot of base-pairs with no apparent functions.

SEQUENCING THE GENOME



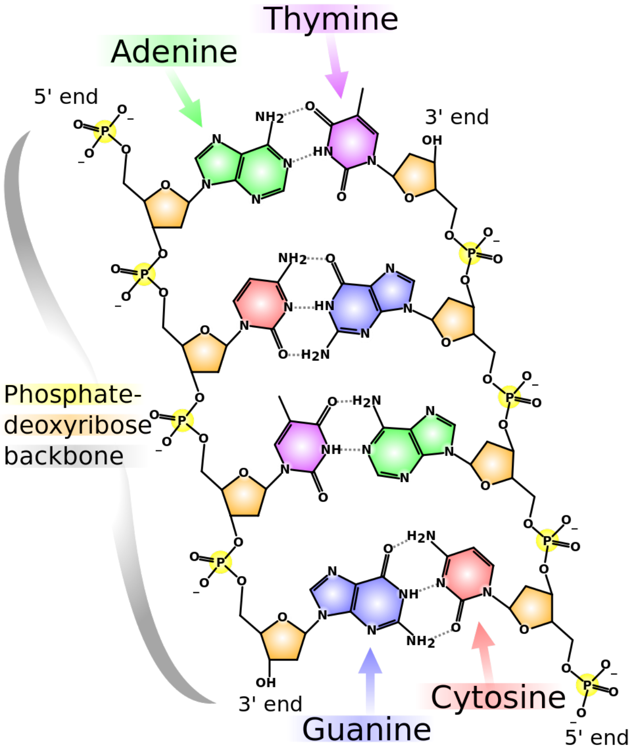
Source: Wikipedia

SEQUENCING THE GENOME



Source: Wikipedia

Sequencing the Genome



Source: Wikipedia

SEQUENCING THE GENOME

- Determining the complete DNA sequence is a grand challenge.
- Very hard to do in one go with wet lab techniques.
- The Shotgun Technique has been found work quite well.

SHOTGUN SEQUENCING

- Break up multiple DNA strands into short segments
  - Chemistry!
- Short segments are sequenced.
  - Chemistry!
- Stitch short sequences computationally.
  - This is where CS comes in.

SHOTGUN SEQUENCING

Strand	Sequence
Original	AGCATGCTGCAGTCATGCTTAGGCTA
First shotgun sequence	AGCATGCTGCAGTCATGCT------- -------------------TAGGCTA
Second shotgun sequence	AGCATG-------------------- -------CTGCAGTCATGCTTAGGCTA
Reconstruction	AGCATGCTGCAGTCATGCTTAGGCTA

Source: Wikipedia

SHOTGUN SEQUENCING

- Suppose you have three strands sequenced

  `catt ag gagtat`
  `cat tagg ag tat`
  `ca tta gga gtat`

- But they really come in a messy way, e.g.,

  `catt ag tta cat tagg ag gagtat`
  `tat ca gga gtat`

- So how do we stitch them?
  - Given a set of overlapping genome subsequences, construct the “best” sequence that includes them all.

SYNOPSIS

- A real world problem: Gene sequencing.
- The computational problem.
- Algorithms

THE ABSTRACT PROBLEM

THE SHORTEST SUPERSTRING PROBLEM

Given

- an alphabet of symbols Σ, and
- a set of finite strings $S \subseteq \Sigma^+$,

return

- a shortest string r that contains every $s \in S$ as a substring of r.

- Σ, Σ^+
- $\Sigma = \{A, C, G, T\}$

SOME OBSERVATIONS

- Ignore strings that are already in other strings. Why?

$$\{\text{catt}, \text{ag}, \text{gagtat}, \text{cat}, \text{tagg}, \text{ag}, \text{tat}, \text{ca}, \text{tta}, \text{gga}, \text{gtat}\}$$

$$\Downarrow$$

$$\{\text{catt}, \text{gagtat}, \text{tagg}, \text{tta}, \text{gga},\}$$

- Each string must start at a distinct position in the result. Why?

SYNOPSIS

- A real world problem: Gene sequencing.
- The computational problem.
- Algorithms:
    - The Brute Force Algorithm

THE BRUTE FORCE ALGORITHM

THE BRUTE FORCE TECHNIQUE

Enumerate all possible candidate solutions for a problem

- score each solution, and/or
- check each satisfies the problem constraints

Return the best solution.

- How does this apply to the SS Problem?
    - Generate permutations
    - Remove overlaps
    - Stitch strings
    - Select the shortest resulting string

THE BRUTE FORCE ALGORITHM

- `catt tta tagg gga gagtat`
- `catt tta tagg gga gagtat`
- `cattaggagtat`

LEMMA

Given a finite set of strings $S \subseteq \Sigma^{+}$, the brute force technique finds the shortest superstring.

- See handout.

- So what is the problem with this technique?

THE BRUTE FORCE ALGORITHM

- There are just too many permutations!
- So, $n = 100 \rightarrow 100! \approx 10^{158}$ permutations.
- Testing at 10^{10} permutations/sec, you need
  - $\approx 10^{148}$ seconds
  - $\approx 10^{143}$ days ($\approx 10^{5}$ seconds/day)
  - $\approx 2.7 \times 10^{140}$ years
  - $\approx 2.7 \times 10^{138}$ centuries
- Not bloody likely you will test each permutation before hell freezes over!
  - Even if every subatomic particle in the universe was a processor

PROSPECTS FOR A FASTER ALGORITHM?

- SS belongs to very important class of problems called NP (for Nondeterministic Polynomial).
- For such problems, no algorithm with polynomial work is known.
- But solutions can be verified in polynomial work!
- Wait for 15-451 and 15-453 for the gory details!
- But usually there are approximation algorithms
  - with bounds on the quality of results, and
  - perform better in practice.

SYNOPSIS

- A real world problem: Gene sequencing.
- The computational problem.
- Algorithms:
    - The Brute Force Algorithm
    - Reducing SS to TSP

PROBLEM REDUCTION

- A reduction is a mapping from one problem (A) to another problem (B), so that the solution B problem can be used to solve A.
  - Solving a set of linear equations, reduces to inverting a matrix.
- Map the instance of problem A to an instance of B,
- Solve using algorithms for B
- Map the resulting solution back.

REDUCING SS TO TSP

THE (ASYMMETRIC) TRAVELING SALESPERSON PROBLEM (TSP)

Given a weighted directed graph

- find the shortest path that starts at vertex s, and
- visits each vertex once, and
- returns to s.

- $\equiv$ Hamiltonian path with the lowest total sum of weights
- So, how is this related to SS?

REDUCING SS TO TSP

- If s_i is followed by s_j in how much will the SS length increase?
  - $s_i = tagg$ followed by $s_j = gga \quad \rightarrow tag\underline{ga}$

- General case?
  - $w_{i,j} = |s_j| - overlap(s_i, s_j)$
  - $overlap(\text{"tagg"}, \text{"gga"}) = 2$
  - $|\text{"gga"}| - 2 = 1$

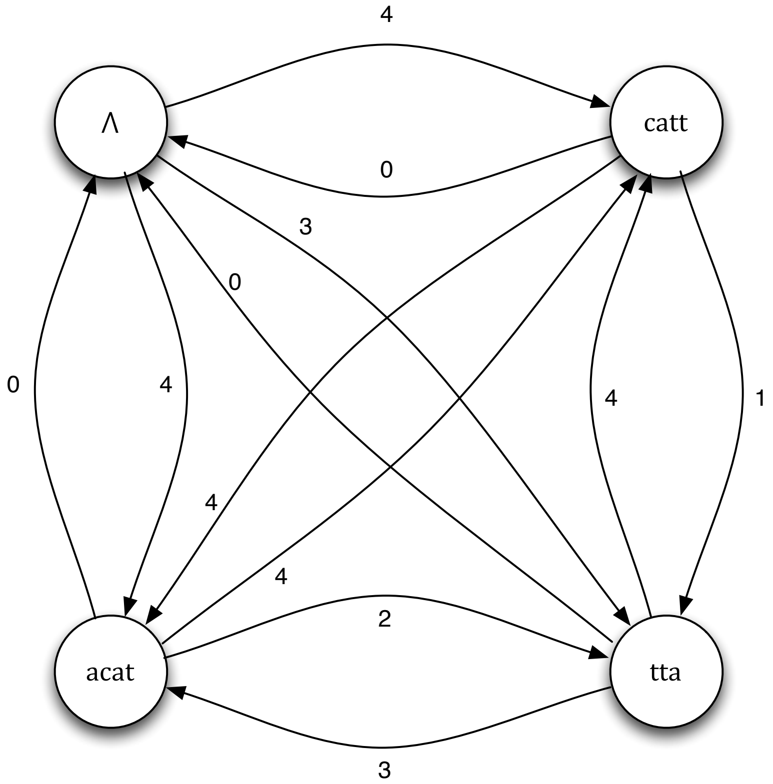
REDUCING SS TO TSP

Build a graph $D = (V, A)$

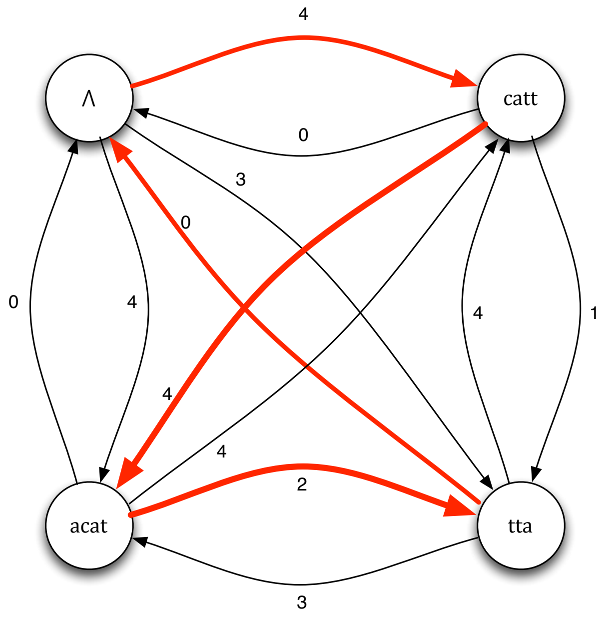
- One vertex for each s_i and one for special “null” node, Λ
- A directed edge from s_i to s_j has weight $w_{i,j} = |s_j| - \mathit{overlap}(s_i, s_j)$
- $w_{\Lambda,i} = |s_i| \rightarrow$ no overlap, maximal increase
- $w_{i,\Lambda} = 0 \rightarrow$, no overlap, no increase

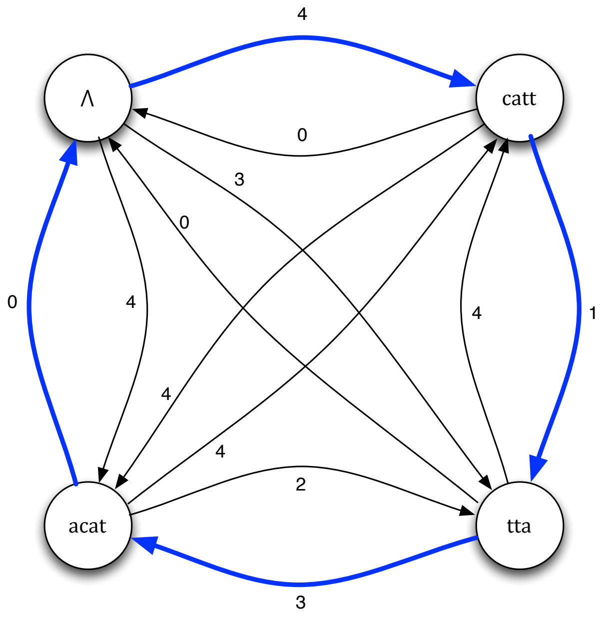
REDUCING SS TO TSP

- $S = \{\texttt{catt}, \texttt{tta}, \texttt{acat}\}$



REDUCING SS TO TSP





- This tour ≡ `cattacatta`
- Length 10

- This tour ≡ `cattacat`
- Length 8

REDUCING SS TO TSP

- TSP considers all Hamiltonian paths (hence is brute force)

- TSP finds the minimum cost Hamiltonian path.
  - Total cost is the length of the SS

- TSP is also NP-hard.

SYNOPSIS

- A real world problem: Gene sequencing.
- The computational problem.
- Algorithms:
  - The Brute Force Algorithm
  - Reducing SS to TSP
  - The Greedy Algorithm

THE GREEDY TECHNIQUE

THE GREEDY TECHNIQUE

Given a sequence of steps to be made, at each decision point

- make a locally optimal decision
- without ever backtracking on previous decisions.

- Greedy is a quite general algorithmic paradigm.
- In general, it does not get the best solution.
  - But it does work for some other problems (e.g., Huffman Encoding, MST)

THE GREEDY APPROXIMATION TO SS

- Start with a pair of strings with maximal overlap (Why?)
- Continue with strings that adds the least extension every time.
    - This is the locally optimal decision!

    - We already defined $overlap(s_i, s_j)$
    - $join(s_i, s_j) \equiv$ concatenate s_j to s_i and remove overlap.
        - $join(\text{"tagg"}, \text{"gga"}) = \text{"tagga"}$

THE GREEDY APPROXIMATION TO SS

GREEDYAPPROXSS

```
fun greedyApproxSS($S$) $=$
  if $|S| = 1$ then $s_0$
  else let
    val $O = \{(\texttt{overlap}(s_i, s_j), s_i, s_j) : s_i \in S, s_j \in S, s_i \neq s_j\}$
    val $(o, s_i, s_j) = \texttt{maxval} <_{\#1} O$
    val $s_k = \texttt{join}(s_i, s_j)$
    val $S' = (\{s_k\} \cup S) \setminus \{s_i, s_j\}$
  in
    greedyApproxSS($S'$)
  end
```

- `S′` gets smaller by one string after each recursion.

THE GREEDY APPROXIMATION TO SS

- `GreedyApproxSS` returns a string with length within 3.5 times the shortest string.
- Conjectured to return within a factor of 2.
- Does much better in practice.

THE GREEDY APPROXIMATION TO SS

- Let's do an example.
- $S = \{\text{catt}, \text{gagtat}, \text{tagg}, \text{tta}, \text{gga},\}$

SUMMARY

- Interfaces vs Implementations
    - Precise interfaces are key.
- The Shortest Superstring Problem
    - The brute-force approach
    - Reduction to TSP
    - Approximate solution using greedy paradigm

15-210
PARALLEL AND SEQUENTIAL ALGORITHMS AND DATA STRUCTURES

LECTURE 2

ALGORITHMIC COST MODELS

SYNOPSIS

- Cost Models
- Parallelism
- Scheduling
- Cost Analysis for the Shortest Super String Problem
    - The Brute Force Algorithm
    - The Greedy Algorithm

COST MODELS

- Sequential: the Random Access Machine (RAM) model
- Parallel: the Parallel RAM model
- Parallel: the 15-210 model
    - Tied to high-level programming constructs – operational semantics
    - Think parallel!

15-210 COST MODEL

- $W(e)$: Work needed to evaluate e
- $S(e)$: Span of the evaluation of e

- Parameterized with relevant problem size measures.

- Asymptotic Models
    - How do algorithms scale to large problems!

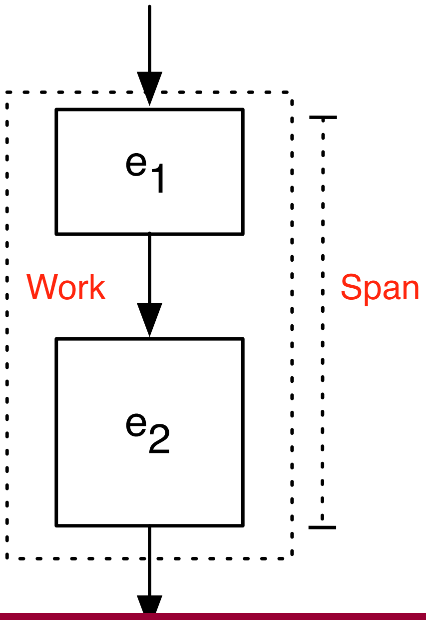
PARAMETERIZATION

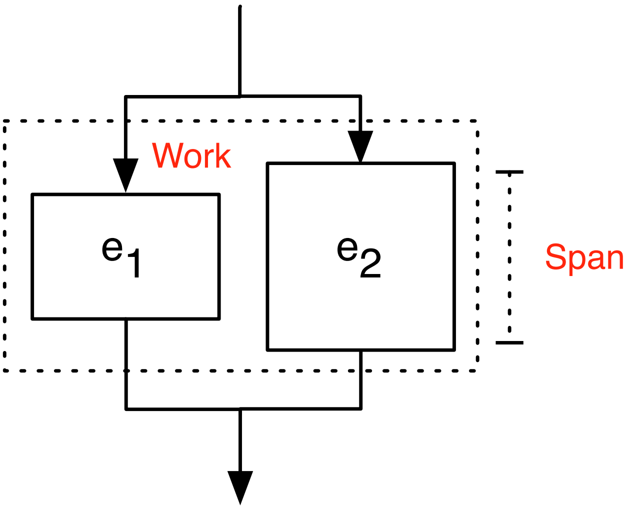
- We measure the size of **representation** of the input.

- Sorting: Number of items to sort
- Map, Reduce: Number of items in the sequence
- Graph Problems: Number of Nodes, Edges
- Searching: Number of items in the database
- Matrix operations: Number of rows and columns
- Prime number testing: Size – number of bits to represent the number (not the value!)
- Computing n^{th} Fibonacci number: Size – number of bits to represent the number (not the value!)

RULES OF COMPOSITION

- (e_1, e_2): Sequential Composition
  - Add work and span
- $e_1 || e_2$: Parallel Composition
  - Add work but take the maximum span





RULES OF COMPOSITION

e	**W**(**e**)	**S**(**e**)
c	1	1
op e	1	1
(e_1, e_2)	$1 + W(e_1) + W(e_2)$	$1 + S(e_1) + S(e_2)$
$(e_1 \| \| e_2)$	$1 + W(e_1) + W(e_2)$	$1 + \max(S(e_1), S(e_2))$
`let val` $x = e_1$ `in` e_2 `end`	$1 + W(e_1) +$ $W(e_2[\mathrm{Eval}(e_1)/x])$	$1 + S(e_1) +$ $S(e_2[\mathrm{Eval}(e_1)/x])$
$\{f(x) \mid x \in A\}$	$1 + \sum_{x \in A} W(f(x))$	$1 + \max_{x \in A} S(f(x))$

RULES OF COMPOSITION

- $\{f(x) \mid x \in A\} \equiv \texttt{map}\ f\ A$

$$\bullet W(\texttt{map}\ f\ \langle\, s_0, \ldots, s_{n-1}\,\rangle) = 1 + \sum_{i=0}^{n-1} W(f(s_i))$$

$$\bullet S(\texttt{map}\ f\ \langle\, s_0, \ldots, s_{n-1}\,\rangle) = 1 + \max_{i=0}^{n-1} S(f(s_i))$$

UPPER AND LOWER BOUNDS

- Upper bound: The maximum asymptotic work (and span) that a given algorithm needs for all inputs of size n.

- Lower bound: The minimum asymptotic work (and span) that any algorithm for a problem needs for all inputs of size n.

SYNOPSIS

- Cost Models
- Parallelism
- Scheduling
- Cost Analysis for the Shortest Super String Problem
  - The Brute Force Algorithm
  - The Greedy Algorithm

PARALLELISM

- For a given W and S, what is the maximum number of processors you can utilize?
  - $\mathbb{P} = \frac{W}{S}$
- Why?
- Mergesort has $W = \theta(n \log n)$ and $S = \theta(\log^2 n)$
- $\mathbb{P} = \theta(\frac{n}{\log n})$
  - The larger the problem is, the higher the parallelism

DESIGNING PARALLEL ALGORITHMS

- Keep work as low as possible
  - No unnecessary computation

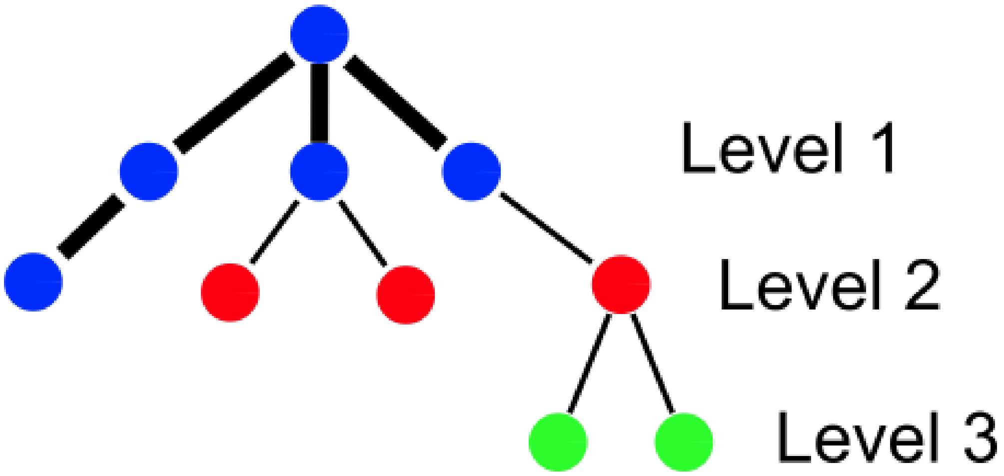
- Keep span as low as possible
  - Hence get high-parallelism

SYNOPSIS

- Cost Models
- Parallelism
- Scheduling
- Cost Analysis for the Shortest Super String Problem
    - The Brute Force Algorithm
    - The Greedy Algorithm

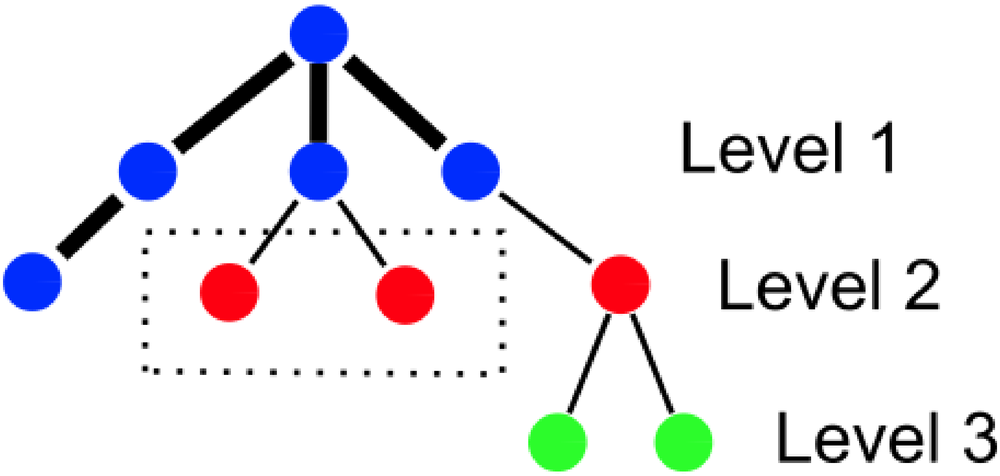
UNDER THE HOOD: TASK SCHEDULING

- Mapping from a computation graph to processors



GREEDY SCHEDULING

- A greedy scheduler will schedule a ready task on an available processor.



A LOWER BOUND

- Let T_p be the "time" needed when using p processors,

$$\max(\frac{W}{p}, S) \leq T_p$$

- Why?

AN UPPER BOUND

- With p processors

$$T_p < \frac{W}{p} + S$$

- Why?

TYING THINGS TOGETHER

- Speed-up is $\frac{W}{T_p}$
  - Maximum possible speed-up is p.

$$\begin{aligned} T_p &< \frac{W}{p} + S \\ &= \frac{W}{p} + \frac{W}{\mathbb{P}} \\ &= \frac{W}{p}\left(1 + \frac{p}{\mathbb{P}}\right) \end{aligned}$$

- $\mathbb{P} \gg p \rightarrow$ near perfect parallelism

SYNOPSIS

- Cost Models
- Parallelism
- Scheduling
- Cost Analysis for the Shortest Super String Problem
  - The Brute Force Algorithm
  - The Greedy Algorithm

COSTS FOR THE BRUTE FORCE SS ALGORITHM

- The brute-force algorithm
  - For each permutation
    - Remove overlaps
    - Stitch strings
  - Output (one of) the shortest string(s)

- $\text{overlap}(s_i, s_j)$ will be needed many times.
  - Preprocess S once and store overlaps as a table
    - What prefix to remove
    - Increase in length

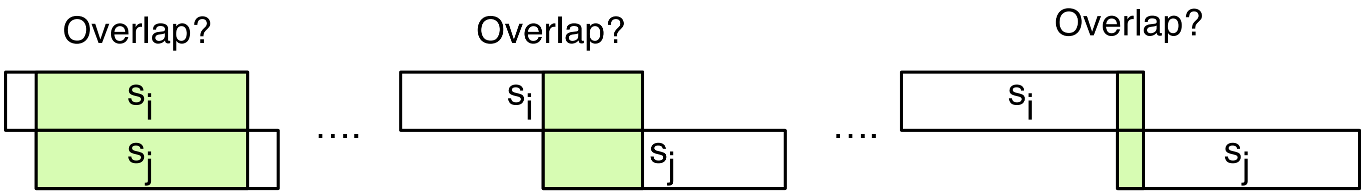
PREPROCESSING – INPUTS

- A set S is n strings, $s_1, s_2, \cdots, s_n$
- Define

$$m = \sum_{i=1}^{n} |s_i|$$

and observe $n \leq m$.

PREPROCESSING A PAIR



- Work and span for preprocessing one pair, s_i and s_j?
  - $W = O(|s_i| \cdot |s_j|)$ Why?
  - $S = O(\log(|s_i| + |s_j|))$ Why?

PREPROCESSING – WORK

$$
\begin{aligned}
W_{ov} &\leq \sum_{i=1}^{n}\sum_{j=1}^{n} W(\texttt{overlap}(s_i, s_j)) \\
&= \sum_{i=1}^{n}\sum_{j=1}^{n} O(|s_i||s_j|) \\
&\leq \sum_{i=1}^{n}\sum_{j=1}^{n} (k_1 + k_2|s_i||s_j|) \\
&= \sum_{i=1}^{n}\sum_{j=1}^{n} k_1 + \sum_{i=1}^{n}\sum_{j=1}^{n} (k_2|s_i||s_j|) \\
&= k_1 n^2 + k_2 \sum_{j=1}^{n} |s_j| (\sum_{i=1}^{n} |s_i|) = k_1 n^2 + k_2 m^2 \in \mathbf{O(m^2)}
\end{aligned}
$$

PREPROCESSING – SPAN

- All s_i, s_j pairs can be processed in parallel.

$$\begin{aligned} S_{ov} &\le \max_{i=1}^{n} \max_{j=1}^{n} S(\texttt{overlap}(s_i, s_j))) \\ &\in \mathbf{O}(\log \mathbf{m}) \end{aligned}$$

BRUTE FORCE SS ALGORITHM

- Work:
  - $O(n)$ lookups each with $O(1)$ work. Why?
  - $n!$ permutations
  - $O(n \cdot n!) = \mathbf{O((n+1)!)}$
  - W_{ov} can be ignored!
- Span:
  - All permutations can be done in parallel, but!

```
func permutations S =
    if |S| = 1 then {S}
    else
       {append([s], p) :
              s in S, p in permutations(S\s)}
```

  - This has span $O(n)$. Why?
  - S_{ov} can be ignored.

SYNOPSIS

- Cost Models
- Parallelism
- Scheduling
- Cost Analysis for the Shortest Super String Problem
    - The Brute Force Algorithm
    - The Greedy Algorithm

THE GREEDY SS ALGORITHM

```
fun greedyApproxSS($S$) =
  if $|S| = 1$ then $s_0$
  else let
    val $O = \{(\texttt{overlap}(s_i, s_j), s_i, s_j) : s_i \in S, s_j \in S, s_i \neq s_j\}$
    val $(o, s_i, s_j) = \texttt{maxval} <_{\#1} O$
    val $s_k = \texttt{join}(s_i, s_j)$
    val $S' = (\{s_k\} \cup S) \setminus \{s_i, s_j\}$
  in
    greedyApproxSS($S'$)
  end
```

THE GREEDY SS ALGORITHM

```
fun greedyApproxSS($S$) =
  if $|S| = 1$ then $s_0$
  else let
    val $O = \{(\texttt{overlap}(s_i, s_j), s_i, s_j) : s_i \in S, s_j \in S, s_i \neq s_j\}$
    val $(o, s_i, s_j) = \texttt{maxval} <_{\#1} O$
    val $s_k = \texttt{join}(s_i, s_j)$
    val $S' = (\{s_k\} \cup S) \setminus \{s_i, s_j\}$
  in
    greedyApproxSS($S'$)
  end
```

- $W_{ov} = O(m^2),\ S_{ov} = O(\log m)$

THE GREEDY SS ALGORITHM

```
fun greedyApproxSS($S$) =
  if $|S| = 1$ then $s_0$
  else let
    val $O = \{(\texttt{overlap}(s_i, s_j), s_i, s_j) : s_i \in S, s_j \in S, s_i \neq s_j\}$
    val $(o, s_i, s_j) = \texttt{maxval} <_{\#1} O$
    val $s_k = \texttt{join}(s_i, s_j)$
    val $S' = (\{s_k\} \cup S) \setminus \{s_i, s_j\}$
  in
    greedyApproxSS($S'$)
  end
```

- $W_{maxval} = O(m^2), \; S_{maxval} = O(\log m)$
- Why?

THE GREEDY SS ALGORITHM

```
fun greedyApproxSS($S$) =
  if $|S| = 1$ then $s_0$
  else let
    val $O = \{(\texttt{overlap}(s_i, s_j), s_i, s_j) : s_i \in S, s_j \in S, s_i \neq s_j\}$
    val $(o, s_i, s_j) = \texttt{maxval} <_{\#1} O$
    val $s_k = \texttt{join}(s_i, s_j)$
    val $S' = (\{s_k\} \cup S) \setminus \{s_i, s_j\}$
  in
    greedyApproxSS($S'$)
  end
```

- No more than $W = O(m^2)$, $S = O(\log m)$
- Why?

THE GREEDY SS ALGORITHM

```
fun greedyApproxSS($S$) =
  if $|S| = 1$ then $s_0$
  else let
    val $O = \{(\texttt{overlap}(s_i, s_j), s_i, s_j) : s_i \in S, s_j \in S, s_i \neq s_j\}$
    val $(o, s_i, s_j) = \texttt{maxval} <_{\#1} O$
    val $s_k = \texttt{join}(s_i, s_j)$
    val $S' = (\{s_k\} \cup S) \setminus \{s_i, s_j\}$
  in
    greedyApproxSS($S'$)
  end
```

- At most n (sequential) calls to `greedyApproxSS`
  - Each with $W = O(m^2)$, $S = O(\log m)$
- $W_{greedy} = O(nm^2)$ and $S_{greedy} = O(n \log m)$
- Why?

SUMMARY

- Cost Models: Rules of Composition
- Parallelism and Scheduling
- Cost Analysis for the Shortest Super String Problem
    - Preprocessing for overlaps
    - The Brute Force Algorithm
    - The Greedy Algorithm

15-210
PARALLEL AND SEQUENTIAL ALGORITHMS AND DATA STRUCTURES

LECTURE 3

ALGORITHMIC TECHNIQUES AND DIVIDE-AND-CONQUER

SYNOPSIS

- Algorithmic Techniques
- Divide-and-Conquer
    - Analysis of Costs
- The Maximum Contiguous Subsequence Sum Problem

ALGORITHMIC TECHNIQUES

- Brute Force
  - Try all possibilities
  - Almost always intractable
  - Useful for testing small cases
  - Code usually easy to write

- Reducing one problem to another
  - Transform the structure or the instance of a problem.
  - Shortest Superstring → Traveling Salesperson Problem
  - Apply algorithms for the new problem

INDUCTIVE TECHNIQUES

- Solve one or more smaller problems to solve the large problem.
- Techniques differ on
  - The number of subproblems
  - How subproblem solutions are used

- Divide-and-Conquer
- Greedy
- Contraction
- Dynamic Programming

DIVIDE-AND-CONQUER

- Divide a problem of size n into $k > 1$ problems
  - Sizes $n_1, n_2, \ldots, n_k$
- Solve each problem recursively.
- Combine the subproblem solutions.

GREEDY

- Given a problem of size n
- Remove one (or more) elements using a greedy approach
  - Smallest, two smallest, nearest, lowest, etc.
- Solve the remaining smaller problem
  - Usually smaller by 1 or 2 items.

CONTRACTION

- Given a problem of size n
- Generate a significantly smaller (contracted) instance
    - e.g., of size $n/2$
- Solve the smaller instance
- Use the result to solve the original problem.
- One recursive call instead of multiple!

DYNAMIC PROGRAMMING

- Like Divide-and-Conquer
- Solutions to subproblems used multiple times!
- Compute once and store, and then reuse.

ADTS AND DATA STRUCTURES

- Techniques rely on Abstract Data Types (for functionality)
  - and on data structures that implement them (for costs)

- Sequences, Sets, Tables, Priority Queues, Graphs, Trees, . . .

RANDOMIZATION

- Introduce randomness at a choice point
  - Quicksort: choose a pivot randomly

- Testing for primality
  - Miller-Rabin primality test
  - $3/4$ of numbers $< n$ are "witnesses" to n's compositeness.
  - Randomly choose 100 numbers $< n$
  - $P(\text{Failing to find a witness}) = 1 - (\frac{1}{4})^{100}$
  - $P(n \text{ is prime}) = 1 - (\frac{1}{4})^{100} = 0.9999 \ldots 9327 \ldots$

SYNOPSIS

- Algorithmic Techniques
- Divide-and-Conquer
    - Analysis of Costs
- The Maximum Contiguous Subsequence Sum Problem

DIVIDE-AND-CONQUER

- Very versatile.
- Easy to implement.
- Parallelizable
- Code follows the structure of a proof.
- Cost reasoning follows code structure.
  - Recurrences

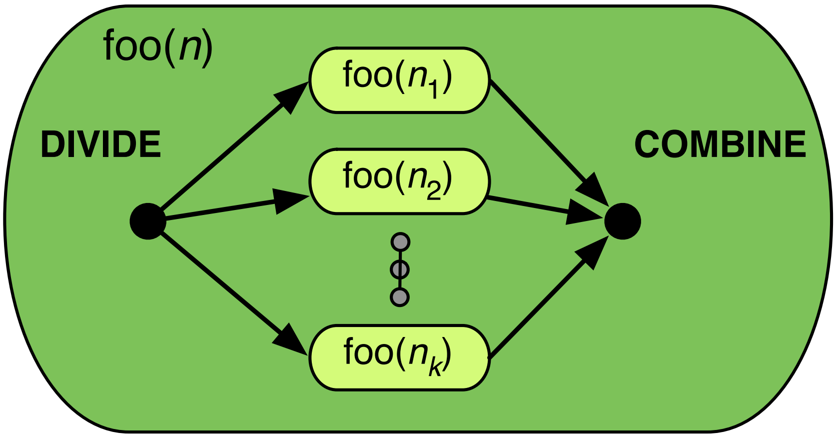
STRENGTENING THE PROBLEM

- Compute more than “superficially” needed.
- No increase to work or span.
- More efficient combine step.
- At the end, this extra information can be discarded.

GENERAL STRUCTURE

- Base case(s)
  - When problem small enough, use a different technique.
  - For example, in quicksort, switch to insertion sort to sort < 30 elements.
- Inductive Step
  - Divide into parts
    - Sometimes quite simple: e.g., mergesort
    - Sometimes a bit tricky: e.g., quicksort
  - Solve subproblems (in parallel)
  - Combine results
    - Sometimes quite simple: e.g., quicksort
    - Sometimes a bit tricky: e.g., mergesort
- Costs can be in the divide or combine steps or in both.

GENERAL STRUCTURE



$$W(n) = W_{\text{divide}}(n) + \sum_{i=1}^{k} W(n_i) + W_{\text{combine}}(n)$$

$$S(n) = S_{\text{divide}}(n) + \max_{i=1}^{k} S(n_i) + S_{\text{combine}}(n)$$

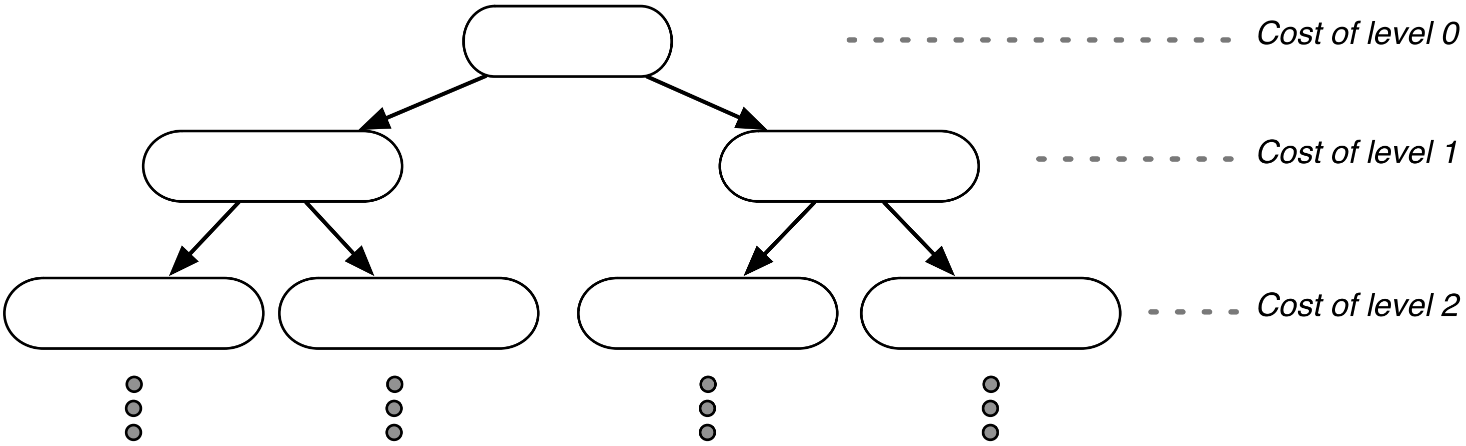
SOLVING RECURRENCES

- Tree method (Brick method)

- Substitution method

THE TREE METHOD

- Expand recurrence into a tree structure.



Cost of level 0

Cost of level 1

Cost of level 2

- Add/Max costs at levels.

THE TREE METHOD

- Solve $W(n) = 2W(n/2) + O(n)$
- In general, solve

$$W(n) = 2W(n/2) + g(n)$$

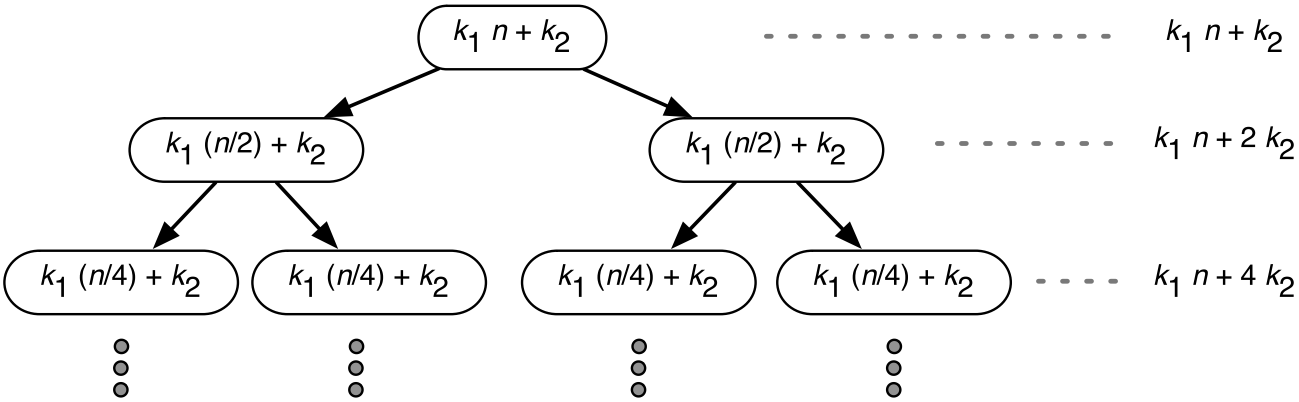
where $g(n) \in O(f(n))$

THE TREE METHOD

- $g(n) \in O(f(n)) \Rightarrow g(n) \leq c \cdot f(n)$
  - For some $c > 0, N_0 > 0$ and $n \geq N_0$
- $g(n) \leq k_1 \cdot f(n) + k_2$ for some k_1, k_2 and $n \geq 1$
  - e.g., $k_1 = c$ and $k_2 = \sum_{i=1}^{N_0} |g(i)|$ (Why?)
- Solve $W(n) \leq 2W(n/2) + k_1 \cdot n + k_2$
  - $f(n) = n$ in our case.

THE TREE METHOD

- Solving $W(n) \leq 2W(n/2) + k_1 \cdot n + k_2$



$k_1\ n + k_2$ $k_1\ n + k_2$

$k_1\ (n/2) + k_2$ $k_1\ (n/2) + k_2$ $k_1\ n + 2\ k_2$

$k_1\ (n/4) + k_2$ $k_1\ (n/4) + k_2$ $k_1\ (n/4) + k_2$ $k_1\ (n/4) + k_2$ $k_1\ n + 4\ k_2$

- Questions:
    - Number of levels in the tree?
    - Problem size at level i?
    - Cost for each node at level i?
    - Number of nodes at level i?
    - Total cost at level i?

THE TREE METHOD

- Total cost at level *i* is at most

$$2^i \cdot \left(k_1 \frac{n}{2^i} + k_2\right) \quad = \quad k_1 \cdot n + 2^i \cdot k_2$$

- Total cost over all levels is

$$\begin{aligned} W(n) &\leq \sum_{i=0}^{\log_2 n} (k_1 \cdot n + 2^i \cdot k_2) \\ &= k_1 n(1 + \log_2 n) + k_2(2^0 + 2^1 + \cdots + 2^{log_2 n}) \\ &\leq k_1 n(1 + \log_2 n) + 2k_2 n \text{ (Why?)} \\ &\in O(n \log n) \end{aligned}$$

THE BRICK METHOD

- Look at the cost structure at the levels of the cost tree
    - Leaves dominated
    - Balanced
    - Root dominated

LEAVES-DOMINATED COST TREES

- For some $\rho > 1$, for all levels *i*

$$\text{cost}_{i+1} \geq \rho \cdot \text{cost}_i$$

$$\begin{array}{c} ++ \\ ++++ \\ ++++++ \\ ++++++++ \end{array}$$

- Overall cost is $O(\text{cost}_d)$ where *d* is the depth.

BALANCED COST TREES

- All levels have about the same cost

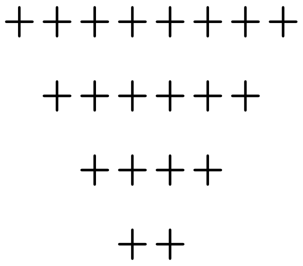
+++++++++
+++++++++
+++++++++
+++++++++

- Overall cost is $O(d \cdot \max_i \text{cost}_i)$ where d is the depth.

ROOT-DOMINATED COST TREES

- For some $\rho < 1$, for all levels i

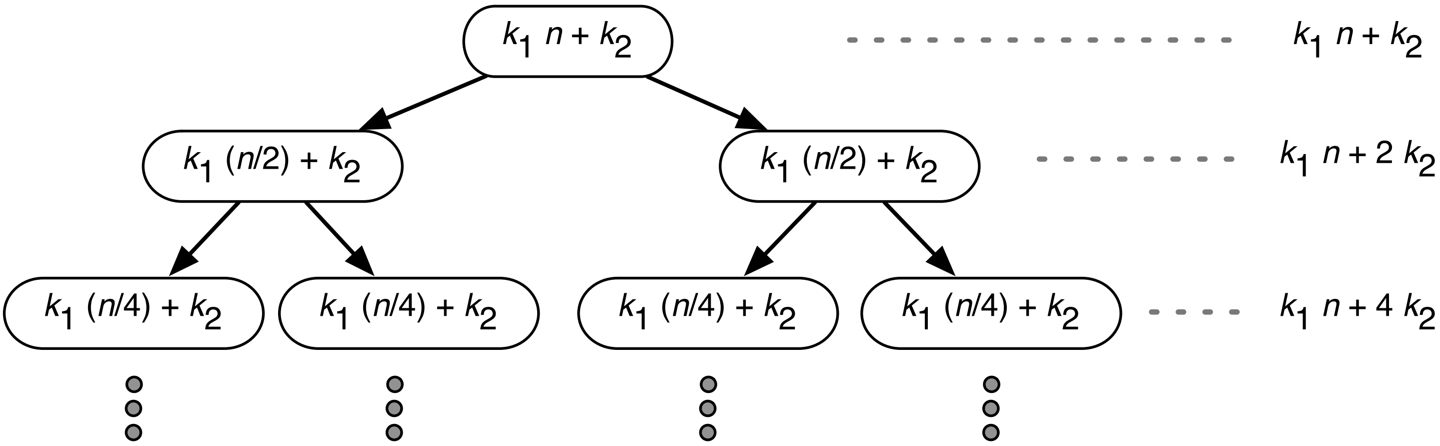
$$\text{cost}_{i+1} \leq \rho \cdot \text{cost}_i$$



- Overall cost is $O(\text{cost}_0)$ where d is the depth.

THE BRICK METHOD

- What type of a cost tree is this?



SYNOPSIS

- Algorithmic Techniques
- Divide-and-Conquer
    - Analysis of Costs
- The Maximum Contiguous Subsequence Sum Problem

THE MCSS PROBLEM

THE MAXIMUM CONTIGUOUS SUBSEQUENCE SUM PROBLEM

- Given a sequence of numbers $S = \langle s_1, \ldots, s_n \rangle$,
- Find

$$\mathrm{mcss}(S) = \max_{1 \leq i \leq j \leq n} \left\{ \sum_{k=i}^{j} s_k \right\}$$

- $S = \langle 0, -1, \mathbf{2}, -\mathbf{1}, \mathbf{4}, -1, 0 \rangle$, $mcss(S) = 5$
- How many possible subsequences are there?
- All positive numbers?

BRUTE FORCE ALGORITHM

- Compute the sum of all $O(n^2)$ possible subsequences (in parallel)
  - Use plus reduce
- Subsequence (i, j) needs
  - $O(j - i)$ work (Why?)
  - $O(\log(j - i))$ span (Why?)

$$
\begin{aligned}
W(n) &= 1 + \sum_{1 \le i \le j \le n} W_{\text{reduce}}(j - i) \le 1 + n^2 \cdot W_{\text{reduce}}(n) \\
&= 1 + n^2 \cdot O(n) \in O(n^3) \\
S(n) &= 1 + \max_{1 \le i \le j \le n} S_{\text{reduce}}(j - i) \le 1 + S_{\text{reduce}}(n) \in O(\log n)
\end{aligned}
$$

BRUTE FORCE ALGORITHM

- Compute maximum over all $O(n^2)$ sums
  - Use max reduce
  - Needs $O(n^2)$ work and $O(\log n)$ span
  - Can be ignored (Why?)

- Total costs for brute force are:
  - $O(n^3)$ work
  - $O(\log n)$ span

DIVIDE-AND-CONQUER – I

$$\langle \text{———} \; L \; \text{———} \parallel \text{———} \; R \; \text{———} \rangle$$

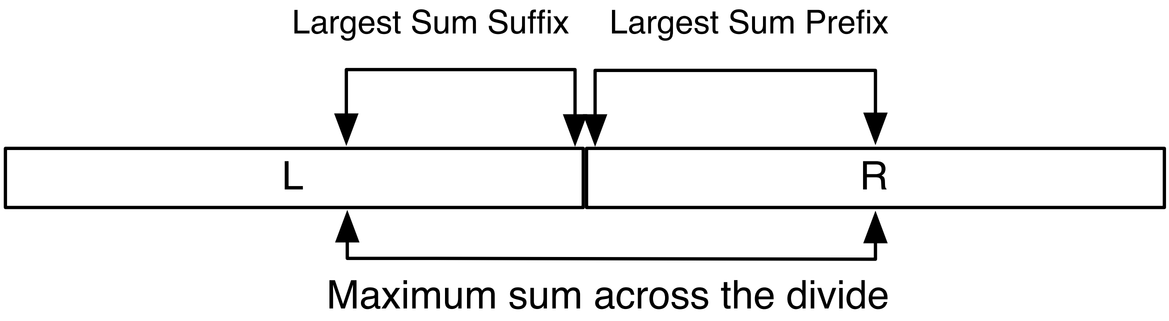
$$\Downarrow$$

$$L = \underbrace{\langle \quad \cdots \quad \rangle}_{\text{mcss}=56} \qquad\qquad R = \underbrace{\langle \quad \ldots \quad \rangle}_{\text{mcss}=17}$$

- Let's solve $S = \langle -2, -1, 2, 3, 2, -2 \rangle$
- Is this right?
- How do we combine subproblem results?

DIVIDE-AND-CONQUER – I

- Recursion handles
  - When mcss(S) subsequence is in the left.
  - When mcss(S) subsequence is in the right.
- What happens when mcss(S) spans across the divide point?



DIVIDE-AND-CONQUER – I

```
fun mcss(s) =
  case (showt s)
    of EMPTY = $-\infty$
     | ELT(x) = x
     | NODE(L, R) =
         let val $(m_L, m_R) = (mcss(L) \parallel mcss(R))$
             val $m_A$ = bestAcross(L, R)
         in max$\{m_L, m_R, m_A\}$
         end
```

- $W(n) = 2W(n/2) + O(n)$ (Why?) $\rightarrow W(n) \in O(n \log n)$
- $S(n) = S(n/2) + O(\log n)$ (Why?) $\rightarrow S(n) \in O(\log^2 n)$

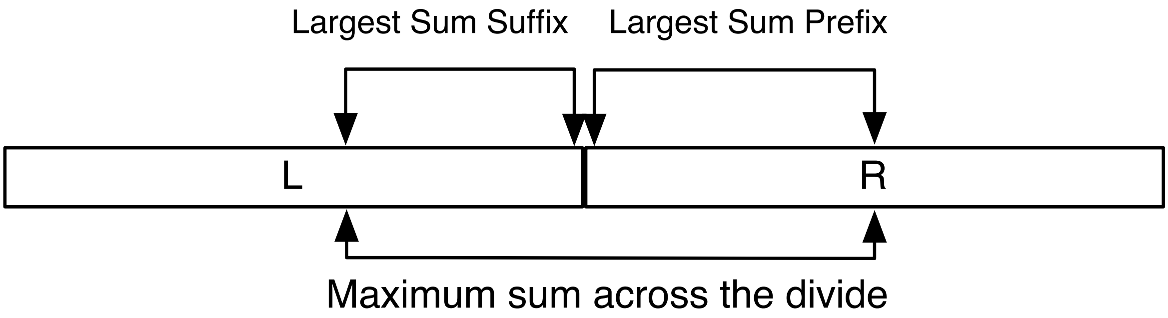
DIVIDE-AND-CONQUER – II

IMPORTANT QUESTIONS

- Can we do better than $O(n \log n)$ work?
- What part of the divide-and-conquer is the bottleneck?
  - Combine takes linear work? (Why?)
- How can we improve?

DIVIDE-AND-CONQUER – II

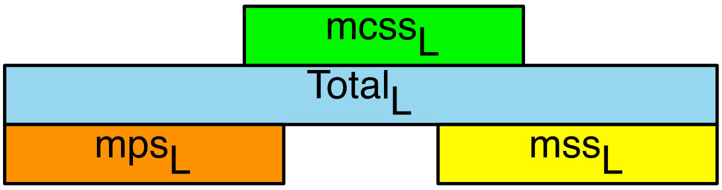
- The answers lie here



- Strengthen the subproblems
  - Compute additional information

DIVIDE-AND-CONQUER – II

Left Subproblem



mcss_L

Total_L

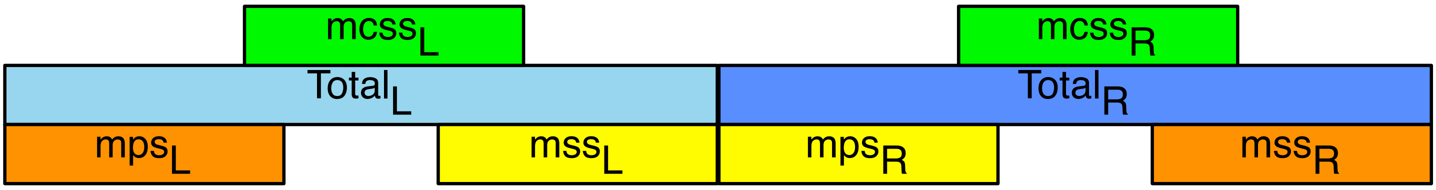
mps_L mss_L

mps = maximum prefix sum mss = maximum suffix sum

DIVIDE-AND-CONQUER – II

Left Subproblem

Right Subproblem



mcss_L

Total_L

mps_L

mss_L

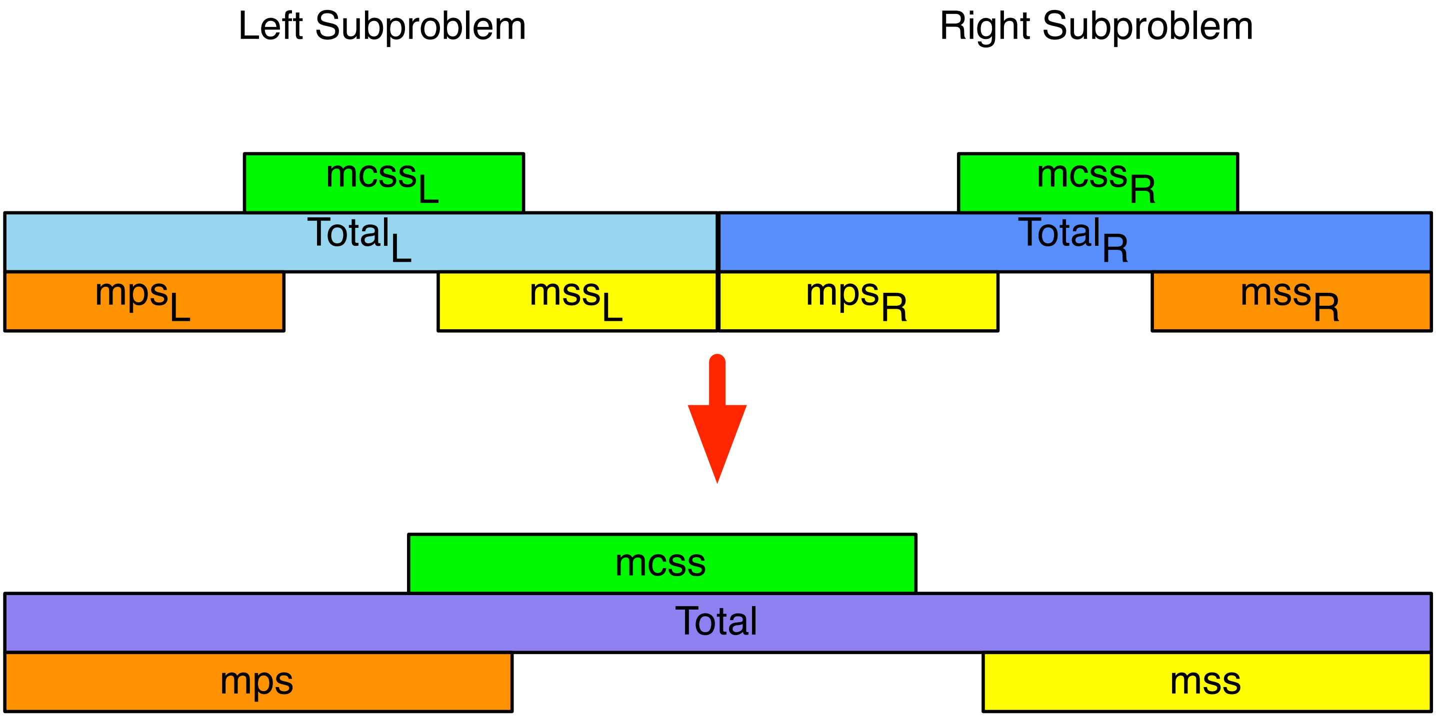
mcss_R

Total_R

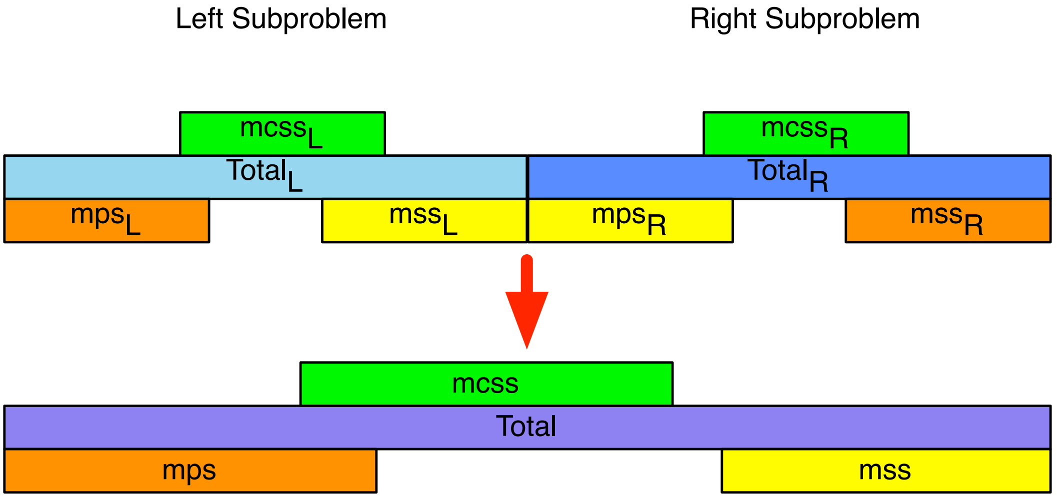
mps_R

mss_R

DIVIDE-AND-CONQUER – II



DIVIDE-AND-CONQUER – II



Left Subproblem

Right Subproblem

mcss_L

Total_L

mps_L

mss_L

mcss_R

Total_R

mps_R

mss_R

mcss

Total

mps

mss

$\text{Total} = \text{Total}_L + \text{Total}_R$

$\text{mcss} = \max\,(\text{mcss}_L, \text{mcss}_R, \text{mss}_L + \text{mps}_R)$

$\text{mps} = \max\,(\text{mps}_L, \text{Total}_L + \text{mps}_R)$

$\text{mss} = \max\,(\text{mss}_L + \text{Total}_R, \text{mss}_R)$

DIVIDE-AND-CONQUER – II

```
**fun** $mcss(a) =$
**let**
  **fun** $mcss'(a)$
    **case** $(showt\ a)$
      **of** $EMPTY = (-\infty, -\infty, -\infty, 0)$
       $|$ $ELT(x) = (x, x, x, x)$
       $|$ $NODE(L, R) =$
          **let**
            **val** $((m_1, p_1, s_1, t_1), (m_2, p_2, s_2, t_2)) = (\,mcss(L)\ \|\ mcss(R)\,)$
          **in**
            $(\max(s_1 + p_2, m_1, m_2),\ \max(p_1, t_1 + p_2),\ \max(s_1 + t_2, s_2),\ t_1 + t_2)$
          **end**
  **val** $(m, p, s, t) = mcss'(a)$
**in** $m$ **end**
```

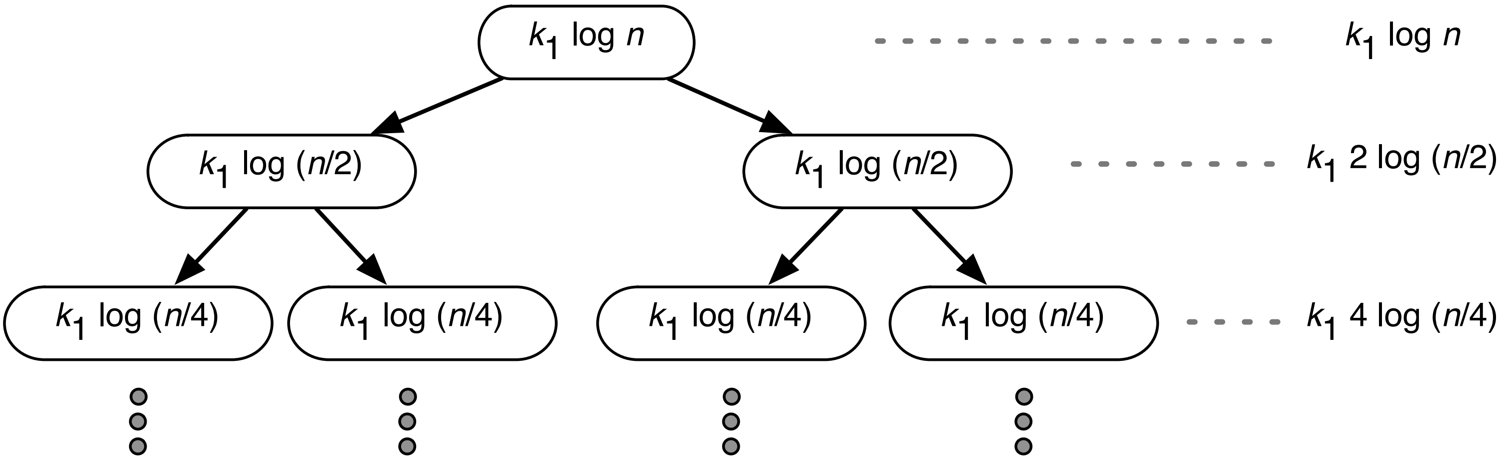
COST ANALYSIS

```
fun mcss(a) =
let
  fun mcss'(a)
    case (showt a)
      of EMPTY = $(-\infty, -\infty, -\infty, 0)$
       | ELT(x) = $(x, x, x, x)$
       | NODE(L, R) =
         let
           val $((m_1, p_1, s_1, t_1), (m_2, p_2, s_2, t_2))$ = ( mcss(L) || mcss(R) )
         in
           $(\max(s_1 + p_2, m_1, m_2),\ \max(p_1, t_1 + p_2),\ \max(s_1 + t_2, s_2),\ t_1 + t_2)$
         end
  val $(m, p, s, t) = mcss'(a)$
in m end
```

- Assuming `showt` has $O(\log n)$ work and span.
  - $W(n) = 2W(n/2) + O(\log n)$
  - $S(n) = S(n/2) + O(\log n)$

COST ANALYSIS

- $W(n) = 2W(n/2) + O(\log n)$



- $W(n) \leq \sum_{i=0}^{\log n} k_1 2^i \log(n/2^i)$

SUBSTITUTION METHOD

- Solve $W(n) \leq 2W(n/2) + k \cdot \log n$
  - $k > 0$
  - $W(n) \leq k$ for $n \leq 1$
- Guess $W(n) \leq \kappa_1 n - \kappa_2 \log n - \kappa_3$
  - Need to find κ_1, κ_2, and κ_3.
- Base case: $W(1) \leq k \Rightarrow \kappa_1 - \kappa_3 \leq k$

SUBSTITUTION METHOD

- Inductive Step

$$
\begin{aligned}
W(n) &\leq 2W(\frac{n}{2}) + k \cdot \log n \\
&\leq 2(\kappa_1 \frac{n}{2} - \kappa_2 \log(\frac{n}{2}) - \kappa_3) + k \cdot \log n \\
&= \kappa_1 n - 2\kappa_2(\log n - 1) - 2\kappa_3 + k \cdot \log n \\
&= (\kappa_1 n - \kappa_2 \log n - \kappa_3) + (k \log n - \kappa_2 \log n + 2\kappa_2 - \kappa_3) \\
&\leq \kappa_1 n - \kappa_2 \log n - \kappa_3
\end{aligned}
$$

- Choose $\kappa_2 = k$ and $2\kappa_2 - \kappa_3 \leq 0$ (Why?)
- For example, $\kappa_2 = k, \kappa_1 = 3k, \kappa_3 = 2k$ satisfies the constraints.

SUMMARY

- Algorithmic Paradigms
- Divide-and-Conquer
  - General Form
  - Cost Analysis
  - Tree and Brick Methods
  - Substitution Method
- Maximum Contiguous Subsequence Problem
  - Brute Force
  - Divide-and-Conquer
  - Divide-and-Conquer with Subproblem Strengthening

15-210
PARALLEL AND SEQUENTIAL ALGORITHMS AND DATA STRUCTURES

LECTURE 4

DIVIDE-AND-CONQUER CONTINUED

SYNOPSIS

- The Euclidian Travelling Salesperson Problem
- Divide-and-Conquer Heuristic Algorithm
- Analysis of Costs

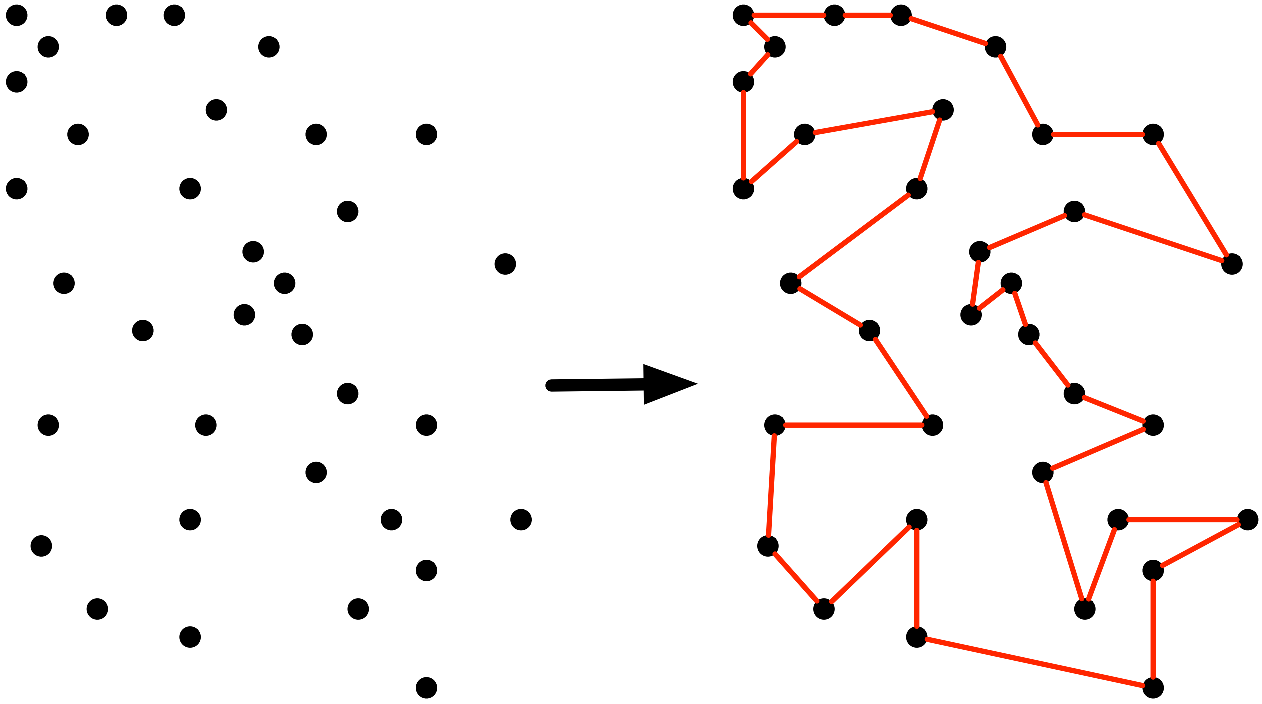
THE EUCLIDIAN TSP

- Given a set of points in a *n*-dimensional Euclidian space.
  - What is a Euclidian space?

- Find the shortest Hamiltonian cycle.
  - What is a Hamiltonian cycle?

- We get a planar Euclidian Traveling Salesperson Problem when the points are in 2-dimensional space.

THE PLANAR TSP



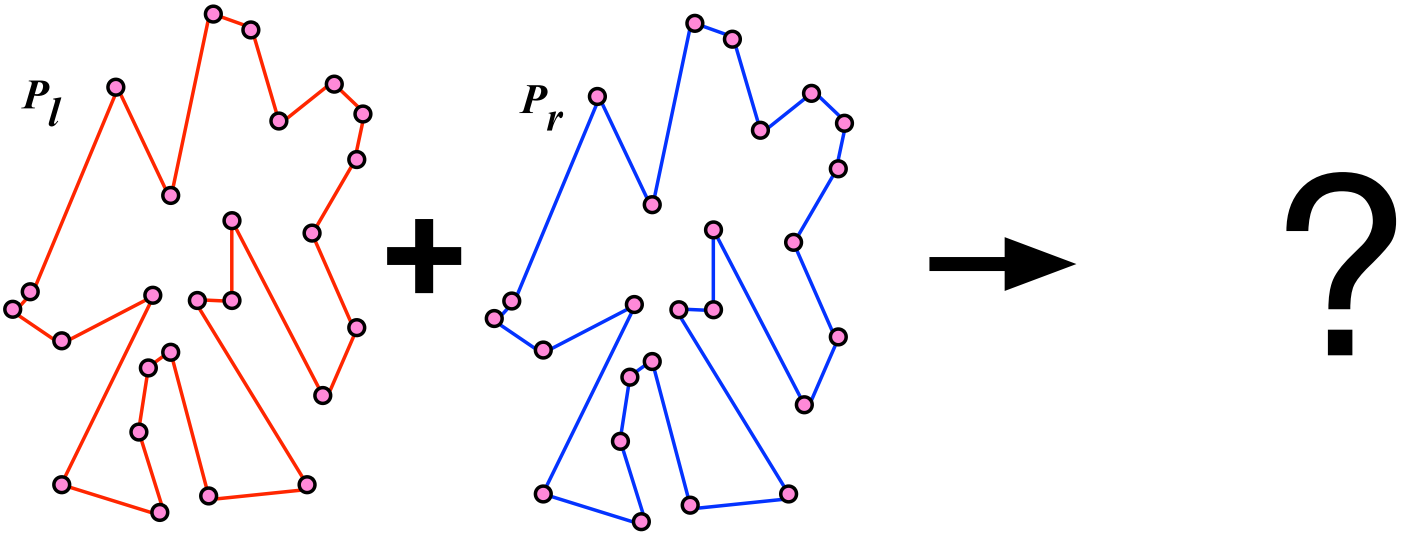
SYNOPSIS

- The Euclidian Travelling Salesperson Problem
- Divide-and-Conquer Heuristic Algorithm
- Analysis of Costs

A DIVIDE-AND-CONQUER HEURISTIC

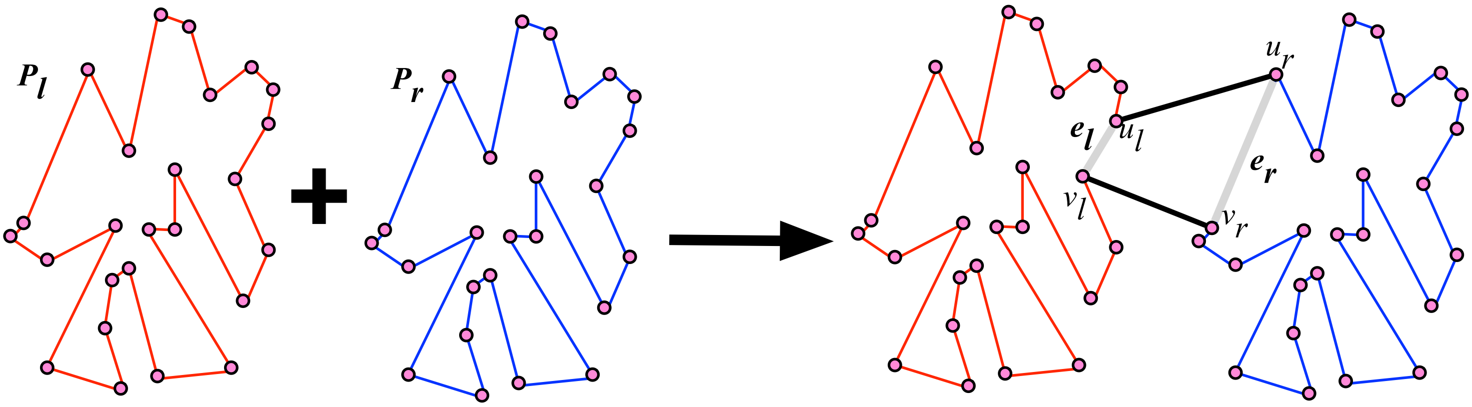
- What is a heuristic?
- Approximation algorithm
  - Resulting tour length is guaranteed to be close to the actual minimum tour length
  - If you spend enough work (but polynomial).
- The Divide-and-Conquer does work both before and after the recursive calls.

A DIVIDE-AND-CONQUER HEURISTIC



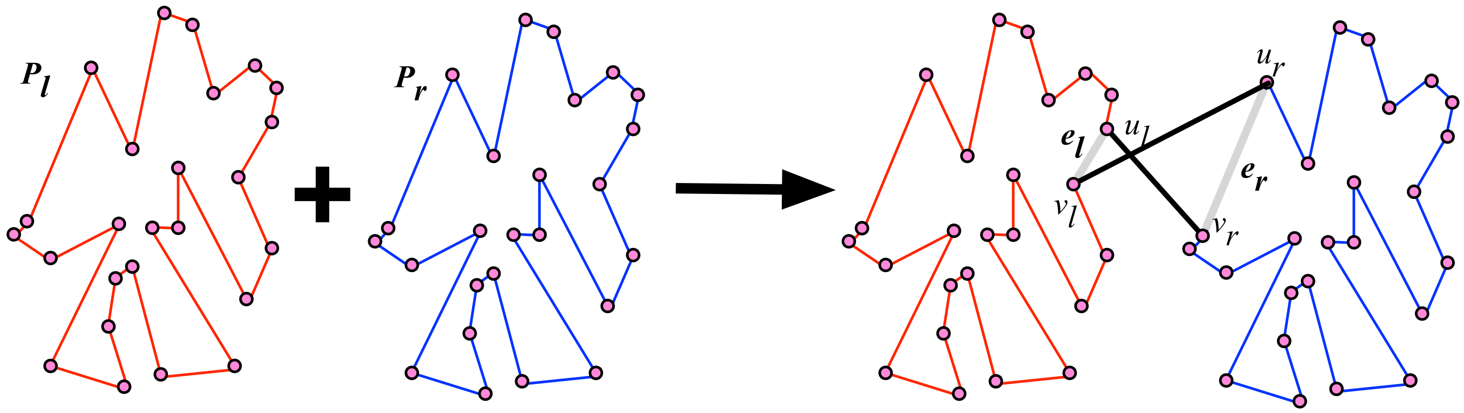
- Assume P_ℓ and P_r have tour lengths T_ℓ and T_r.
- Tour length for the combination?

A Divide-and-Conquer Heuristic



$$T_\ell + T_r + \underbrace{\|u_\ell - u_r\| + \|v_\ell - v_r\|}_{\text{Add these}} - \underbrace{\|u_\ell - v_\ell\| - \|u_r - v_r\|}_{\text{Subtract these}}$$

A DIVIDE-AND-CONQUER HEURISTIC



$$T_\ell + T_r + \underbrace{\|u_\ell - v_r\| + \|v_\ell - u_r\|}_{\text{Add these}} - \underbrace{\|u_\ell - v_\ell\| - \|u_r - v_r\|}_{\text{Subtract these}}$$

A DIVIDE-AND-CONQUER HEURISTIC

- Try all pairs of edges e_ℓ from P_ℓ and e_r from P_r
    - How many pairs are there?
- For each pair of edges, find the smallest increase.
- Then combine the small tours into a large tour.

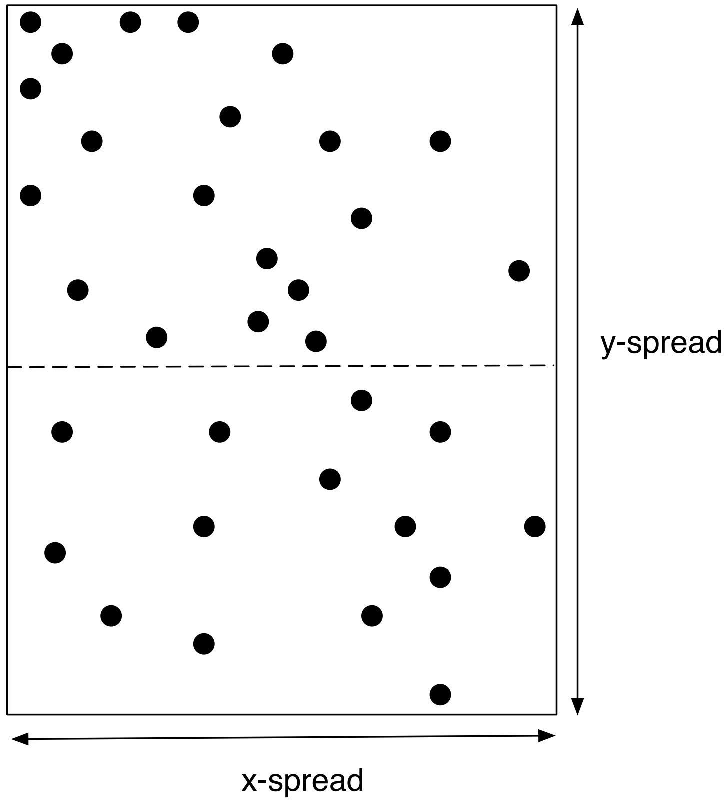
A Divide-and-Conquer Heuristic

```
fun $\mathtt{eTSP}(P) =$
   case $(|P|)$
      of $0, 1 \Rightarrow$ raise $\mathit{TooSmall}$
       | $2 \Rightarrow \{(P[0], P[1]), (P[1], P[0])\}$
       | $n \Rightarrow$ let
                  val $(\mathtt{P}_\ell, \mathtt{P}_r) = \mathtt{splitLongestDim}(P)$
                  val $(\mathtt{L}, \mathtt{R}) = (\,\mathtt{eTSP}(P_\ell) \;\|\; \mathtt{eTSP}(P_r)\,)$
                  val $(c, (e'_\ell, e'_r)) =$
                    $\mathtt{minval}\ <_{\#1} \{(\mathtt{swapCost}(e_\ell, e_r), (e_\ell, e_r)) :$
                                        $e_\ell \in L, e_r \in R\}$
                  in
                    $\mathtt{swapEdges}(\mathtt{append}(L, R), e'_\ell, e'_r)$
         end
```

A Divide-and-Conquer Heuristic

```
fun $eTSP(P) =$
   case $(|P|)$
      of $0, 1$ $\Rightarrow$ raise $TooSmall$
       | $2$ $\Rightarrow$ $\{(P[0], P[1]), (P[1], P[0])\}$
       | $n$ $\Rightarrow$ let
                   val $(P_\ell, P_r) = splitLongestDim(P)$
                   val $(L, R) = (\, eTSP(P_\ell) \;\|\; eTSP(P_r)\,)$
                   val $(c, (e'_\ell, e'_r)) =$
                    $minval <_{\#1} \{(swapCost(e_\ell, e_r), (e_\ell, e_r)) :$
                                        $e_\ell \in L, e_r \in R\}$
                   in
                    $swapEdges(append(L, R), e'_\ell, e'_r)$
         end
```

SPLITTING THE POINTS



- Split at the median along the longer spread dimension.

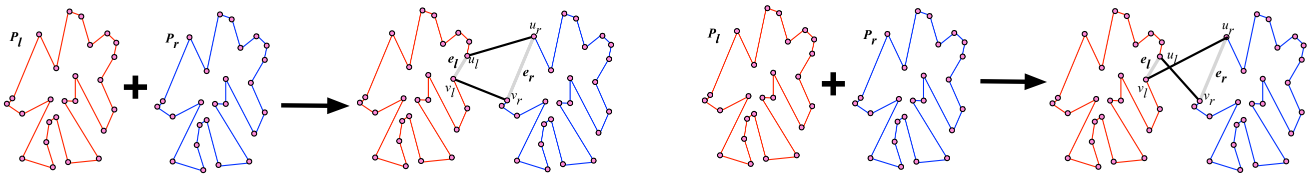
Swap Cost

- Given $e_\ell = (u_\ell, v_\ell) \in L$ and $e_r = (u_r, v_r) \in R$

$$\text{swapCost}((u_\ell, v_\ell), (u_r, v_r)) = \text{Cost Added} - \text{Cost Removed}$$

$$\begin{aligned}\text{Cost Added} = \min(&\|u_\ell - u_r\| + \|v_\ell - v_r\|\,,\\ &\|u_\ell - v_r\| + \|v_\ell - u_r\|)\end{aligned}$$

$$\text{Cost Removed} = \|u_\ell - v_\ell\| + \|u_r - v_r\|$$



Swapping Edges

- `swapEdges(append(L,R),e'`$_\ell$`,e'`$_r$`)`
- Appends the Tour edge lists from subproblems
- Then removes and adds appropriate edges.

SYNOPSIS

- The Euclidian Travelling Salesperson Problem
- Divide-and-Conquer Heuristic Algorithm
- Analysis of Costs

COST ANALYSIS

```
fun $eTSP(P) =$
  case $(|P|)$
  of $0, 1 \Rightarrow$ raise $TooSmall$
   | $2 \Rightarrow \{(P[0], P[1]), (P[1], P[0])\}$
   | $n \Rightarrow$ let
         val $(P_\ell, P_r) = splitLongestDim(P)$   $O(n)$ work $O(\log n)$ span (Why?)
         val $(L, R) = (eTSP(P_\ell) \parallel eTSP(P_r))$   $2W(n/2)$ work $S(n/2)$ span
         val $(c, (e'_\ell, e'_r)) =$
           $minval <_{\#1} \{(swapCost(e_\ell, e_r), (e_\ell, e_r)) :$
                    $e_\ell \in L, e_r \in R\}$   $O(n^2)$ work $O(\log n)$ span (Why?)
           in
             $swapEdges(append(L, R), e'_\ell, e'_r)$   $O(\log n)$ span (Why?)
       end
```

COST ANALYSIS

$$W(n) = 2W(n/2) + O(n^2)$$

$$S(n) = S(n/2) + O(\log n)$$

$$S(n) \in O(\log^2 n)$$

COST ANALYSIS

- Solve (directly)

$$W(n) = 2W(n/2) + k \cdot n^{1+\varepsilon}$$

for constant $\varepsilon > 0$.

  - Depth is $\log_2 n$ (Is this technically right?)
  - At level i, we have 2^i nodes each costing $k \cdot (n/2^i)^{1+\varepsilon}$

$$\begin{aligned} W(n) &= \sum_{i=0}^{\log n} k \cdot 2^i \cdot \left(\frac{n}{2^i}\right)^{1+\varepsilon} \\ &= k \cdot n^{1+\varepsilon} \cdot \sum_{i=0}^{\log n} 2^{-i \cdot \varepsilon} \\ &\leq k \cdot n^{1+\varepsilon} \cdot \sum_{i=0}^{\infty} 2^{-i \cdot \varepsilon} \\ W(n) &\in O(n^{1+\varepsilon})(\text{ Why?}) \end{aligned}$$

SUMMARY

- Euclidian Traveling Salesperson Problem
  - Divide-and-Conquer Heuristic
  - Processing before and after the subproblem solutions.
- Cost Analysis

15-210
PARALLEL AND SEQUENTIAL ALGORITHMS AND DATA STRUCTURES

LECTURE 5

DATA ABSTRACTION AND SEQUENCES

SYNOPSIS

- Abstractions and Implementations
  - Meldable Priority Queues
- The Sequence ADT
- The `scan` operation
- Introduction to contraction

ABSTRACTIONS AND IMPLEMENTATIONS

	Abstraction	Implementation
Functions	Problem	Algorithm
Data	Abstract Data Type	Data Structure

MELDABLE PRIORITY QUEUES

- Priority Queues
  - Insert an item – `insert`
  - Return and delete the item with the minimum priority – `deleteMin`
- Meldable Priority Queue
  - Join two priority queues into one – `meld`

MELDABLE PRIORITY QUEUES

- $\mathbb{S}$ is a totally ordered set (integers, strings, reals, $\ldots$).
- $\mathbb{T}$ is a type representing *subsets* of $\mathbb{S}$.

$$
\begin{array}{lcll}
\texttt{empty} & : & \mathbb{T} & = \{\} \\
\texttt{insert}(S, e) & : & \mathbb{T} \times \mathbb{S} \to \mathbb{T} & = S \cup \{e\} \\
\texttt{deleteMin}(S) & : & \begin{array}{r} \mathbb{T} \to \mathbb{T} \times \\ (\mathbb{S} \cup \{\bot\}) \end{array} & = \begin{cases} (S, \bot) & S = \{\} \\ (S \setminus \{\min S\}, \min S) & \textit{otherwise} \end{cases} \\
\texttt{meld}(S_1, S_2) & : & \mathbb{T} \times \mathbb{T} \to \mathbb{T} & = S_1 \cup S_2
\end{array}
$$

MPQ DEFINITION IN SML

```
signature MPQ
 sig
   struct S : ORD
   type t
   val empty : t
   val insert : t * S.t -> t
   val deleteMin : t -> t * S.t option
   val meld : t * t -> t
 end
```

- No semantics, only the types.

MPQ: COST SPECIFICATIONS

- Implementation 1:

Operation	**Work**
`insert`(S, e)	$O(\|S\|)$
`deleteMin`(S)	$O(1)$
`meld`(S_1, S_2)	$O(\|S_1\| + \|S_2\|)$

- What is the underlying data structure? Sorted Array
- `meld` is actually an array merge.

MPQ: COST SPECIFICATIONS

- Implementation 2:

Operation	**Work**
`insert`(S, e)	$O(\log \lvert S \rvert)$
`deleteMin`(S)	$O(\log \lvert S \rvert)$
`meld`(S_1, S_2)	$O(\lvert S_1 \rvert + \lvert S_2 \rvert)$

- What is the underlying data structure? Heaps

MPQ: COST SPECIFICATIONS

- Implementation 3:

Operation	Work
`insert`(S, e)	$O(\log\lvert S\rvert)$
`deleteMin`(S)	$O(\log\lvert S\rvert)$
`meld`(S_1, S_2)	$O(\log(\lvert S_1\rvert + \lvert S_2\rvert))$

- Later!

ABSTRACTIONS AND IMPLEMENTATIONS

- The Abstract Data Type
  - Functionality
  - Correctness
- The Cost Specification
  - Multiple Cost Specifications
  - We only need these to do cost analysis.
- Underlying Data Structure
  - Multiple Data Structures

THE SEQUENCE ADT - SOME BASICS

- A *relation* is a set of ordered pairs.
  - First from set A, second from set B
- A relation $\rho \subseteq A \times B$.
- A *function* is a relation ρ, where for every $a \in A$ there is only one b such that $(a, b) \in \rho$.
- A *sequence* is a function where $A = \{0, \ldots, n-1\}$ for some $n \in \mathbb{N}$.

THE SEQUENCE ADT – FUNCTIONALITY

- A *sequence* is a type $\mathbb{S}_\alpha$ representing functions from $\{0, \ldots, n-1\}$ to α.

$$
\begin{array}{llcl}
\texttt{empty} & : \mathbb{S}_\alpha & = & \{\} \\
\texttt{length}(A) & : \mathbb{S}_\alpha \to \mathbb{N} & = & |A| \\
\texttt{singleton}(v) & : \alpha \to \mathbb{S}_\alpha & = & \{(0, v)\} \\
\texttt{nth}(A, i) & : \mathbb{S}_\alpha \to \alpha & = & A(i) \\
\texttt{map}(f, A) & : (\alpha \to \beta) \times \mathbb{S}_\alpha \to \mathbb{S}_\beta & = & \{(i, f(v)) : (i, v) \in A\} \\
\texttt{tabulate}(f, n) & : (\mathbb{N} \to \alpha) \times \mathbb{N} \to \mathbb{S}_\alpha & = & \{(i, f(i)) : i \in \{0, \ldots, n-1\}\} \\
\texttt{take}(A, n) & : \mathbb{S}_\alpha \times \mathbb{N} \to \mathbb{S}_\alpha & = & \{(i, v) \in A \mid i < n\} \\
\texttt{drop}(A, n) & : \mathbb{S}_\alpha \times \mathbb{N} \to \mathbb{S}_\alpha & = & \{(i-n, v) : (i, v) \in A \mid i \geq n\} \\
\texttt{append}(A, B) & : \mathbb{S}_\alpha \times \mathbb{S}_\alpha \to \mathbb{S}_\alpha & = & A \cup \{(i + |A|, v) : (i, v) \in B\}
\end{array}
$$

THE SEQUENCE ADT – COST SPECS

	ArraySequence *Work*	 *Span*
`length`(T)	$O(1)$	$O(1)$
`nth`(T)	$O(1)$	$O(1)$
`append`(S_1, S_2)	$O(\|S_1\| + \|S_2\|)$	$O(1)$

The Sequence ADT – Cost Specs

	ArraySequence	
	Work	*Span*
`tabulate` $f\ n$	$O\left(\sum_{i=0}^{n} W(f(i))\right)$	$O\left(\max_{i=0}^{n} S(f(i))\right)$
`map` $f\ S$	$O\left(\sum_{s \in S} W(f(s))\right)$	$O\left(\max_{s \in S} S(f(s))\right)$

THE SEQUENCE ADT – COST SPECIFICATIONS

	TreeSequence	
	Work	*Span*
`length`(T)	$O(1)$	$O(1)$
`nth`(T)	$O(\log n)$	$O(\log n)$
`append`(S_1, S_2)	$O(\log(\|S_1\| + \|S_2\|))$	$O(\log(\|S_1\| + \|S_2\|))$

THE SEQUENCE ADT – COST SPECIFICATIONS

	TreeSequence	
	Work	*Span*
`tabulate` f n	$O\left(\sum_{i=0}^{n} W(f(i))\right)$	$O\left(\log n + \max_{i=0}^{n} S(f(i))\right)$
`map` f S	$O\left(\sum_{s \in S} W(f(s))\right)$	$O\left(\log \lvert S \rvert + \max_{s \in S} S(f(s))\right)$

SOME NOTATIONAL CONVENTIONS

S_i	The i^{th} element of sequence S
$\|S\|$	The length of sequence S
$\langle\ \rangle$	The empty sequence
$\langle\, v \,\rangle$	A sequence with a single element v
$\langle\, i, \ldots, j \,\rangle$	A sequence of integers starting at i and ending at $j \geq i$.
$\langle\, e : p \in S \,\rangle$	**Map** the expression e to each element p of sequence S. The same as "`map` (**fn** $p \Rightarrow e$) S" in ML.
$\langle\, p \in S \mid e \,\rangle$	**Filter** out the elements p in S that satisfy the predicate e. The same as "`filter` (**fn** $p \Rightarrow e$) S" in ML.

- More examples are given in the "Syntax and Costs" document.

THE SCAN OPERATION

- Related to `reduce`.

$$\mathtt{scan}\ f\ I\ S : (\alpha \times \alpha \to \alpha) \to \alpha \to \alpha\ \mathtt{seq}$$

$$\to (\alpha\ \mathtt{seq} \times \alpha)$$

- I is the identity value
- f is an (associative) function
- S is a sequence
- Produces $\langle\, I, f(I, S_0), f(f(I, S_0), S_1), \ldots \,\rangle$ and $\mathtt{reduce}\ f\ I\ S$
  - $\mathtt{scan} + 0\,\langle\, 2, 1, 4, 6 \,\rangle = (\langle\, 0, 2, 3, 7 \,\rangle\,, 13)$

THE SCAN OPERATION

- `scan` computes prefix sums.

```
**fun** $\texttt{scan}\ f\ I\ S =$
    $(\langle\, \texttt{reduce}\ f\ I\ (\texttt{take}(S, i)) : i \in \langle\, 0, \ldots n-1 \,\rangle \,\rangle\,,$
      $\texttt{reduce}\ f\ I\ S)$
```

- S has n elements
- Apply `reduce` to each prefix of S of i elements, $0 \leq i \leq n-1$
  - Gives you the $\alpha\ \texttt{seq}$ part
- Apply `reduce` to S
  - Gives you the α part
- So you get $(\alpha\ \texttt{seq} \rightarrow \alpha)$

THE SCAN OPERATION

$$\begin{aligned} \texttt{scan} + 0\ \langle 2,1,3\rangle &= (\langle\ \texttt{reduce} + 0\ \langle\ \rangle, \\ &\qquad \texttt{reduce} + 0\ \langle 2\rangle, \\ &\qquad \texttt{reduce} + 0\ \langle 2,1\rangle\ \rangle \\ &\qquad \texttt{reduce} + 0\ \langle 2,1,3\rangle) \\ &= (\langle 0,2,3\rangle, 6) \end{aligned}$$

- This is obviously not efficient!
- We will see how to do this with

$$\begin{aligned} W(\texttt{scan}\ f\ I\ S) &= O(|S|) \\ S(\texttt{scan}\ f\ I\ S) &= O(\log|S|) \end{aligned}$$

THE INCLUSIVE SCAN OPERATION

- `reduce` all prefixes ending at position i, $0 \leq i \leq n-1$

$$\mathtt{scanI}\ +\ 0\ \langle\, 2, 1, 3\,\rangle\ =\ \langle\, 2, 3, 6\,\rangle$$

USING SCAN IN THE MCSS PROB.

THE MAXIMUM CONTIGUOUS SUBSEQUENCE SUM PROBLEM

- Given a sequence of numbers $S = \langle s_1, \ldots, s_n \rangle$,
- Find

$$\mathrm{mcss}(S) = \max_{1 \le i \le j \le n} \left\{ \sum_{k=i}^{j} s_k \right\}$$

- $S = \langle 0, -1, \mathbf{2}, -\mathbf{1}, \mathbf{4}, -1, 0 \rangle$, $mcss(S) = 5$

USING SCAN IN THE MCSS PROB.

- Consider $S = \langle\, 1, -2, 3, -1, 2, -3 \,\rangle$
- Let $X = \texttt{scanI} + 0\ S = \langle\, 1, -1, 2, 1, 3, 0 \,\rangle$
- What is $X_j - X_i$ for $j > i$?
  - $\sum_{k=i+1}^{j} S_k$
  - $X_4 - X_0 = 3 - 1 = 2$

USING SCAN IN THE MCSS PROB.

- Define R_j as the maximum sum that starts at some i and ends at $j > i$.

$$\begin{aligned}
R_j &= \max_{i=0}^{j} \sum_{k=i}^{j} S_k \\
&= \max_{i=0}^{j} (X_j - X_{i-1}) \\
&= X_j + \max_{i=0}^{j} (-X_{i-1}) \\
&= X_j + \max_{i=0}^{j-1} (-X_i) = X_j - \min_{i=0}^{j-1} X_i \text{ (Why?)}
\end{aligned}$$

USING SCAN IN THE MCSS PROB.

$$R_j = X_j - \min_{i=0}^{j-1} X_i$$

- You need X_j and the minimum previous $X_i, i < j$
  - can be done by a minimum `scan`

$$(M, \_) = \texttt{scan}\ \min\ 0\ X = (\langle 0, 0, -1, -1, -1, -1 \rangle, -1)$$

$$R = \langle X_j - M_j : 0 \leq j < |S| \rangle = \langle 1, -1, 3, 2, 4, 1 \rangle$$

LET'S RECAP

- Given $S = \langle\, 1, -2, 3, -1, 2, -3 \,\rangle$
- We computed X with a $+$ `scanI`.
  - $X = \langle\, 1, -1, 2, 1, 3, 0 \,\rangle$
- We computed M with a `min` `scan`
  - $M = \langle\, 0, 0, -1, -1, -1, -1 \,\rangle$
- We computed $R = \langle\, X_j - M_j : 0 \leq j < |S| \,\rangle$
  - $R = \langle\, 1, -1, 3, 2, 4, 1 \,\rangle$
- A final max `reduce` on R gives us the MCSS, 4.

USING SCAN IN THE MCSS PROB.

```
fun $MCSS(S) =$
let
    val $X = scanI\ +\ 0\ S$
    val $(M, \_) = scan\ \min\ 0\ X$
in
    $\max \langle X_j - M_j : 0 \leq j < |S| \rangle$
end
```

- Work? $O(n)$
- Span? $O(\log n)$

COPY SCAN

- Scan can also be used to pass information along a sequence.

$$\langle\, \mathit{NONE},\ \mathit{SOME}(7),\ \mathit{NONE},\ \mathit{NONE},\ \mathit{SOME}(3),\ \mathit{NONE}\,\rangle$$

$$\downarrow$$

$$\langle\, \mathit{NONE},\ \mathit{NONE},\ \mathit{SOME}(7),\ \mathit{SOME}(7),\ \mathit{SOME}(7),\ \mathit{SOME}(3)\,\rangle$$

- Each element receives the nearest previous `SOME()` value.
- Easy to do sequentially with *`iter`*.

COPY SCAN

- Can we do this with $\mathtt{scan}$?
- $f : \alpha\ \textit{option} \times \alpha\ \textit{option} \rightarrow \alpha\ \textit{option}$

```
fun  $copy(a, b) =$
    case  $b$  of
        $SOME(\_) \Rightarrow b$
      | $NONE \Rightarrow a$
```

- Passes its right argument if it is $SOME$, else passes its left argument.
- How do you show $\mathtt{copy}$ is associative.

IMPLEMENTING SCAN – CONTRACTION

- `scan` looks inherently sequential.
  - Naive implementation needs $O(n^2)$ work.
  - Slightly clever sequential implementation needs $O(n)$ work.
  - Divide an Conquer approaches do not break the sequentiality. (Why?)
- Contraction
  1. Construct a much smaller instance of the problem
  2. Solve the smaller instance recursively
  3. Construct solution to the original instance.

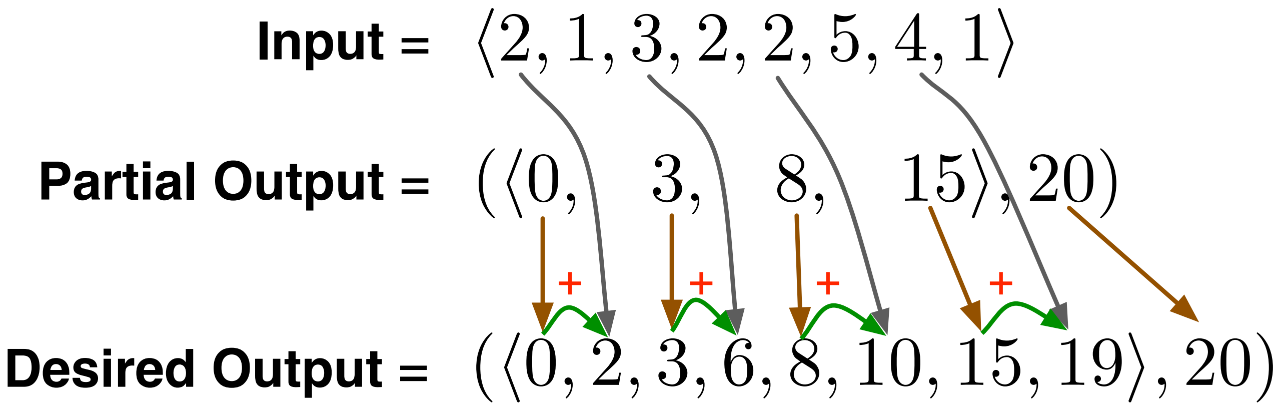
IMPLEMENTING REDUCE WITH CONTRACTION

- Given $\langle\, 2, 1, 3, 2, 2, 5, 4, 1 \,\rangle$
- Apply $+$ pairwise and (in parallel) to get $\langle\, 3, 5, 7, 5 \,\rangle$
  - This is the contracted instance!
- Apply $+$ pairwise to get $\langle\, 8, 12 \,\rangle$
- Finally apply $+$ pairwise to get $\langle\, 20 \,\rangle$
- The 3^{rd} step of the contraction does nothing in this case.

IMPLEMENTING SCAN WITH CONTRACTION

- Given $S = \langle\, 2, 1, 3, 2, 2, 5, 4, 1 \,\rangle$
    - $\mathtt{scan}\ +\ 0\ S = (\langle\, 0, 2, 3, 6, 8, 10, 15, 19 \,\rangle\,, 20)$
- First do pairwise $+$ on S to get $\langle\, 3, 5, 7, 5 \,\rangle$
- Now (recursively) do scan on this to get $(\langle\, 0, 3, 8, 15 \,\rangle\,, 20)$
    - What is the relation to the final scan?
- We have every other element of the final scan!
- How do we fill in the rest?

IMPLEMENTING SCAN WITH CONTRACTION



Input = $\langle 2, 1, 3, 2, 2, 5, 4, 1 \rangle$

Partial Output = $(\langle 0, \quad 3, \quad 8, \quad 15 \rangle, 20)$

Desired Output = $(\langle 0, 2, 3, 6, 8, 10, 15, 19 \rangle, 20)$

Implementing Scan with Contraction

```
% implements: the Scan problem on sequences that have a power of 2 length
fun scanPow2 f i s =
   case $|s|$ of
        $0 \Rightarrow (\langle\rangle, i)$
      | $1 \Rightarrow (\langle i \rangle, s[0])$
      | $n \Rightarrow$
         let
            val $s' = \langle f(s[2i], s[2i+1]) : 0 \leq i < n/2 \rangle$
            val $(r, t) =$ scanPow2 f i $s'$
         in
            $(\langle p_i : 0 \leq i < n \rangle, t)$, where $p_i = \begin{cases} r[i/2] & \textbf{if } even(i) \\ f(r[i/2], s[i-1]) & \textbf{otherwise}. \end{cases}$
         end
```

- General case is in the course notes.

SUMMARY

- Abstractions and Implementations
  - Meldable Priority Queues
- The Sequence ADT
- The *scan* operation
- Introduction to contraction
- Implementing `scan` with contraction.

15-210
PARALLEL AND SEQUENTIAL ALGORITHMS AND DATA STRUCTURES

LECTURE 6

SEQUENCES - II

SYNOPSIS

- The `reduce` operation
- Implementing divide and conquer with `reduce`
    - Implementing MCSS with `reduce`
- Analyzing cost of higher order functions
    - Cost analysis for `reduce`

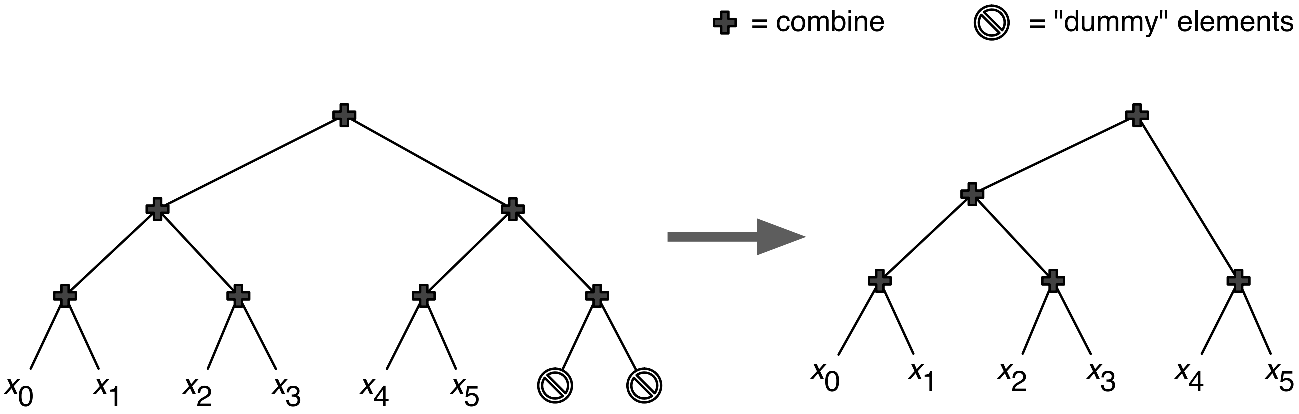
THE REDUCE OPERATION

$$\texttt{reduce}\ f\ I\ S\ :\ (\alpha \times \alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \alpha\ seq \rightarrow \alpha$$

- When *f* is associative, `reduce` returns sum with respect to *f*.
- Same result as `iter` *f I S*
  - `iter` is sequential and allows more general *f* (e.g., $\beta \times \alpha \rightarrow \beta$
- If *f* is not associative, `reduce` and `iter` results may differ.

THE REDUCE OPERATION

- Specific combination based on a perfect binary tree.



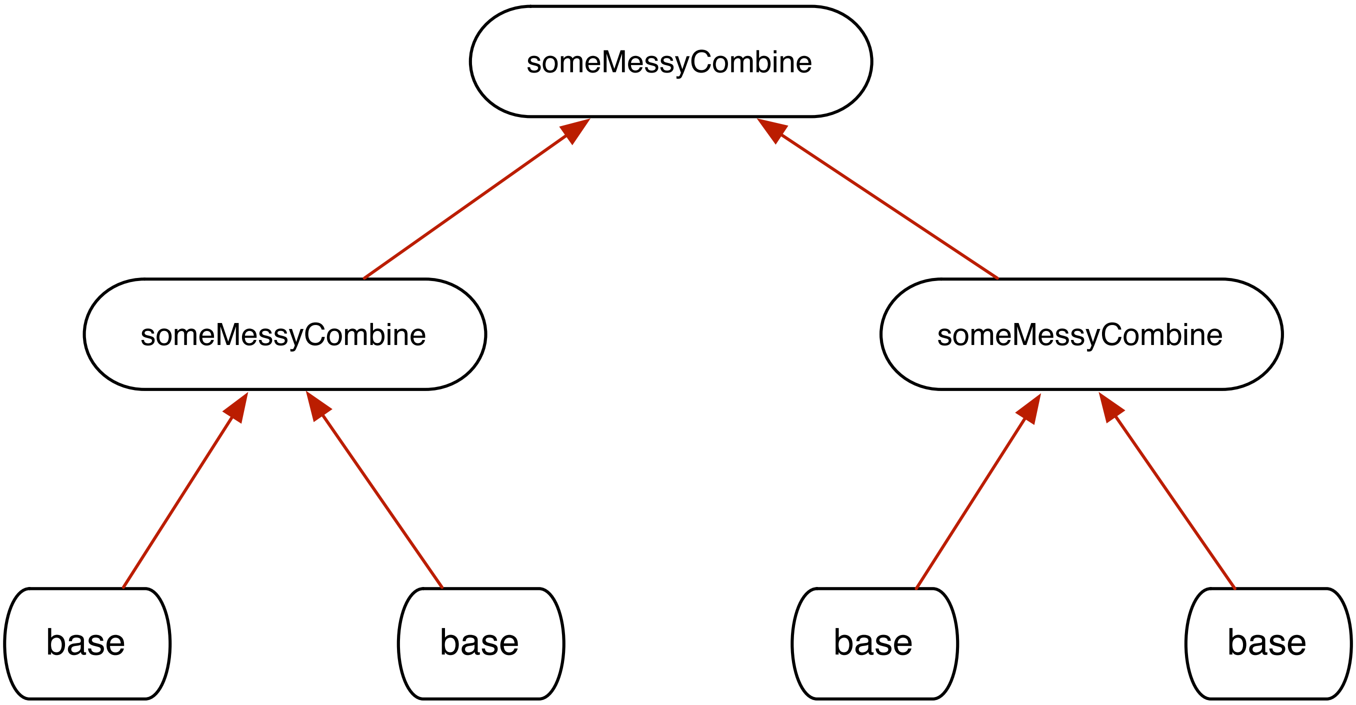
DIVIDE AND CONQUER WITH REDUCE

- Many divide and conquer have the following structure

```
fun myDandC($S$) =
    case showt($S$) of
        EMPTY ⇒ emptyVal
      | ELT($v$) ⇒ base($v$)
      | NODE(L, R) ⇒ let
            val $(L', R') = ($ myDandC($L$) $\|$ myDandC($R$) $)$
        in
            someMessyCombine($L', R'$)
        end
```

- This corresponds to a binary tree combination scheme.

DIVIDE AND CONQUER WITH REDUCE


someMessyCombine

someMessyCombine

someMessyCombine

base

base

base

base

DIVIDE AND CONQUER WITH REDUCE

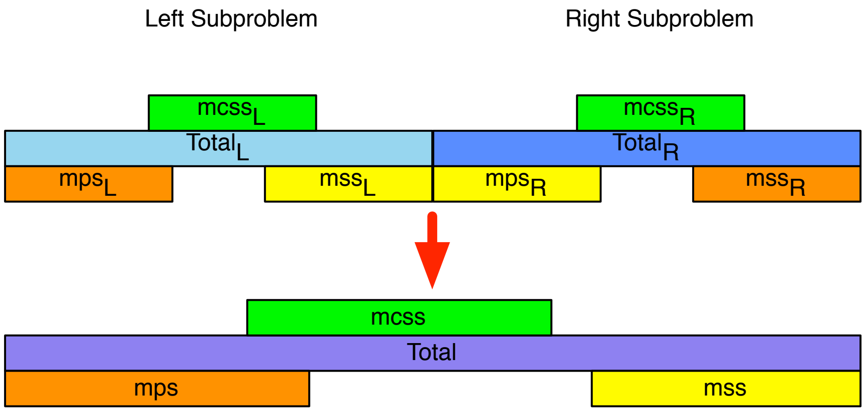
```
fun myDandC(S) =
   case showt(S) of
      EMPTY ⇒ emptyVal
    | ELT(v) ⇒ base(v)
    | NODE(L, R) ⇒ let
         val (L′, R′) = (myDandC(L) || myDandC(R))
      in
         someMessyCombine(L′, R′)
      end
```

↓

```
reduce someMessyCombine emptyVal (map base S)
```

MCSS USING REDUCE

$$\mathrm{mcss}(s) = \max_{1 \le i \le j \le n} \left\{ \sum_{k=i}^{j} s_k \right\}$$



$\mathrm{Total} = \mathrm{Total}_L + \mathrm{Total}_R$

$\mathrm{mcss} = \max(\mathrm{mcss}_L, \mathrm{mcss}_R, \mathrm{mss}_L + \mathrm{mps}_R)$

$\mathrm{mps} = \max(\mathrm{mps}_L, \mathrm{Total}_L + \mathrm{mps}_R)$

$\mathrm{mss} = \max(\mathrm{mss}_L + \mathrm{Total}_R, \mathrm{mss}_R)$

MCSS USING REDUCE

$$\text{mcss}(s) \quad = \quad \max_{1\leq i\leq j\leq n}\left\{\sum_{k=i}^{j} s_k\right\}$$

$$\begin{aligned}&\textbf{fun}\ \ \mathtt{combine}((M_L, P_L, S_L, T_L), (M_R, P_R, S_R, T_R)) = \\ &\quad(\max(S_L + P_R, M_L, M_R), \quad \max(P_L, T_L + P_R), \\ &\qquad\qquad \max(S_R, S_L + T_R), \quad T_L + T_R)\end{aligned}$$

$$\textbf{fun}\ \ \mathtt{base}(v) = (v, v, v, v)$$

$$\textbf{val}\ \ \mathtt{emptyVal} = (-\infty, -\infty, -\infty, 0)$$

$$\begin{aligned}&\textbf{fun}\ \ \mathtt{mcss}(S) = \\ &\quad \mathtt{reduce\ \ combine\ \ emptyVal}\ \ (\mathtt{map\ \ base}\ \ S)\end{aligned}$$

SOME OBSERVATIONS

- Which code to use is a matter of taste
    - `reduce` version is shorter
    - Divide and Conquer version exposes the inductive structure.
- `reduce` version not applicable when split is complicated
    - e.g., in Quicksort

Scan via Reduce

- How do we implement the divide and conquer `scan` with this template?
  - `scan` $f\ I\ S \equiv$ `reduce combine emptyVal` $($`map base` $S)$
- `emptyVal`$=?\ (\langle\ \rangle, I)$
- **fun** `base`$(v) = ?\ (\langle\ I\ \rangle, f(I, v))$
- **fun** `combine` $= ?$

  fun `combine`$((S_1, T_1), (S_2, T_2)) =$
  `append`$(S_1, ($`map` $(fn\ \ x\ \ \Rightarrow\ \ f(x, T_1))\ \ S_2),\ \ f(T_1, T_2))$

  - Is this right?

  fun `combine`$((S_1, T_1), (S_2, T_2)) =$
  $($`append`$(S_1, ($`map` $(fn\ \ x\ \ \Rightarrow\ \ {\color{red} f(T_1, x)})\ \ S_2),\ \ f(T_1, T_2)$

COST OF HIGHER ORDER FUNCTIONS

- We assume that f runs in $O(1)$ work and span.
  - $\Rightarrow$ `reduce` has $O(n)$ work and $O(\log n)$ span
- Easy for `map` and `tabulate`

$$
\begin{aligned}
W(\texttt{map}\ f\ S) &= 1 + \sum_{s \in S} W(f(s)) \\
S(\texttt{map}\ f\ S) &= 1 + \max_{s \in S} S(f(s))
\end{aligned}
$$

- How about `reduce`?

MERGESORT VIA REDUCE

- Remember the reduce template for divide and conquer?

$$\mathtt{reduce}\ \boxed{\mathtt{combine}}\ \boxed{\mathtt{emptyVal}}\ (\mathit{map}\ \boxed{\mathtt{base}}\ S)$$

$$\textbf{val}\ \mathtt{combine} = \mathtt{merge}_{<}$$

$$\textbf{val}\ \mathtt{base} = \mathtt{singleton}$$

$$\textbf{val}\ \mathtt{emptyVal} = \mathtt{empty}()$$

$$\textbf{fun}\ \mathtt{reduceSort}(S) =$$
$$\mathtt{reduce}\ \mathtt{combine}\ \mathtt{emptyVal}\ (\mathit{map}\ \mathtt{base}\ S)$$

COST OF REDUCESORT

- $\mathtt{merge}_<$ is an associative function with costs:

$$\begin{aligned} W(\mathtt{merge}_<(S_1, S_2)) &= O(n_1 + n_2) \\ S(\mathtt{merge}_<(S_1, S_2)) &= O(\log(n_1 + n_2)) \end{aligned}$$

- f has no longer $O(1)$ work and span.
- What is the cost of `reduceSort`.

COST OF REDUCESORT

- For costs, reduction sequence matters!
- Sequential order
- On input $x = \langle\, x_0, x_1, \ldots, x_{n-1} \,\rangle$, we get

$$\texttt{merge}_<(\ldots\ \texttt{merge}_<(\texttt{merge}_<(\texttt{merge}_<(I, \langle\, x_0 \,\rangle), \langle\, x_1 \,\rangle), \langle\, x_2 \,\rangle), \ldots)$$

- Left arg. has length 0 through $n - 1$
- Right arg. always has length 1.
- Work:

$$W(\texttt{reduceSort}\ S) \ \leq\ \sum_{i=0}^{n-1} c \cdot (1 + i) \ \in\ O(n^2) \text{ Why?}$$

MERGESORT WITH REDUCE

- Equivalent to `iter` version

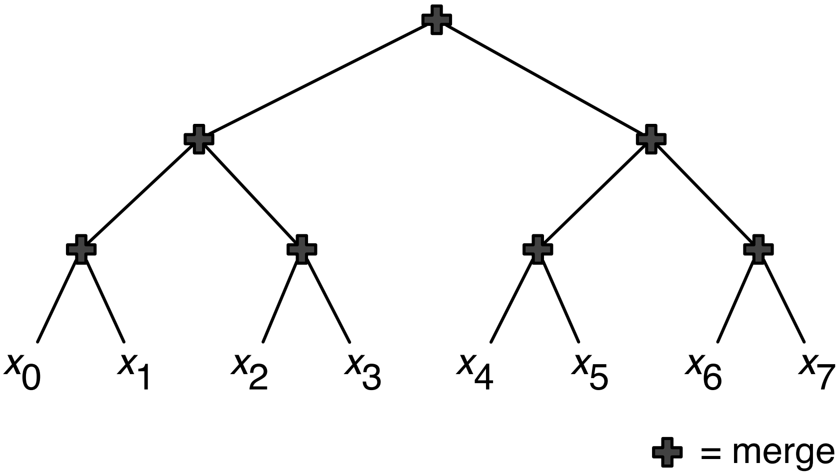
$$\textbf{fun}\ \ \mathtt{reduceSort'}(S) = \\ \quad \mathtt{iter}\ \mathtt{merge}_{<}\ (\mathtt{emptyVal}\ (\mathtt{map}\ \mathtt{base}\ S)$$

- Works really as an Insertion Sort.
  - Inserts each element into a sorted prefix!
- No parallelism except in `merge`

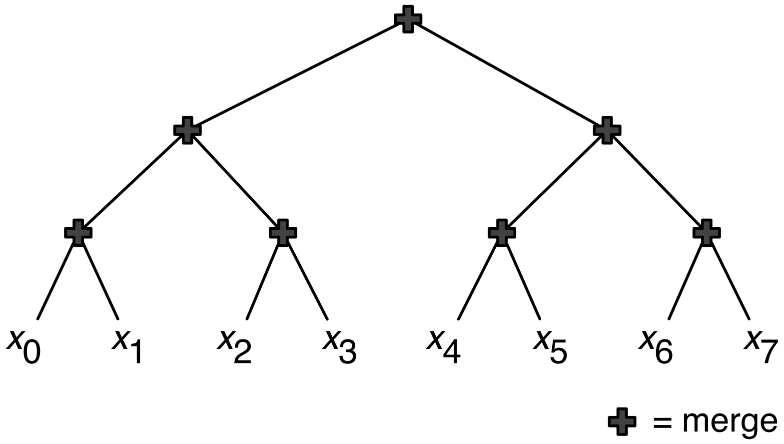
$$S(\mathtt{reduceSort}\ S) \ \leq\ \sum_{i=0}^{n-1} c \cdot \log(1+i) \ \in\ O(n \log n)$$

MERGESORT WITH REDUCE

- The reduction tree is very unbalanced!
- Suppose $n = 2^k$ and merge tree is as follows

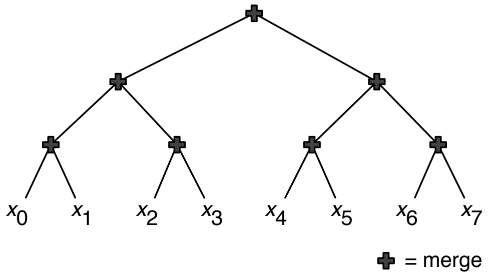


MERGESORT WITH REDUCE



- n nodes at constant cost at each leaf ($i = \log_2 n$)
- $n/2$ nodes one level up, each costing $c(1 + 1)$ ($i = \log_2 \frac{n}{2}$) (Why?)
- In general, for level i, we have 2^i nodes merging two sequences each length $\frac{n}{2^{i+1}}$

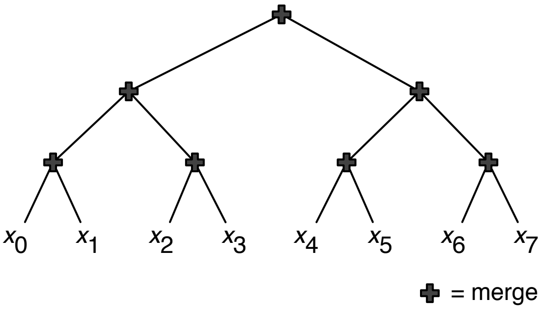
MERGESORT WITH REDUCE



- For level i, we have 2^i nodes merging two sequences each length $\frac{n}{2^{i+1}}$

$$
\begin{aligned}
W(\texttt{reduceSort } x) &\leq \sum_{i=0}^{\log n} 2^i \cdot c\left(\frac{n}{2^{i+1}} + \frac{n}{2^{i+1}}\right) \\
&= \sum_{i=0}^{\log n} 2^i \cdot c\left(\frac{n}{2^i}\right) \in O(n \log n)
\end{aligned}
$$

MERGESORT WITH REDUCE



- $W($`reduceSort` $S) \in O(n \log n) \Rightarrow$ `mergeSort`.
- `mergeSort` and `insertionSort` are special cases of `reduceSort` using different reduction orders.

REDUCE ORDER

- Result of `reduce` depends on the order when *f* is not associative
- When *f* is associative, different orders result in different costs.

15-210
PARALLEL AND SEQUENTIAL ALGORITHMS AND DATA STRUCTURES

LECTURE 7

COLLECT, SETS AND TABLES

SYNOPSIS

- The collect operation
- The map-collect-reduce paradigm
- Sets
- Tables

THE COLLECT OPERATION

- Group items that share a common key.

```
val Data = ⟨("jack sprat", "15-210"),
            ("jack sprat", "15-213"),
            ("mary contrary", "15-210"),
            ("mary contrary", "15-251"),
            ("mary contrary", "15-213"),
            ("peter piper", "15-150"),
            ("peter piper", "15-251"),
            ...⟩
                          ↓
val rosters = ⟨("15-150", ⟨"peter piper", ...⟩)
               ("15-210", ⟨"jack sprat", "mary contrary", ...⟩)
               ("15-213", ⟨"jack sprat", ...⟩)
               ("15-251", ⟨"mary contrary", "peter piper"⟩)
               ...⟩
```

THE COLLECT OPERATION

- Very common operation in Relational Databases
- Usually called the Group by operation.

$$
\begin{array}{ll}
\textbf{val}\ \mathtt{rosters} = & \langle(\text{“15-150”},\ \langle\text{“peter piper”}, \ldots\rangle) \\
 & (\text{“15-210”},\ \langle\text{“jack sprat”}, \text{“mary contrary”}, \ldots\rangle) \\
 & (\text{“15-213”},\ \langle\text{“jack sprat”}, \ldots\rangle) \\
 & (\text{“15-251”},\ \langle\text{“mary contrary”}, \text{“peter piper”}\rangle) \\
 & \ldots\rangle
\end{array}
$$

- Students are grouped by Course Numbers.

THE COLLECT OPERATION

$$\texttt{collect} : (\alpha \times \alpha \rightarrow \textit{order}) \rightarrow (\alpha \times \beta)\ \textit{seq}$$
$$\rightarrow (\alpha \times \beta\ \textit{seq})\ \textit{seq}$$

1. $\alpha \times \alpha \rightarrow$ *order* is a function for comparing keys of type α
2. $(\alpha \times \beta)$ *seq* is a sequence of key-value pairs
3. $(\alpha \times \beta$ *seq*$)$ *seq* is the resulting sequence:
    - each unique α value is paired with a sequence of all β values it appears with

THE COLLECT OPERATION

val $\mathtt{collectStrings} = \mathtt{collect\ \ String.compare}$

val $\mathtt{rosters} = \mathtt{collectStrings}(\langle\, (n, c) : (c, n) \in \mathtt{Data}\,\rangle)$

val $\mathtt{rosters} = \langle(\text{“15-150”},\ \langle\,\text{“peter piper”}, \ldots\rangle)$
$(\text{“15-210”},\ \langle\,\text{“jack sprat”}, \text{“mary contrary”}, \ldots$
$(\text{“15-213”},\ \langle\,\text{“jack sprat”}, \ldots\rangle)$
$(\text{“15-251”},\ \langle\,\text{“mary contrary”}, \text{“peter piper”}\,\rangle)$
$\ldots\rangle$

- $\langle\, (n, c) : (c, n) \in \mathtt{Data}\,\rangle$ arranges the data appropriately.

THE COLLECT OPERATION

- How would you implement `collect`?
  - Sort the items on their keys
  - Partition the resulting sequence
  - Pull out pairs between each key change

THE COLLECT OPERATION

- The dominant cost of `collect` is in sorting.
- Work is $O(W_c n \log n)$, Span is $O(S_c \log^2 n)$
  - W_c work bound for the comparison function
  - S_c span bound for the comparison function
- A $O(n)$ work can be implemented with hashing.
  - Need a separate hash function
  - Output not in sorted order

USING COLLECT IN MAP-REDUCE

- The map-reduce paradigm is used to process very large collection of documents.
  - A document is a collection of words/strings.
  - Not the `mapReduce` of 15-150!
- map-reduce paradigm $\equiv$ map-collect-reduce(s).

USING COLLECT IN MAP-REDUCE

- f_m maps each document to a sequence of key-value pairs.
    - $f_m : (document \rightarrow (key \times \alpha)\ seq)$

- All key-value pairs in a document are collected.

- f_r is applied to the keys to get a single value for a key.
    - $f_r : key \times \alpha\ seq \rightarrow \beta$

AN EXAMPLE

$$\begin{aligned}\mathtt{docs} \quad &= \langle\textit{“this is a document”},\\ &\quad \textit{“this is is another document”},\\ &\quad \textit{“a last document”}\rangle\end{aligned}$$

$$\downarrow$$

$$\begin{aligned}&\langle(\textit{“this”}, 1), (\textit{“is”}, 1), (\textit{“a”}, 1), (\textit{“document”}, 1),\\ &\ (\textit{“this”}, 1), (\textit{“is”}, 1), (\textit{“is”}, 1)(\textit{“a”}, 1), (\textit{“another”}, 1),\\ &\ (\textit{“document”}, 1), (\textit{“a”}, 1), (\textit{“last”}, 1), (\textit{“document”}, 1)\rangle\end{aligned}$$

$$\downarrow$$

$$\begin{aligned}&\langle(\textit{“a”}, 2), (\textit{“another”}, 1), (\textit{“document”}, 3), (\textit{“is”}, 3), (\textit{“last”}, 1),\\ &\ (\textit{“this”}, 2)\rangle\end{aligned}$$

MAPREDUCE IN SML

```
fun mapCollectReduce $f_m$ $f_r$ docs =
    let
        val pairs = flatten $\langle f_m(s) : s \in$ docs $\rangle$
        val groups = collect String.compare pairs
    in
        $\langle f_r(g) : g \in$ groups $\rangle$
    end
```

- $\texttt{flatten}\langle\langle a, b, c\rangle, \langle d, e\rangle\rangle \Rightarrow \langle a, b, c, d, e\rangle$

MapReduce in SML

```
fun mapCollectReduce $f_m$ $f_r$ docs =
    let
        val pairs = flatten $\langle f_m(s) : s \in docs \rangle$
        val groups = collect String.compare pairs
    in
        $\langle f_r(g) : g \in groups \rangle$
    end
```

```
fun $f_m(doc) = \langle (w, 1) : \texttt{tokens doc} \rangle$
fun $f_r(w, s) = (w, \texttt{reduce} + 0\ s)$
```

MAPREDUCE EXAMPLE IN SML

$$\textbf{fun}\ \ f_m(\mathit{doc}) = \langle\, (w, 1) : \mathtt{tokens}\ \ \mathit{doc}\,\rangle$$

$$\textbf{fun}\ \ f_r(w, s) = (w, \mathtt{reduce}\ \ +\ \ 0\ \ s)$$

$$\textbf{val}\ \ \mathtt{countWords} = \mathtt{mapCollectReduce}\ \ f_m\ \ f_r$$

$$\begin{aligned} \mathtt{countWords}\ \ \langle & \text{“}\mathit{this\ is\ a\ document}\text{”}, \\ & \text{“}\mathit{this\ is\ is\ another\ document}\text{”}, \\ & \text{“}\mathit{a\ last\ document}\text{”}\rangle \end{aligned}$$

$$\Rightarrow \langle(\text{“}\mathit{a}\text{”},\ 2), (\text{“}\mathit{another}\text{”},\ 1), (\text{“}\mathit{document}\text{”},\ 3), (\text{“}\mathit{is}\text{”},\ 3),$$
$$(\text{“}\mathit{last}\text{”},\ 1), (\text{“}\mathit{this}\text{”},\ 2)\rangle$$

SETS

- Sets play a very important role in math.
- Often needed in many algorithms.
- Many languages either support sets directly or have libraries for sets.
- In 15-210 we use a purely functional definition for sets:
  - When updates are done, a new set is returned.

Sets as an ADT

- $\mathbb{U}$ is a universe of elements.
- The SET ADT is a type $\mathbb{S}$ that represents the power set of $\mathbb{U}$.

$$\begin{array}{llll}
\texttt{empty} & : \; \mathbb{S} & = & \emptyset \\
\texttt{size}(S) & : \; \mathbb{S} \to \mathbb{Z}_{\geq 0} & = & |S| \\
\texttt{singleton}(e) & : \; \mathbb{U} \to \mathbb{S} & = & \{e\} \\
\texttt{filter}(f, S) & : \; ((\mathbb{U} \to \{\texttt{T}, \texttt{F}\}) & = & \{s \in S \mid f(s)\} \\
 & \quad \times \mathbb{S}) \to \mathbb{S} & &
\end{array}$$

SETS AS AN ADT

$$
\begin{array}{llcl}
\texttt{find}(S, e) & : \mathbb{S} \times \mathbb{U} & = & |\{s \in S \mid s = e\}| = 1 \\
 & \quad \rightarrow \{\mathrm{T}, \mathrm{F}\} & & \\
\texttt{insert}(S, e) & : \mathbb{S} \times \mathbb{U} \rightarrow \mathbb{S} & = & S \cup \{e\} \\
\texttt{delete}(S, e) & : \mathbb{S} \times \mathbb{U} \rightarrow \mathbb{S} & = & S \setminus \{e\}
\end{array}
$$

$$
\begin{array}{llcl}
\texttt{intersection}(S_1, S_2) & : \mathbb{S} \times \mathbb{S} \rightarrow \mathbb{S} & = & S_1 \cap S_2 \\
\texttt{union}(S_1, S_2) & : \mathbb{S} \times \mathbb{S} \rightarrow \mathbb{S} & = & S_1 \cup S_2 \\
\texttt{difference}(S_1, S_2) & : \mathbb{S} \times \mathbb{S} \rightarrow \mathbb{S} & = & S_1 \setminus S_2
\end{array}
$$

- What is the relationship between these two groups?

SETS AS AN ADT

- We do not really need `find`, `insert`, `delete`!

$$\begin{aligned}
\texttt{find}(S, e) &= \texttt{size}(\texttt{intersection}(S, \texttt{singleton}(e))) = 1 \\
\texttt{insert}(S, e) &= \texttt{union}(S, \texttt{singleton}(e)) \\
\texttt{delete}(S, e) &= \texttt{difference}(S, \texttt{singleton}(e))
\end{aligned}$$

- `intersection`, `union`, and `difference`
    - can operate on multiple elements, and
    - are suitable for parallelism

COST MODEL FOR SETS

- Underlying data structure can be
  - hash-tables
  - balanced trees
- We will assume a balanced-tree implementation.
- We will assume comparison of two set elements take
  - W_c work and S_c span.

COST MODEL FOR SETS

	Work	*Span*
`size`(S) `singleton`(e)	$O(1)$	$O(1)$
`filter`(f, S)	$O\left(\sum_{e \in S} W(f(e))\right)$	$O\left(\log \lvert S \rvert + \max_{e \in S} S(f(e))\right)$
`find`(S, e) `insert`(S, e) `delete`(S, e)	$O(W_c \cdot \log \lvert S \rvert)$	$O(S_c \cdot \log \lvert S \rvert)$

COST MODELS FOR SETS

`intersection`(S_1, S_2)
`union`(S_1, S_2)
`difference`(S_1, S_2)

$$\Rightarrow \quad Work = O\left(W_c \cdot m \cdot \log(1 + \frac{n}{m})\right) \qquad Span = O\left(S_c \cdot \log(n+m)\right)$$

$n = \max(|S_1|, |S_2|)$ $\qquad$ $m = \min(|S_1|, |S_2|)$

- Sets are equal size $(n = m)$
  - Work $= O(W_c \cdot m \cdot \log(1+1)) = O(W_c \cdot n)$
  - Span $= O(S_c \cdot \log n)$
- One of the sets is a singleton $(m = 1)$
  - Work $= O(W_c \cdot \log(1+n)) = O(W_c \cdot \log n)$
  - Span $= O(S_c \cdot \log(n+1)) = O(S_c \cdot \log n)$

TABLES

- Table is an ADT for sets of key-value pairs.

- $\{(k_1 \mapsto v_1), (k_2 \mapsto v_2), \ldots, (k_n \mapsto v_n)\}$
- $\{(k_1, v_1), (k_2, v_2), \ldots, (k_n, v_n)\}$

- Each key appears only once
- Many languages provide either built-in support or libraries.

TABLES

- $\mathbb{K}$ is the universe of keys.
- $\mathbb{V}$ is the universe of values.
- $\mathbb{T}$ is a type that represents the power set of $\mathbb{K} \times \mathbb{V}$
  - restricted so that each key appears at most once.
  - $\mathbb{S}$ is the power set of $\mathbb{K}$.
  - $\mathbb{Z}_{\geq 0}$ denotes the positive integers.

TABLE FUNCTIONS

$$
\begin{array}{llll}
\texttt{empty} & : & \mathbb{T} & = \quad \emptyset \\
\texttt{size}(T) & : & \mathbb{T} \to \mathbb{Z}_{\geq 0} & = \quad |T| \\
\texttt{singleton}(k, v) & : & \mathbb{K} \times \mathbb{V} \to \mathbb{T} & = \quad \{(k, v)\} \\
\texttt{filter}(f, T) & : & ((\mathbb{V} \to \{\texttt{T}, \texttt{F}\}) \times \mathbb{T}) & \\
 & & \quad \to \mathbb{T} & = \quad \{(k, v) \in T \mid f(v)\} \\
\texttt{map}(f, T) & : & ((\mathbb{K} \times \mathbb{V} \to \mathbb{V}) \times \mathbb{T}) & \\
 & & \quad \to \mathbb{T} & = \quad \{(k, f(k, v)) \mid ((k, v) \in T)\} \\
\texttt{insert}(f, T, (k, v)) & : & (\mathbb{V} \times \mathbb{V} \to \mathbb{V}) \times \mathbb{T} & \\
 & & \quad \times (\mathbb{K} \times \mathbb{V}) \to \mathbb{T} & = \\
 & & & \\
 & & \quad (T \setminus \{(k, v)\}) \cup & \\
 & & \qquad \{(k, f(v, v'))\} & \\
 & & \qquad \text{if } (k, v') \in T & \\
 & & \quad T \cup \{(k, v)\} & \\
 & & \qquad \text{otherwise} & \\
\texttt{delete}(T, k)) & : & \mathbb{T} \times \mathbb{K} \to \mathbb{T} & = \quad \{(k', v) \in T | k \neq k'\}
\end{array}
$$

TABLE FUNCTIONS

$$
\begin{array}{lcll}
\texttt{find}(T,k) & : & \mathbb{T}\times\mathbb{K}\rightarrow(\mathbb{V}\cup\bot) & = \begin{cases} v & (k,v)\in T \\ \bot & \mathit{otherwise} \end{cases} \\
\texttt{merge}(f,T_1,T_2) & : & (\mathbb{V}\times\mathbb{V}\rightarrow\mathbb{V})\times\mathbb{T}\times\mathbb{T}\rightarrow\mathbb{T} & = \\
 & & \multicolumn{2}{l}{\displaystyle\bigcup_{k\in\mathbb{K}} \begin{cases} \{(k,f(v_1,v_2))\} & (k,v_1)\in T_1\wedge(k,v_2)\in T_2 \\ \{(k,v_1)\} & (k,v_1)\in T_1\wedge(k,v_2)\notin T_2 \\ \{(k,v_2)\} & (k,v_2)\in T_2\wedge(k,v_1)\notin T_1 \end{cases}} \\
\texttt{extract}(T,S) & : & \mathbb{T}\times\mathbb{S}\rightarrow\mathbb{T} & = \{(k,v)\in T|k\in S\} \\
\texttt{erase}(T,S) & : & \mathbb{T}\times\mathbb{S}\rightarrow\mathbb{T} & = \{(k,v)\in T|k\notin S\}
\end{array}
$$

TABLE EXAMPLES

- Suppose we have the two tables:
    - $Summer = \{tree \mapsto green,\ sky \mapsto blue,\ cmuq \mapsto tan\}$
    - $Fall = \{grass \mapsto gray,\ tree \mapsto brown\}$

- `merge` $(fn\ (a, b) \Rightarrow b)\ Summer\ Fall$
    - $\{grass \mapsto gray,\ tree \mapsto brown,\ sky \mapsto blue,\ cmuq \mapsto tan\}$

TABLE EXAMPLES

- Suppose we have the two tables:
  - $Summer = \{tree \mapsto green,\ sky \mapsto blue,\ cmuq \mapsto tan\}$
  - $Fall = \{grass \mapsto gray,\ tree \mapsto brown\}$

- `extract`$(Summer, \{sky,\ grass\})$
  - $\{sky \mapsto\ blue\}$

TABLE EXAMPLES

- Suppose we have the two tables:
  - $\mathit{Summer} = \{\mathit{tree} \mapsto \mathit{green},\ \mathit{sky} \mapsto \mathit{blue},\ \mathit{cmuq} \mapsto \mathit{tan}\}$
  - $\mathit{Fall} = \{\mathit{grass} \mapsto \mathit{gray},\ \mathit{tree} \mapsto \mathit{brown}\}$

- $\texttt{erase}(\mathit{Summer}, \{\mathit{sky},\ \mathit{grass}\})$
  - $\{\mathit{tree} \mapsto \mathit{green},\ \mathit{cmuq} \mapsto \mathit{tan}\}$

TABLE EXAMPLES

- Other useful functions from the library
- `collect`:$(key \times \alpha)\ seq \rightarrow (\alpha\ seq)\ table$
- `fromSeq`: $(key \times \alpha)\ seq \rightarrow\ \alpha\ table$
  - `fromSeq`$(A) = \{k \mapsto s_0 : (k \mapsto S) \in collect(A)\}$

TABLE FUNCTIONS

- Major differences from sets:
  - `find` returns the value if key is in the table else returns ⊥ (`NONE`).
  - `insert/merge` need a function to combine if the key is already in the/both table(s).
- Just as with sets, there is symmetry between
  - `extract` and `find`
  - `merge` and `insert`
  - `erase` and `delete`

COST MODELS FOR TABLES

	Work	*Span*
`size`(T) `singleton`(k, v)	$O(1)$	$O(1)$
`filter`(f, T)	$O\left(\sum_{(k,v)\in T} W(f(v))\right)$	$O\left(\log\lvert T\rvert + \max_{(k,v)\in T} S(f(v))\right)$
`map`(f, T)	$O\left(\sum_{(k,v)\in T} W(f(k, v))\right)$	$O\left(\max_{(k,v)\in T} S(f(k, v))\right)$

COST MODELS FOR TABLES

	Work	*Span*
`find`(S, k) `insert`$(T, (k, v))$ `delete`(T, k)	$O(W_c \log \lvert T \rvert)$	$O(S_c \log \lvert T \rvert)$
`extract`(T_1, T_2) `merge`(T_1, T_2) `erase`(T_1, T_2)	$O\left(W_c m \log(1 + \frac{n}{m})\right)$	$O\left(S_c \log(n + m)\right)$

$n = \max(\lvert T_1 \rvert, \lvert T_2 \rvert) \quad m = \min(\lvert T_1 \rvert, \lvert T_2 \rvert)$

SUMMARY

- Collect
- Map-Collect-Reduce
- Sets
- Tables

15-210
PARALLEL AND SEQUENTIAL ALGORITHMS AND DATA STRUCTURES

LECTURE 8

SETS AND TABLES–II

SYNOPSIS

- How search engines work
- Single-threaded sequences

BUILDING A SEARCH ENGINE

How do search engines work?

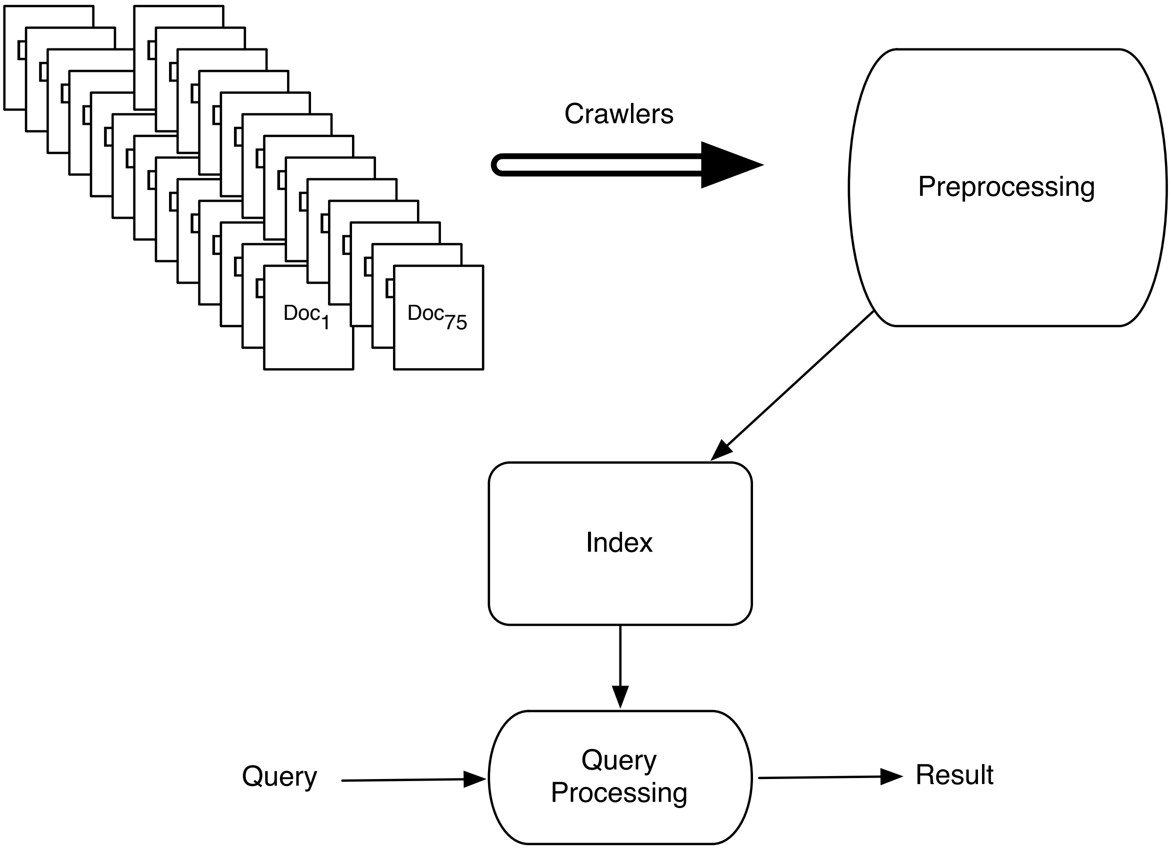
- What are the inputs?
  - (Billions and billions of) documents consisting of “words”.
- How do we interact with the search engine
  - (Boolean) Keyword queries
  - List of matching documents (URLS)

HOW DOES THE SEARCH REALLY WORK?

- User inputs a query (say a couple of words)
- SE starts searching for the words in each document one-by-one
- Returns documents when they match.

- Not really!
    - Not scalable (even for one user)
- Preprocessing

PREPROCESSING



PLAN

- What kinds of queries we want to have.
- What is the interface we want to have.
- How do we implement all these

QUERIES

- Bingle (:-) supports logical queries on words involving
    - `And`: “15210” `And` “course” `And` “slides”
    - `Or`: “15210” `Or` “15150”
    - `AndNot`: “15210” `AndNot` “Pittsburgh”
- “CMU” `And` “fun” `And` (“courses `Or` “clubs”)
- Result would be a list of webpages/documents that match.

THE INTERFACE

```
signature INDEX = sig
  type word = string
  type docId = string
  type 'a seq
  type index
  type docList

  val makeIndex : (docId * string) seq -> index

  val find : index -> word -> docList
  val And : docList * docList -> docList
  val AndNot : docList * docList -> docList
  val Or : docList * docList -> docList
  val size : docList -> int
  val toSeq : docList -> docId seq
```

DOCUMENTS

- Indexing a tweet database.

$T = \langle$ (“jack”, “chess club was fun”),
(“mary”, “I had a fun time in 210 class today”),
(“nick”, “food at the cafeteria sucks”),
(“sue”, “In 217 class today I had fun reading my email”),
(“peter”, “I had fun at nick’s party”),
(“john”, “tiddlywinks club was no fun, but more fun than 218”
$\rangle$

- “jack” is a document id
- “chess club was fun” is a document

USING THE INTERFACE

$T = \langle$ ("jack", "chess club was fun"),
("mary", "I had a fun time in 210 class today"),
("nick", "food at the cafeteria sucks"),
("sue", "In 217 class today I had fun reading my email"),
("peter", "I had fun at nick's party"),
("john", "tiddlywinks club was no fun, but more fun than 218"),
$\rangle$

val $f = (\mathtt{find}\ (\mathtt{makeIndex}(T))) : \mathtt{word} \to \mathtt{doclist}$

$\mathtt{toSeq}(\mathtt{And}(f\ \texttt{"fun"},\ \mathtt{Or}(f\ \texttt{"class"},\ f\ \texttt{"club"})))$

$\Rightarrow\ \langle \texttt{"jack"}, \texttt{"mary"}, \texttt{"sue"}, \texttt{"john"} \rangle$

USING THE INTERFACE

$T = \langle$ ("jack", "chess club was fun"),
("mary", "I had a fun time in 210 class today"),
("nick", "food at the cafeteria sucks"),
("sue", "In 217 class today I had fun reading my email"),
("peter", "I had fun at nick's party"),
("john", "tiddlywinks club was no fun, but more fun than 218"),
$\rangle$

`size(AndNot(f "fun", f "tiddlywinks"))`

$\Rightarrow$ 4

THE MAKEINDEX FUNCTION

```
fun $\mathtt{makeIndex}(\mathtt{docs}) =$
let
  fun $\mathtt{tagWords}(\mathtt{id}, \mathtt{str}) = \langle (w, \mathtt{id}) : w \in \mathtt{tokens}(\mathtt{str}) \rangle$
  val $\mathtt{Pairs} = \mathtt{flatten}\langle \mathtt{tagWords}(d) : d \in \mathtt{docs} \rangle$
  val $\mathtt{Words} = \mathtt{Table.collect}\ (\mathtt{Pairs})$
in
  $\{w \mapsto \mathtt{Set.fromSeq}(d) : (w \mapsto d) \in \mathtt{Words}\}$
end
```

- What does `tagWords` do?

$\mathtt{tagWords}$(*“jack”*, *“chess club was fun”*)

⇒ ⟨(*“chess”*,*“jack”*),(*“club”*, *“jack”*), (*“was”*, *“jack”*), (*“fun”*, *“jack”*)⟩

THE PAIRS FUNCTION

```
fun makeIndex(docs) =
let
  fun tagWords(id, str) = $\langle (w, id) : w \in tokens(str) \rangle$
  val Pairs = flatten$\langle tagWords(d) : d \in docs \rangle$
  val Words = Table.collect (Pairs)
in
  $\{w \mapsto$ Set.fromSeq$(d) : (w \mapsto d) \in Words\}$
end
```

- What does `Pairs` do?

$$\texttt{Pairs} = \langle(\text{“chess”},\text{“jack”}),(\text{“club”},\text{“jack”}),(\text{“was”},\text{“jack”}),\\ (\text{“fun”},\text{“jack”}),(\text{“I”},\text{“mary”}),(\text{“had”},\text{“mary”}),\\ (\text{“fun”},\text{“mary”}),\ldots\rangle$$

THE COLLECT FUNCTION

```
fun makeIndex(docs) =
let
  fun tagWords(id, str) = $\langle (w, id) : w \in tokens(str) \rangle$
  val Pairs = flatten$\langle tagWords(d) : d \in docs \rangle$
  val Words = Table.collect (Pairs)
in
  $\{w \mapsto Set.fromSeq(d) : (w \mapsto d) \in Words\}$
end
```

- What does `collect` do?

$$\begin{aligned} \texttt{Words} \quad = \quad & \{(\text{“a”} \mapsto \langle \text{“mary”} \rangle), \\ & (\text{“at”} \mapsto \langle \text{“mary”}, \text{“peter”} \rangle), \\ & \ldots \\ & (\text{“fun”} \mapsto \langle \text{“jack”}, \text{“mary”}, \text{“sue”}, \text{“peter”}, \text{“john”} \rangle), \\ & \ldots \end{aligned}$$

FINAL TOUCHES

```
fun $\mathtt{makeIndex}(\mathtt{docs}) =$
let
  fun $\mathtt{tagWords}(\mathtt{id}, \mathtt{str}) = \langle (w, \mathtt{id}) : w \in \mathtt{tokens}(\mathtt{str}) \rangle$
  val $\mathtt{Pairs} = \mathtt{flatten} \langle \mathtt{tagWords}(d) : d \in \mathtt{docs} \rangle$
  val $\mathtt{Words} = \mathtt{Table.collect\ (Pairs)}$
in
  $\{ w \mapsto \mathtt{Set.fromSeq}(d) : (w \mapsto d) \in \mathtt{Words} \}$
end
```

- What is happening here?
- Sequences are converted to tables.

makeIndex Costs

```
fun makeIndex(docs) =
let
  fun tagWords(id, str) = $\langle (w, id) : w \in tokens(str) \rangle$
  val Pairs = flatten $\langle tagWords(d) : d \in docs \rangle$
  val Words = Table.collect (Pairs)
in
  $\{w \mapsto Set.fromSeq(d) : (w \mapsto d) \in Words\}$
end
```

- Assuming tokens have a upper bound on length
  - $W_{makeIndex}(n) \in O(n \log n)$, $S_{makeIndex} \in O(\log^2 n)$
  - What does n represent?

REST OF THE INTERFACE

```
fun find T v = Table.find T v
fun And(s1, s2) = s1 ∩ s2
fun Or(s1, s2) = s1 ∪ s2
fun AndNot(s1, s2) = s1 \ s2
fun size(s) = |s|
fun toSeq(s) = Set.toSeq(s)
```

SINGLE-THREADED ARRAY SEQUENCES

- Updating an array sequence in an imperative language takes $O(1)$ work.
- In a functional setting, everything is persistent.
- An update to a sequence of n elements needs
  - $O(n)$ work for `arraySequence` implementation to copy and update.
  - $O(\log n)$ work for `treeSequence` implementation (because of substructure sharing)
- Interfaces do not provide functions for updating a single position.
  - to discourage sequential (and expensive) computation.

SINGLE-THREADED ARRAY SEQUENCES

- A *map* can be implemented as follows

```
fun map f S =
   iter (fn ((i, S'), v) ⇒ (i + 1, update (i, f(v)) S'))
        (0, S)
        S
```

- Iterates n times $(i = 0, \ldots n - 1)$
- and updates the value S_i with $f(S_i)$.
- `arraySequence`: Each update will do $O(n)$ work for a total $O(n^2)$ work
- `treeSequence`: Each update will do $O(\log n)$ work for a total $O(n \log n)$ work.

SINGLE-THREADED SEQUENCES

- A new ADT: Single Threaded Sequence: `stseq`
- Useful when a bunch of items need to be updated.
- Straigthforward interface
- Cost specification imply non-functional stuff under the hood!

STSEQ INTERFACE AND COSTS

	Work	Span
$\texttt{fromSeq(S)} : \alpha\ \texttt{seq} \rightarrow \alpha\ \texttt{stseq}$ Converts from a regular sequence to a stseq.	$O(\lvert S\rvert)$	$O(1)$
$\texttt{toSeq(ST)} : \alpha\ \texttt{stseq} \rightarrow \alpha\ \texttt{seq}$ Converts from a stseq to a regular sequence.	$O(\lvert S\rvert)$	$O(1)$
$\texttt{nth ST}\ i : \alpha\ \texttt{stseq} \rightarrow \texttt{int} \rightarrow \alpha$ Returns the i^{th} element of ST. Same as for seq.	$O(1)$	$O(1)$
$\texttt{update}\ (i,v)\ S : (\texttt{int} \times \alpha) \rightarrow \alpha\ \texttt{stseq} \rightarrow \alpha\ \texttt{stseq}$ Replaces the i^{th} element of S with v.	$O(1)$	$O(1)$
$\texttt{inject}\ I\ S : (\texttt{int} \times \alpha)\ \texttt{seq} \rightarrow \alpha\ \texttt{stseq} \rightarrow \alpha\ \texttt{stseq}$ For each $(i, v) \in I$ replaces the i^{th} element of S with v.	$O(\lvert I\rvert)$	$O(1)$

- Cost bounds for `nth` and `update` only valid for the “current” version of the sequence.

MAP WITH STSEQ

```
fun map f S = let
    val $S' = \texttt{StSeq.fromSeq}(S)$
    val $R$ = iter
            (fn $((i, S''), v) \Rightarrow (i + 1,\ \texttt{StSeq.update}\ (i, f(v))\ \ S''))$
            $(0, S')$
            $S'$
in
    $\texttt{StSeq.toSeq}(R)$
end
```

- Overall work and span is $O(n)$ (Why?)
- Multiple updates can be done in $O(n)$ time.

IMPLEMENTING STSEQ

- Keep two full copies of the sequence
    - Original and Current
    - We keep a change log: updates to the original to get Current.
- When Current is updated
    - We create a “new” Current with the update, and update change log.
    - Mark the previous version as old, remove its Current and but keep the old change log.
- Any item from the current version is accessible in constant work.
- Any item from the any previous version is accessible but needs more work.

IMPLEMENTING STSEQ

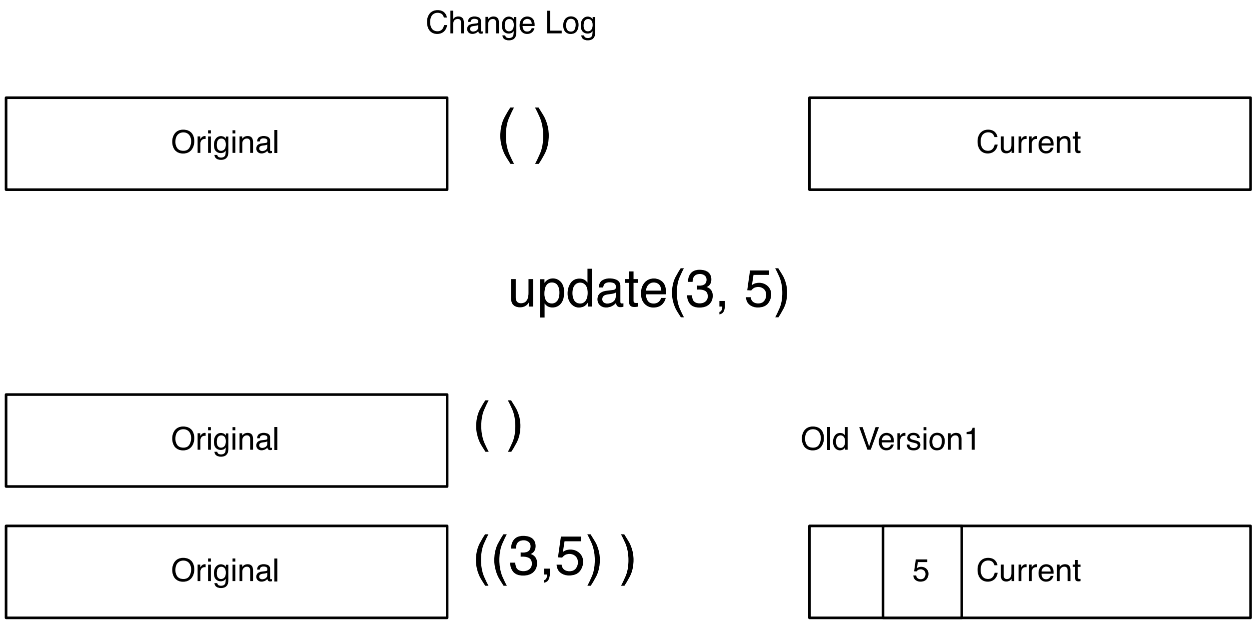
Change Log

Original

( )

Current

IMPLEMENTING STSEQ



Change Log

Original ( ) Current

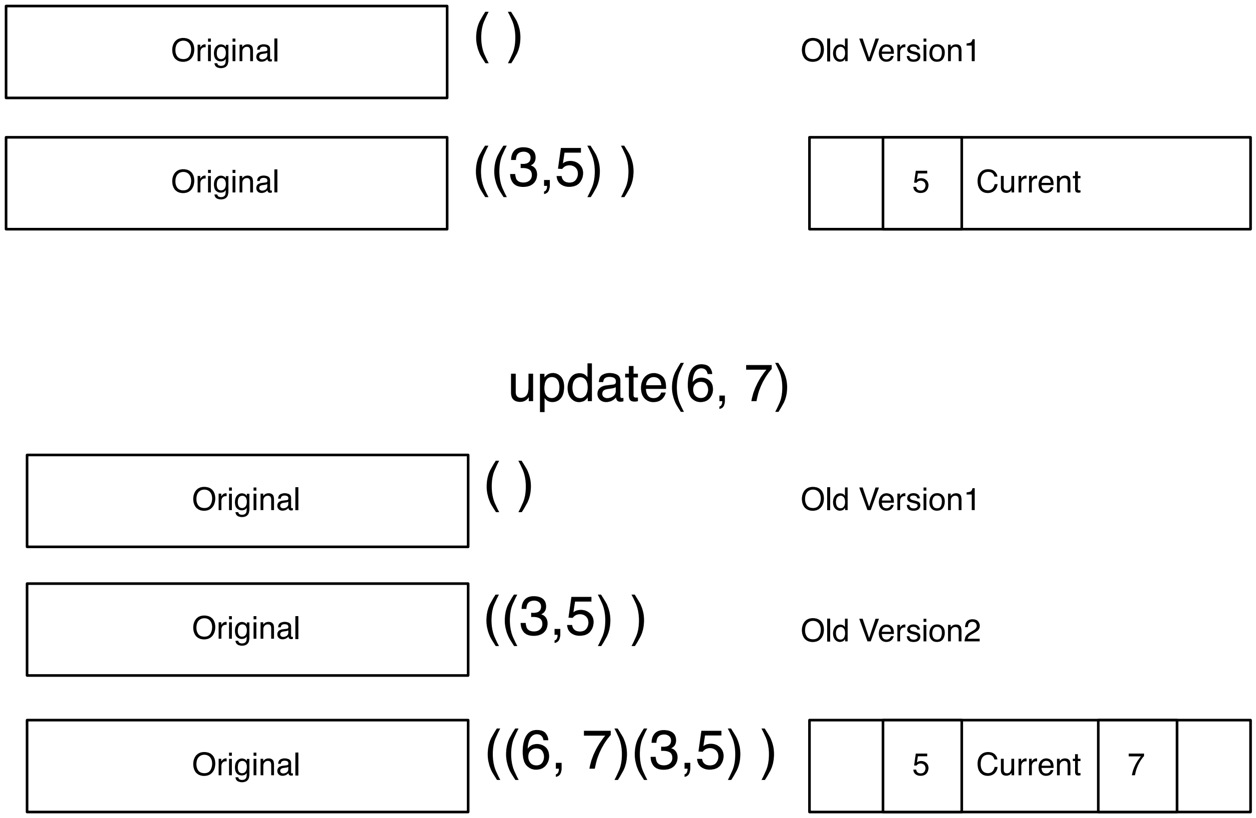
update(3, 5)

Original ( ) Old Version1

Original ((3,5)) 5 Current

- There really is only one copy of the Original.
- All point to that copy.

IMPLEMENTING STSEQ



Original ( ) Old Version1

Original ((3,5)) | | 5 | Current |

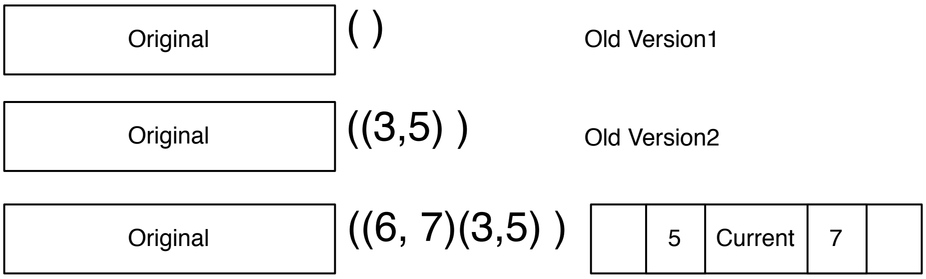
update(6, 7)

Original ( ) Old Version1

Original ((3,5)) Old Version2

Original ((6, 7)(3,5)) | | 5 | Current | 7 | |

IMPLEMENTING STSEQ

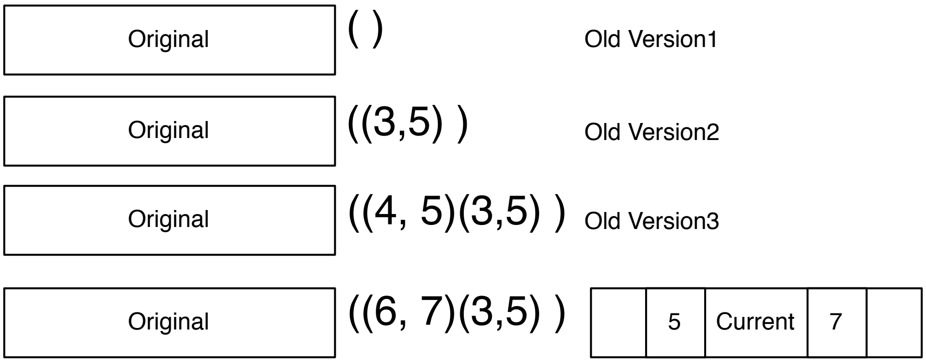


Original ( ) Old Version1

Original ((3,5)) Old Version2

Original ((6, 7)(3,5)) | 5 | Current | 7 |

$\text{update}_{\text{Oldversion2}}(4, 5)$



Original ( ) Old Version1

Original ((3,5)) Old Version2

Original ((4, 5)(3,5)) Old Version3

Original ((6, 7)(3,5)) | 5 | Current | 7 |

SUMMARY

- How search engines work
- Single-threaded sequences

15-210
PARALLEL AND SEQUENTIAL ALGORITHMS AND DATA STRUCTURES

LECTURE 9

GRAPHS

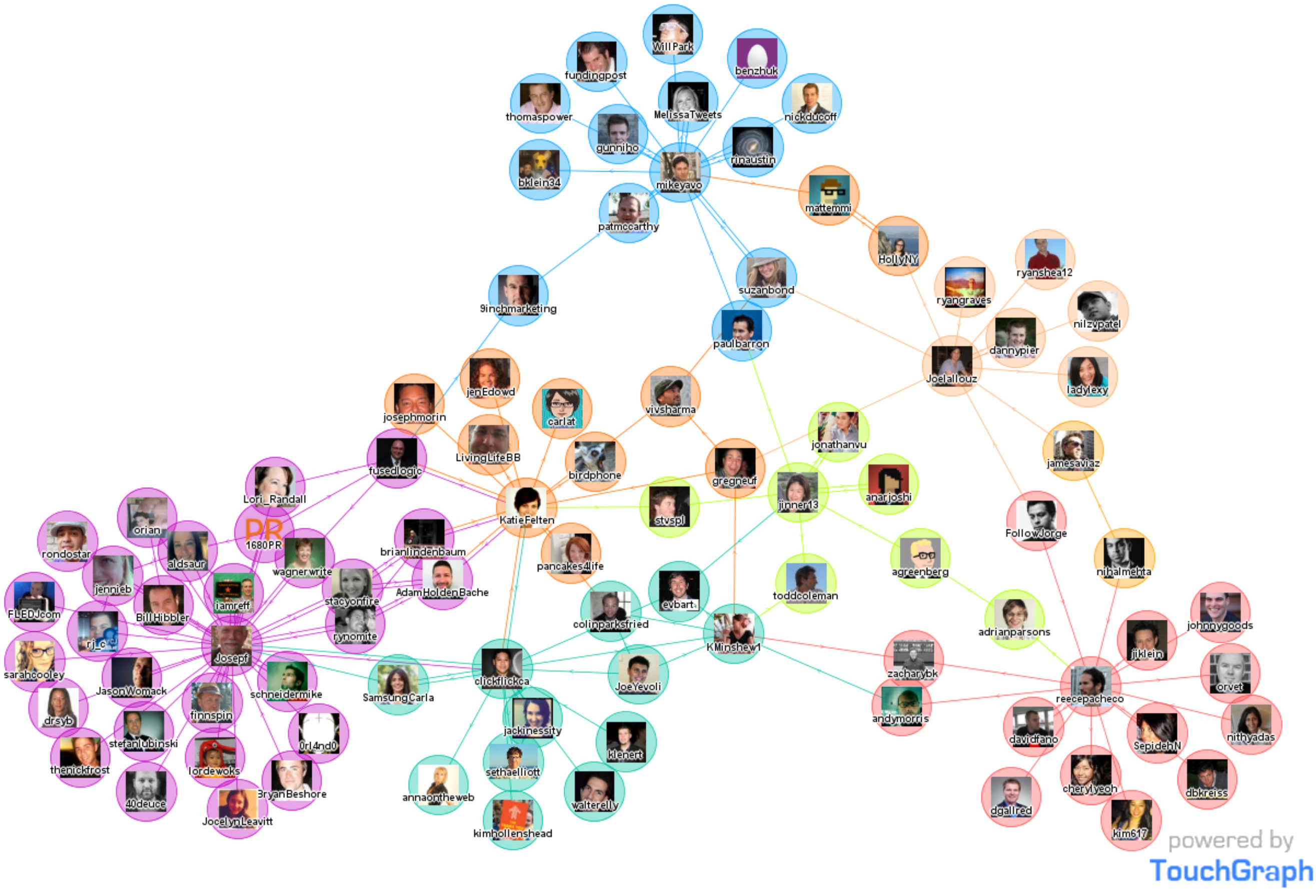
SYNOPSIS

- Graphs
- Graph terminology/definitions
- Graph representations/costs.
- Graph search

GRAPHS

- Most versatile ADT in the study of algorithms
- Captures relationships between pairs of items
- A graph consists of
  - a set of V vertices/nodes
  - a set edges $E \subseteq V \times V$
- Edges represent relationships between nodes.
  - directed edges (asymmetric relationships)
  - undirected edges (symmetric relationships)
- Nodes or edges can have additional weights or values associated.

SOCIAL NETWORKS



SOCIAL NETWORKS - QUESTIONS

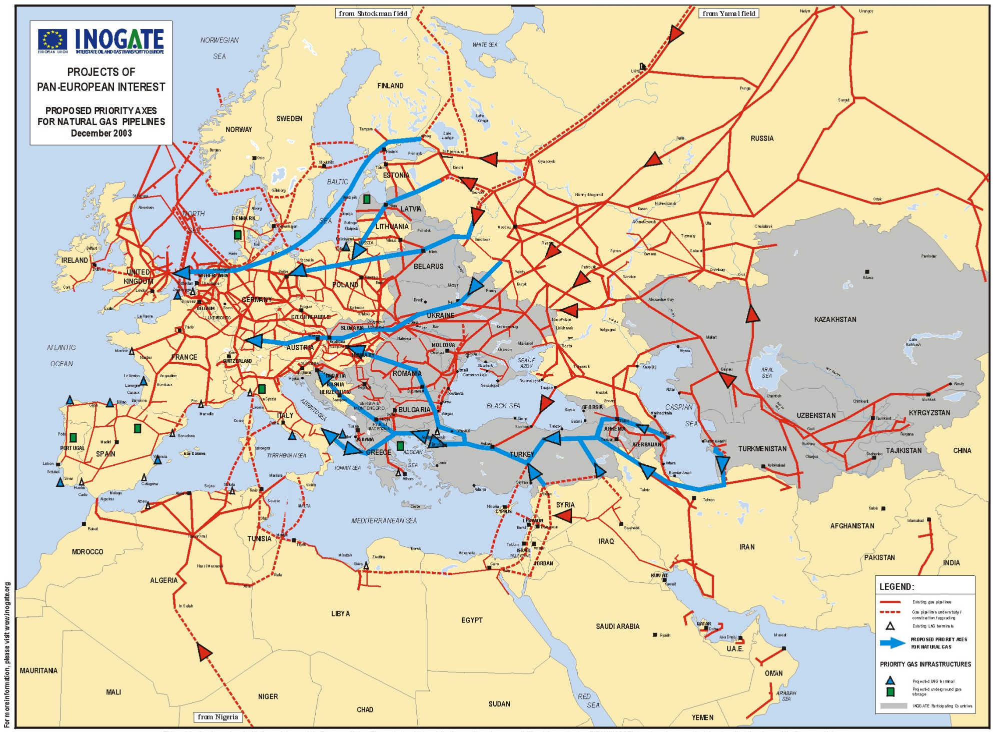
- Who is popular?
- What is the largest “clique”?
- Do I know somebody who knows X?
- What is the “diameter”?

TRANSPORTATION NETWORKS

TRANSPORTATION NETWORKS - QUESTIONS

- What is the shortest route from NYC to Los Angeles?
  - without Toll Roads?
  - without any state roads?
- What is the expected driving time from Boston to Atlanta?
  - considering traffic congestion?

FLOW NETWORKS



This publication is produced with the assistance of the European Union. The contents of this publication are the sole responsibility of the contractor, EIR / Hill / G&Fint, and can in no way be taken to reflect the views of the European Union.

FLOW NETWORKS - QUESTIONS

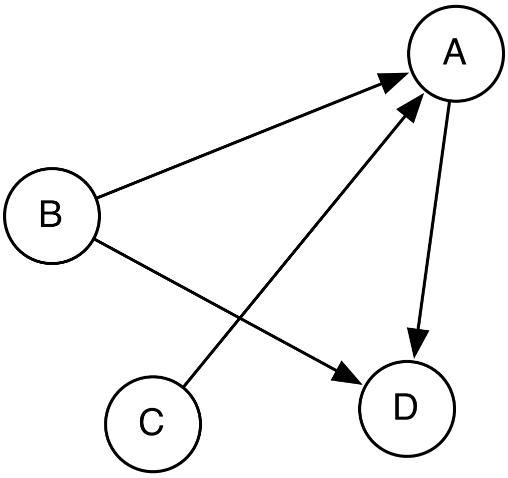
- Is it possible to send 1M cubic meters of gas to Paris daily?
- What is the maximum gas that can be pumped from Azerbaijan to Italy?

OTHER EXAMPLES OF GRAPHS

- Course prerequisite relation graphs (directed-acyclic)
- Web-page linkage graph
- Protein-protein interaction graph
- Neural networks
- Semantic networks

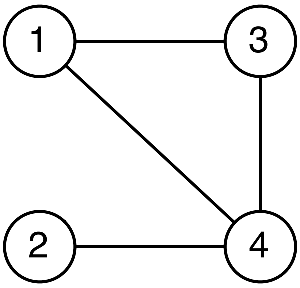
DIRECTED GRAPHS

- A directed graph (digraph) is $G = (V, E)$
  - V is a set of vertices (or nodes), and
  - $E \subseteq V \times V$ is a set of directed edges (or arcs).
- Each arc is an ordered pair $e = (u, v)$
  - Arcs represent asymmetric relationships
  - A graph can have self loops (u, u)



UNDIRECTED GRAPHS

- An undirected graph is $G = (V, E)$
  - V is a set of vertices (or nodes), and
  - $E \subseteq V \times V$ is a set of edges
- Each edge is an unordered pair $e = \{u, v\}$
  - Edges represent symmetric relationships
  - A undirected graphs do not have self-loops.



NEIGHBORS

- In an undirected graph, $G = (V, E)$, a vertex v is a neighbor of u if $\{u, v\} \in E$.
- In an undirected graph, $N_G(v) = \{u \mid \{u, v\} \in E\}$ is the neighborhood of v
- If U is a set of nodes,
  - $N_G(U) = \cup_{v \in U} N_G(v)$ is the neighborhood of U

NEIGHBORS

- In a directed graph, $G = (V, E)$,
    - u is an in-neighbor of v if $(u, v) \in E$
    - u is an out-neighbor of v if $(v, u) \in E$
- In a directed graph
    - $N_G^-(u)$ is the set of in-neighbors of u.
    - $N_G^+(u)$ is the set of out-neighbors of u.
    - When we use $N_G(v)$, we mean out-neighbors.
- If U is a set of nodes,
    - $N_G^+(U) = \cup_{u \in U} N_G^+(u)$ is the out-neighborhood of U.

NODE DEGREES

- Undirected graphs: degree $d_G(v)$ of a vertex v is $|N_G(v)|$
- Directed graphs:
  - in-degree of a vertex v is $d_G^-(v) = |N_G^-(v)|$
  - out-degree of a vertex v is $d_G^+(v) = |N_G^+(v)|$
- We will remove subscript G if it is clear from context.

PATHS

- A path is a sequence of adjacent vertices.
- For a graph $G = (V, E)$

$$Paths(G) = \{P \in V^+ \mid 1 \leq i < |P|, (P_i, P_{i+1}) \in E\}$$

  - V^+ is denotes of sequence of length 1 or more.
  - Repeats are allowed.
- The length of a path is the number of edges.
- A path may have an infinite length.
- A simple path has no repeated vertices.
  - Often “simple” will be dropped.

REACHABILITY

- A vertex v is reachable from a vertex u in G if there is a path starting at u and ending at v in G.
- $R_G(u)$ is the set of vertices reachable from u.
- An undirected graph is connected if all vertices are reachable from all other vertices.
- A directed graph is strongly connected if all vertices are reachable from all other vertices.

CYCLES

- A cycle is a path that starts and ends at the same vertex.
- In a directed graph a cycle can have length 1 (i.e. a self loop).
- In an undirected graph we require that a cycle must have length at least three.
    - Going from u to v and back to u does not count.
- A simple cycle is a cycle that has no repeated vertices other than the start vertex being the same as the end.

TREES, FORESTS AND DAGS

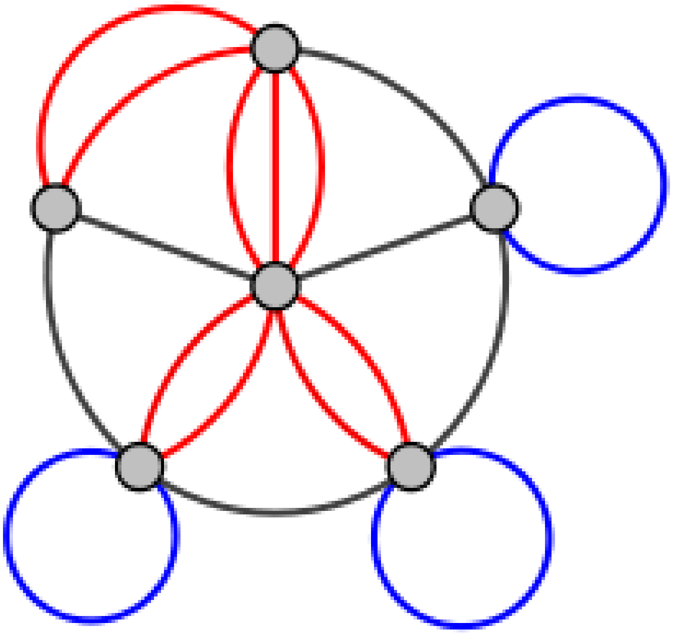
- An undirected graph with no cyles is a forest.
- If it is connected then it is a tree.
- A directed graph is a forest or tree if it becomes a forest or tree when all arcs are made undirected.
- In a rooted tree one node is the root.
- For a directed graph, all edges are either towards the root or away from the root.
- A directed graph with no cycles is a directed acyclic graph (DAG)

DISTANCE AND DIAMETER

- The distance $\delta_G(u, v)$ from a vertex u to a vertex v in a graph G is the *shortest* path (minimum number of edges) from u to v.
- The diameter of a graph is the *maximum shortest path length* over all pairs of vertices: $\text{diam}(G) = \max\{\delta_G(u, v) : u, v \in V\}$.

MULTI-GRAPHS

- **Multi-graphs** allow multiple edges between same pair of vertices.



SPARSE AND DENSE GRAPHS

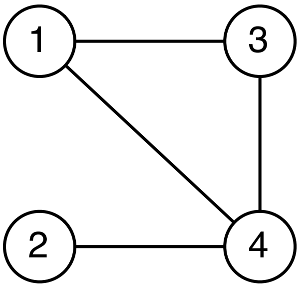
- Let $n = |V|$ and $m = |E|$.
- A directed graph can have at most n^2 edges.
- An undirected graph can have at most $\dfrac{n(n-1)}{2}$ edges.
- A graph is sparse if $m \ll n^2$. Otherwise it is called dense.
- In most applications, that graphs are sparse.
  - Nobody on Twitter has 10^9 followers
  - Though some have very large number– but still small when compared to n.

OPERATIONS ON GRAPHS

1. Map over the vertices $v \in V$.
2. Map over the edges $(u, v) \in E$.
3. Map over the neighbors of a vertex $v \in V$, or in a directed graph the in-neighbors or out-neighbors.
4. Return the degree of a vertex $v \in V$.
5. Determine if an edge (u, v) is in E.
6. Insert or delete vertices.
7. Insert or delete edges.

ADJACENCY MATRIX REPRESENTATION

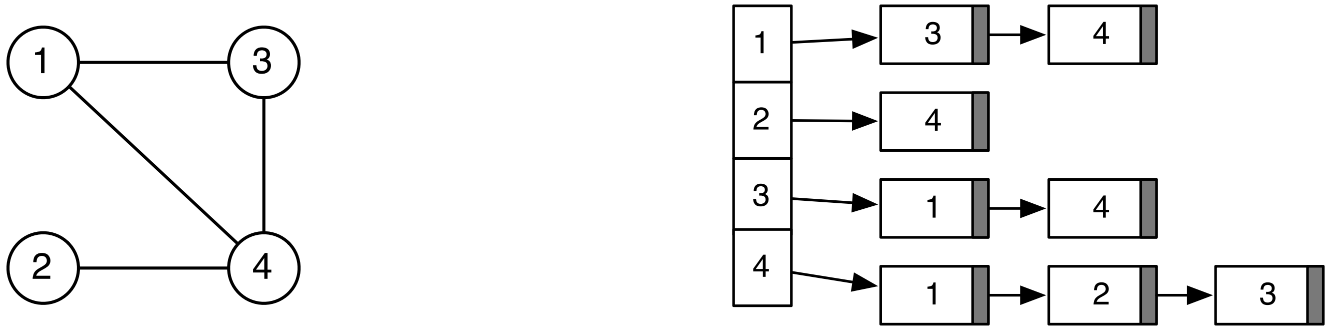
- Assume vertices are numbered $1, 2, \ldots, n$ (or $0, 1, \ldots, n-1$).
- Graph is represented by an $n \times n$ matrix of binary values in which location (i, j) is 1 if $(i, j) \in E$ and 0 otherwise.
  - For undirected graphs, matrix is symmetric and has 0's along the diagonal.



$$\begin{bmatrix} 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix}$$

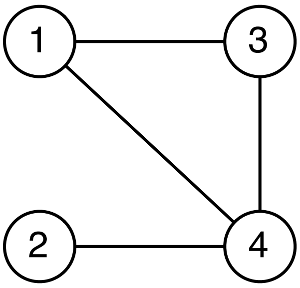
ADJACENCY LIST REPRESENTATION

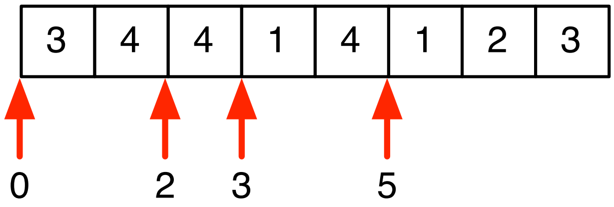
- Graph is represented by an array A of length n where each entry $A[i]$ contains a pointer to a linked list of all the out-neighbors of vertex i.
  - In an undirected graph edge $\{u, v\}$ will appear in the adjacency list for both u and v (not always necessary!)



OTHER REPRESENTATIONS

- Adjacency Array





- Edge List

$((1,3),(1,4),(2,4),(3,1),(3,4),(4,1),(4,2),(4,3))$

MORE ABSTRACT REPRESENTATIONS

- Edge Sets
  - Directed graphs: Set items are pairs (u, v) representing arcs.
  - Undirected graphs: Set items are sets $\{u, v\}$ representing edges.
- Edge Tables
  - Directed graphs: Table items are pairs $((u, v) \mapsto w_{u,v})$ representing arcs and associated values.
  - Undirected graphs: Set items are pairs $(\{u, v\} \mapsto w_{u,v})$ representing edges and associated values.

EDGE SETS AND TABLES

- Similar to edge lists but abstracts from underlying representation.
- Search for an edge needs $O(\log m)$ work.
- Searching for neighbors is not efficient: $O(m)$ work but $O(\log m)$ span. (Why?)

ADJACENCY TABLES

- Table items are (*key*, *value*) pairs.
- Keys are vertex/node labels.
- Values are either sets or tables
  - Sets: All neighbors node labels or out-neighbor node labels.
  - Tables: All pairs of neighbors node labels and associated edge values.
- Accessing neighbors needs $O(\log n)$ work and span.
- (Constant work) Map over neighbors needs $O(d_G(u))$ work and $O(\log d_G(u))$ span.
- Looking up an edge needs $O(\log n)$ work and span.

COST SUMMARY

	edge set		adj table	
	work	*span*	*work*	*span*
`isEdge`$(G, (u, v))$	$O(\log m)$	$O(\log m)$	$O(\log n)$	$O(\log n)$
`map over all edges`	$O(m)$	$O(\log m)$	$O(m)$	$O(\log n)$
`map over neighbors of` v	$O(m)$	$O(\log m)$	$O(\log n + d_G(v))$	$O(\log n)$
$d_G(v)$	$O(m)$	$O(\log m)$	$O(\log n)$	$O(\log n)$

GRAPH SEARCH

- Fundamental operation of graphs
  - Start at some (set of) node(s)
  - Systematically visit all reachable nodes (only once)
- Used for determining properties of graphs/nodes
  - Connected?
  - Bipartite?
  - Node v reachable from node u?
  - Shortest path from u to v?

GRAPH SEARCH

For all graph search methods vertices can be partitioned into three sets at any time during the search:

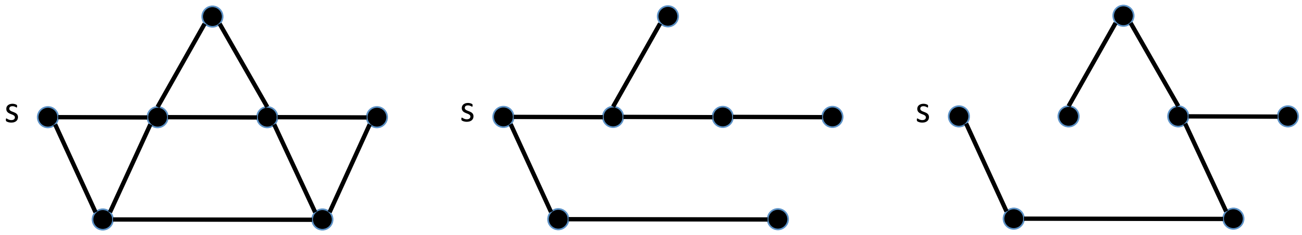
1. vertices already *visited* (X),
2. the unvisited neighbors of the visited vertices, called the *frontier* (F),
3. and the rest.

GRAPH SEARCH METHODS

- Breadth-first Search (BFS)
  - Parallelizable but for shallow graphs!
- Depth-first Search (DFS)
  - Inherently sequential!
- Priority-first Search (PFS)
- All reachable nodes from a source are visited, but in different orders.

GRAPH SEARCH TREES

- Each search starting from a source node creates a search tree.
- We refer to the source node as the root.



- Which search schemes do these correspond to?

SUMMARY

- Graphs
- Graph terminology/definitions
- Graph representations/costs.
- Graph search

15-210
PARALLEL AND SEQUENTIAL ALGORITHMS AND DATA STRUCTURES

LECTURE 10

BREADTH-FIRST SEARCH

SYNOPSIS

- Breadth-first search
- BFS Extensions
- BFS Costs
- BFS with Single-threaded Sequences

GRAPH SEARCH

- Fundamental operation of graphs
  - Start at some (set of) vertex(s)
  - Systematically visit all reachable vertices (only once)
- Used for determining properties of graphs/vertices
  - Connected?
  - Bipartite?
  - Vertex v reachable from vertex u?
  - Shortest path from u to v?

GRAPH SEARCH METHODS

- Breadth-first Search (BFS)
  - Parallelizable but for shallow graphs!
- Depth-first Search (DFS)
  - Inherently sequential!
- Priority-first Search (PFS)
- All reachable vertices from a source are visited, but in different orders.

BREADTH-FIRST SEARCH

- Applicable to a variety of problems
  - Connectedness
  - Reachability
  - Shortest path
  - Diameter
  - Bipartiteness
- Applicable to both directed and undirected graphs
  - For digraphs, we only consider outgoing arcs.

GRAPH SEARCH

- For all graph search methods vertices can be partitioned into three sets at any time during the search:
  1. vertices already *visited* ($X \subseteq V$),
  2. the unvisited neighbors of the visited vertices, called the *frontier* (F),
  3. the rest; unseen vertices.
- The search essential goes as follows:

```
while vertices remain
     -visit some unvisited neighbors
      of the visited set
```

- Web navigation analogy.

BREADTH-FIRST SEARCH

- Starting from a source vertex s
    - Visit all vertices that are (out-)neighbors of s (at distance 1)
    - Visit all vertices at distance 2 from s
    - Visit all vertices at distance 3 from s, etc.
- A vertex at distance $i + 1$ must have a (in-)neighbor at distance i.

BREADTH-FIRST SEARCH

- BFS needs to keep track of vertices already visited
- X_i: all vertices visited at start of level i
  - Vertices in X_i have distance less than i.
- F_i: all unvisited neighbors of vertices in X_i
  - Vertices in F_i have distance exactly i.
- "Visit" ⇒ Do something with the vertices (e.g., print it)
- $X_{i+1} = X_i \cup F_i$
- $F_{i+1} = N_G(F_i) \setminus X_{i+1} \quad (N_G(F_i) = \bigcup_{v \in F_i} N(v))$

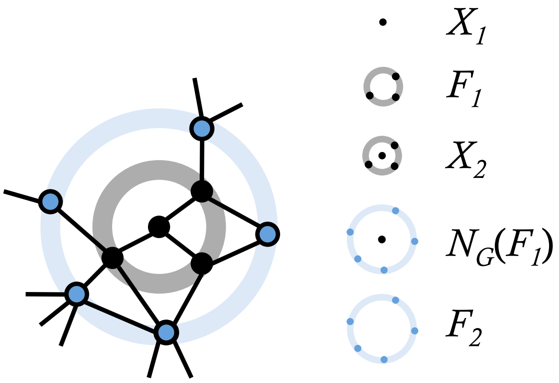
BREADTH-FIRST SEARCH

```
fun $BFS(G = (V, E), s) =$
let
  fun $BFS'(X, F, i) =$
    if $|F| = 0$ then $(X, i)$
    else let
        val $X' = X \cup F$            % Visit the Frontier
        val $N = N_G(F)$               % Determine the neighbors
                                       %  of the frontier
        val $F' = N \setminus X'$      % Remove vertices that have
                                       % been visited
    in $BFS'(X', F', i+1)$             % Next level
    end
in $BFS'(\{\}, \{s\}, 0)$
end
```

SOME DETAILS

- Adjacency table representation
  - Entries of the sort (*Vertex*, {*Neighbors*}).
- Remember $N_G(F) = \bigcup_{v \in F} N(v)$

```
fun $N_G(F)$ = Table.reduce Set.Union {}
                   Table.extract($G, F$)
```



PROVING BFS CORRECT

- State and prove an invariant.
- All reachable vertices are returned.
- Algorithm terminates.

PROVING BFS CORRECT

LEMMA

In algorithm BFS when calling BFS$'(X, F, i)$, we have

- $X = \{v \in V_G \mid \delta_G(s, v) < i\}$, and
- $F = \{v \in V_G \mid \delta_G(s, v) = i\}$

- By induction on levels i
- For base case ($i = 0$) $X_0 = \{\}$, $F_0 = \{s\}$
  - Only s has distance 0 from s
  - No vertex has distance < 0 from s.
- So base case is true!

PROVING BFS CORRECT

- Assume claims are true for i, show for $i + 1$.
- X_{i+1} is the union of
  - X_i: all vertices at distance $< i$
  - F_i: all vertices at distance $= i$
- Hence X_{i+1} must have all vertices at distance $< i + 1$
- $F_{i+1} = N_G(F_i) \setminus X_{i+1}$
  - Vertices in F_i have distance exactly i
  - Vertices in $N_G(F_i)$ have distance no more than $i + 1$
  - Vertices in $N_G(F_i)$ are reachable from a vertex at distance i
  - When we remove X_{i+1} from $N_G(F_i)$ only unvisited vertices at distance exactly $i + 1$ remain.

ADDITIONAL OBSERVATIONS

- If v is reachable from s and has distance d, there must be a vertex u at distance $d - 1$.
  - BSF will not terminate without finding v.
- For any vertex $\delta(s, v) < |V|$, so algorithm will terminate in at most $|V|$ rounds/levels.

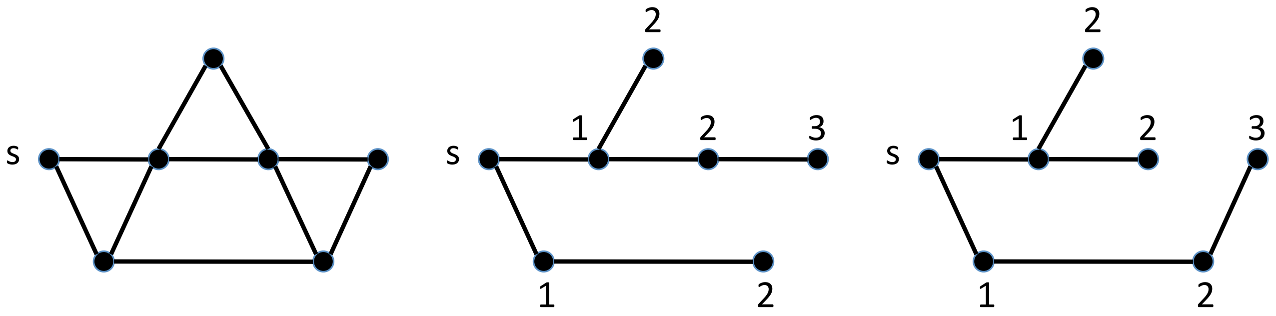
EXTENSIONS TO BFS

- Finding shortest distances
- What do we need to keep?

```
fun $BFS(G, s)$ = let
    fun $BFS'(X, F, i)$ =
        if $|F| = 0$ then $X$
        else let
            val $X' = X \cup \{v \mapsto i : v \in F\}$
            val $F' = N_G(F) \setminus domain(X')$
        in $BFS'(X', F', i + 1)$ end
in $BFS'(\{\}, \{s\}, 0)$ end
```

EXTENSIONS TO BFS

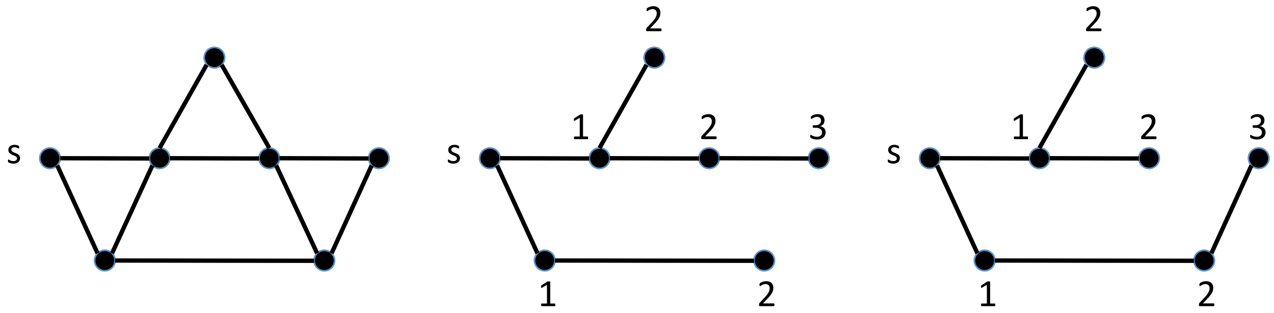
- Finding BFS trees.



- There could be multiple BFS trees.

FINDING BFS TREES

- What do we need to keep for each vertex?
- Record a parent
  - If v is in a frontier, then there should be one or more visited vertices u such that $(u, v) \in E$.
  - Any of those could be the parent of v.



IDENTIFYING PARENTS

- Post-process the BFS distance table
- Identify one (in-)neighbor vertex in $N^{-}(v)$ whose distance is one less.
- Another way is to keep a table of vertices mapping to parents.
  - For each $v \in F$, generate a table $\{u \mapsto v : u \in N(v)\}$
  - Maps each neighbor of v back to v.
- Merge these tables to X
  - Choose one if you have multiple parents.

COST ANALYSIS FOR BFS

- Most graph algorithms do NOT use divide and conquer.
  - So no natural way to develop recurrences and solve them.
- Instead, we just count steps

COST ANALYSIS FOR BFS

- BFS works in a sequence rounds (one per level)
- We can add up work and span in each round.
    - But work at a level depends on number of outgoing edges from the frontier!
- Take a more global view
    - Each vertex appears exactly once in some frontier.
    - All their (out-)edges are processed once.
- $W_{BFS}(n, m) = W_v n + W_e m$
    - $n = |V|$ and $m = |E|$

COSTS ANALYSIS FOR BFS

- Span is a bit more tricky!
- $S_{BFS}(n, m, d) = S_l d$ where d is the maximum distance ($d = \max_{v \in V} \delta(s, v)$)
- Assuming $W_v = O(\log n)$ and $W_e = O(\log n)$ and span/level $S_l = O(\log^2 n)$

$$\begin{aligned} W_{BFS}(n, m) &= O(n \log n + m \log n) \\ &= O(m \log n) \; (Why?) \\ S_{BFS}(n, m, d) &= O(d \log^2 n) \end{aligned}$$

COSTS PER VERTEX AND EDGE

- Nontrivial operations are
  1. $X' = X \cup F$
  2. $N = N_G(F)$
  3. $F' = N \setminus X'$.
- These all depend on size of F and number of outgoing edges from F.
- Let $||F|| = \sum_{v \in F}(1 + d_G^+(v))$
  - Vertices and outgoing edges in f.

COSTS PER VERTEX AND EDGE

	Work	Span
$X \cup F$	$O(\|F\| \log n)$	$O(\log n)$
$N \setminus X'$	$O(\|F\| \log n)$	$O(\log n)$

- These come from set cost specs.

$$Work = O(W_c \cdot |F| \log(1 + \frac{n}{|F|})) = O(|F| \log n)$$

$$Span = O(S_c \cdot \log(n + |F|)) = O(\log n)$$

COSTS PER VERTEX AND EDGE

	Work	Span
$N_G(F)$	$O(\|\|F\|\| \log n)$	$O(\log^2 n)$

- Graph is represented as a table mapping vertices to a set of their outneigbors.

$$\textbf{fun}\ \ N_G(F) = \texttt{Table.reduce Set.Union}\ \{\}\ (\texttt{Table.extract}(G, F))$$

- Extract vertices from table: Work is $O(|F| \log n)$

DIGRESSION – BACK TO REDUCE!

$$\begin{aligned}\textbf{fun}\quad N_G(F) = \texttt{Table.reduce}\ \texttt{Set.Union}\ \{\}\\ (\texttt{Table.extract}(G, F))\end{aligned}$$

$$\mathcal{R}(\texttt{reduce}\ f\ \mathbb{I}\ S) \;=\; \left\{\text{all function applications } f(a,b) \text{ in the reduction tree}\right\}.$$

$$\begin{aligned} W(\texttt{reduce}\ f\ \mathbb{I}\ S) &= O\left(n + \sum_{f(a,b)\in\mathcal{R}(f\ \mathbb{I}\ S)} W(f(a,b))\right)\\ S(\texttt{reduce}\ f\ \mathbb{I}\ S) &= O\left(\log n \max_{f(a,b)\in\mathcal{R}(f\ \mathbb{I}\ S)} S(f(a,b))\right)\end{aligned}$$

DIGRESSION – BACK TO REDUCE!

LEMMA

For any combine function $f : \alpha \times \alpha \to \alpha$ and a monotone size measure $s : \alpha \to \mathcal{R}_+$, if for any x, y,

1. $s(f(x, y)) \leq s(x) + s(y)$ and
2. $W(f(x, y)) \leq c_f\,(s(x) + s(y))$ for some universal constant c_f depending on the function f,

then

$$W(\mathtt{reduce}\ f\ \mathbb{I}\ S) = O\left(\log |S| \sum_{x \in S} (1 + s(x))\right)$$

BACK TO COSTS

- In our case α is the set type, f is `Set.union` , s the size of a set.
  1. Size of the union $\leq$ sum of the sizes.
  2. Work of a union $\leq$ is at most proportional to size of the sets!
- So `Set.union` satisfies the conditions of the lemma.
- $F_{ngh} = \mathtt{Table.extract}(G, F)$
  - F_{ngh} is a set of neighbor sets.

$$W(\mathtt{reduce\ union}\ \{\}\ F_{ngh}) = O\left(\log|F_{ngh}| \sum_{\mathit{ngh} \in F_{ngh}} (1 + |\mathit{ngh}|)\right)$$
$$= O(\log n \cdot ||F||)$$

BACK TO COSTS

$$S(\texttt{reduce union}\ \{\}\ F_{ngh}) = O(\log^2 n)$$

- Each union has span $O(\log n)$
- The reduction tree is bounded by $\log n$ depth.
- So at level i, $W = O(||F_i|| \cdot \log n)$ and each edge is processed once, $\Rightarrow$
  - work per edge is $O(\log n)$.
- Span depends on d $(S_{BFS}(n, m, d) = O(d \log^2 n))$
  - In worst $d \in O(n) \Rightarrow$ BFS is sequential.

BFS WITH ST SEQUENCES

- BFS Costs revisited

$$\begin{aligned} W_{BFS}(n, m) &= O(m \log n) \\ S_{BFS}(n, m, d) &= O(d \log^2 n) \end{aligned}$$

- Using single-threaded sequences reduces costs to

$$\begin{aligned} W_{BFS}(n, m) &= O(m) \\ S_{BFS}(n, m, d) &= O(d \log n) \end{aligned}$$

BFS WITH ST SEQUENCES

- Vertices are labeled with integers:
  - $V = \{0, 1, \ldots, n-1\}$
  - Integer labeled (IL) graphs.
- We use (array) sequences to represent graphs.
  - Constant work access to vertices.
  - Neighbors also stored as integer indices
- IL graphs are implemented with type `(int seq) seq`

BFS WITH ST SEQUENCES

- BFS returns a mapping from each vertex to its parent in the BFS tree.
- Visited vertices are maintained as `(int option) stseq`
    - `NONE`: Vertex has not been visited.
    - `SOME(v)`: Vertex visited from parent *v*.

BFS with ST Sequences

```
fun $\mathtt{BFS}(G : \mathtt{(int\ seq)\ seq},\ s : \mathtt{int}) =$
let
  fun $\mathtt{BFS}'(\mathtt{XF} : \mathtt{int\ option\ stseq},\ F : \mathtt{int\ seq}) =$
    if $|F| = 0$ then $\mathtt{stSeq.toSeq\ XF}$
    else let
            % compute neighbors of the frontier
      val $N = \mathtt{flatten}\langle\,\langle\,(u, \mathtt{SOME}(v)) : u \in G[v] \& XF[u] = \mathtt{NONE}\,\rangle : v \in F\,\rangle$
            % add new parents
      val $\mathtt{XF}' = \mathtt{stSeq.inject}(N,\ \mathtt{XF})$
            % remove duplicates
      val $F' = \langle\, u : (u, v) \in N \mid \mathtt{XF}'[u] = v\,\rangle$
    in $\mathtt{BFS}'(\mathtt{XF}',\ F')$ end
  val $X_0 = \mathtt{stSeq.toSTSeq}(\langle\,\mathtt{NONE} : v \in \langle\, 0, \ldots, |G| - 1\,\rangle\,\rangle)$
in
  $\mathtt{BFS}'(\mathtt{stSeq.update}(s, \mathtt{SOME}(s), X_0),\ \langle\, s\,\rangle)$
end
```

COSTS

	$XF : \mathtt{stseq}$	
line	work	span
`flatten`	$O(\|\|F_i\|\|)$	$O(\log n)$
`inject`	$O(\|\|F_i\|\|)$	$O(1)$
remove dup.	$O(\|\|F_i\|\|)$	$O(\log n)$
total across all d rounds	$O(m)$	$O(d \log n)$

SUMMARY

- Breadth-first search
- BFS Extensions
- BFS Costs
- BFS with Single-threaded Sequences

15-210
PARALLEL AND SEQUENTIAL ALGORITHMS AND DATA STRUCTURES

LECTURE 11

DEPTH-FIRST SEARCH

SYNOPSIS

- Depth-first search
- Cycle-detection in directed and undirected graphs
- Topological Sorting
- Generalizing DFS
- DFS with Single-threaded Sequences

GRAPH SEARCH

- Fundamental operation of graphs
    - Start at some (set of) node(s)
    - Systematically visit all reachable nodes (only once)
- Used for determining properties of graphs/nodes
    - Connected?
    - Bipartite?
    - Node v reachable from node u?
    - Shortest path from u to v?

GRAPH SEARCH METHODS

- Breadth-first Search (BFS)
  - Parallelizable but for shallow graphs!
- Depth-first Search (DFS)
  - Inherently sequential!
- Priority-first Search (PFS)
- All reachable nodes from a source are visited, but in different orders.

BREADTH-FIRST SEARCH

- Applicable to a variety of problems
    - Connectedness
    - Reachability
    - Shortest path
    - Diameter
    - Bipartedness
- Applicable to both directed and undirected graphs
    - For digraphs, we only consider outgoing arcs.

GRAPH SEARCH

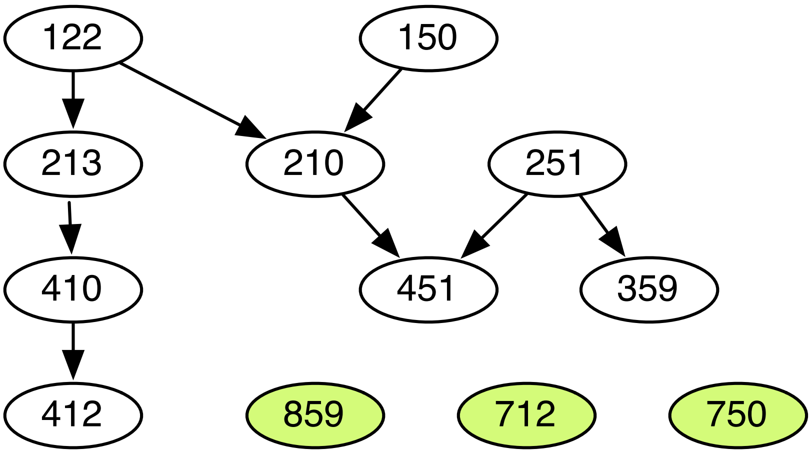
- For all graph search methods vertices can be partitioned into three sets at any time during the search:
  1. vertices already *visited* ($X \subseteq V$),
  2. the unvisited neighbors of the visited vertices, called the *frontier* (F),
  3. the rest; unseen vertices.
- The search essentially goes as follows:

```
while vertices remain
     -visit some invisited neighbors
      of the visited set
```

- Web navigation analogy.

TAKING CS COURSES

- Take the following courses – but one per semester



- What are some possible orders?

TOPOLOGICAL SORTING

- This problem is known as topological sorting.
  - Put vertices in a linear order that respects the graph precedence relationships.
- How can we know if a schedule is even possible?
  - There should be no cycles!
- Both these problems can be solved by depth-first search (DFS)
  - DFS looks at any edge at most twice.

DFS VS BFS

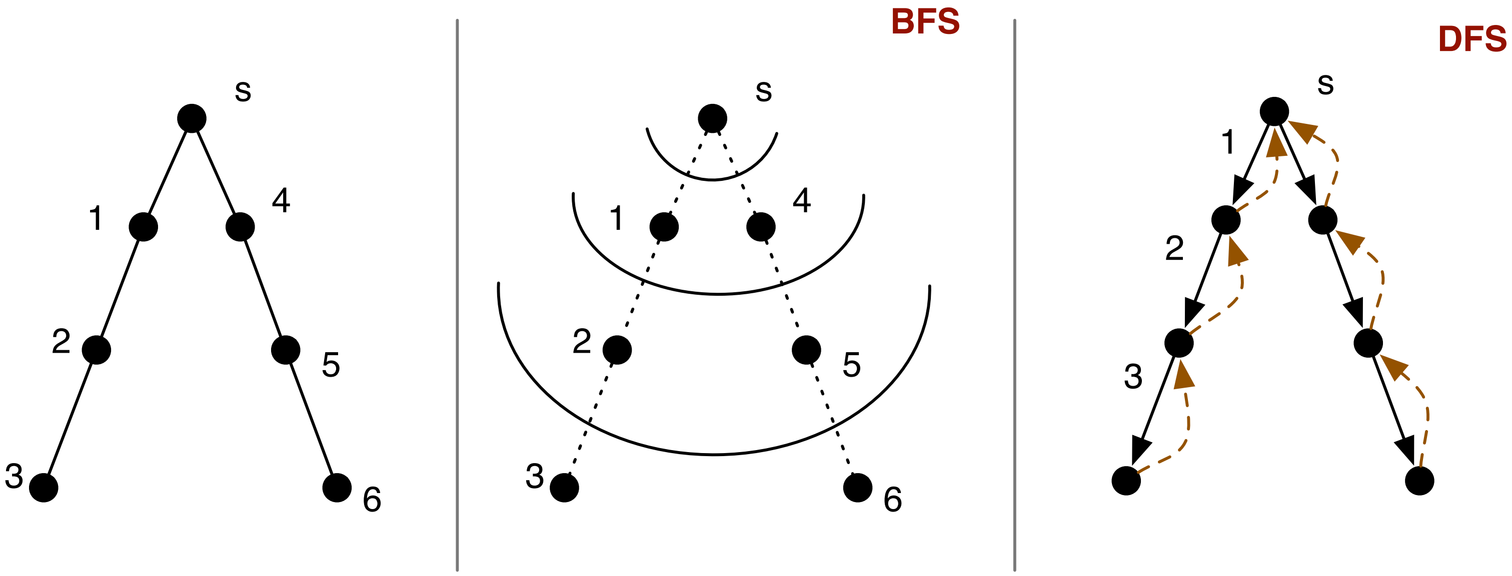
BFS

- Explores vertices one level at a time.
  - Increases breadth
  - No backtracking
- Can solve/generate
  - reachability
  - connectedness
  - spanning tree
- Not suitable for topological sort

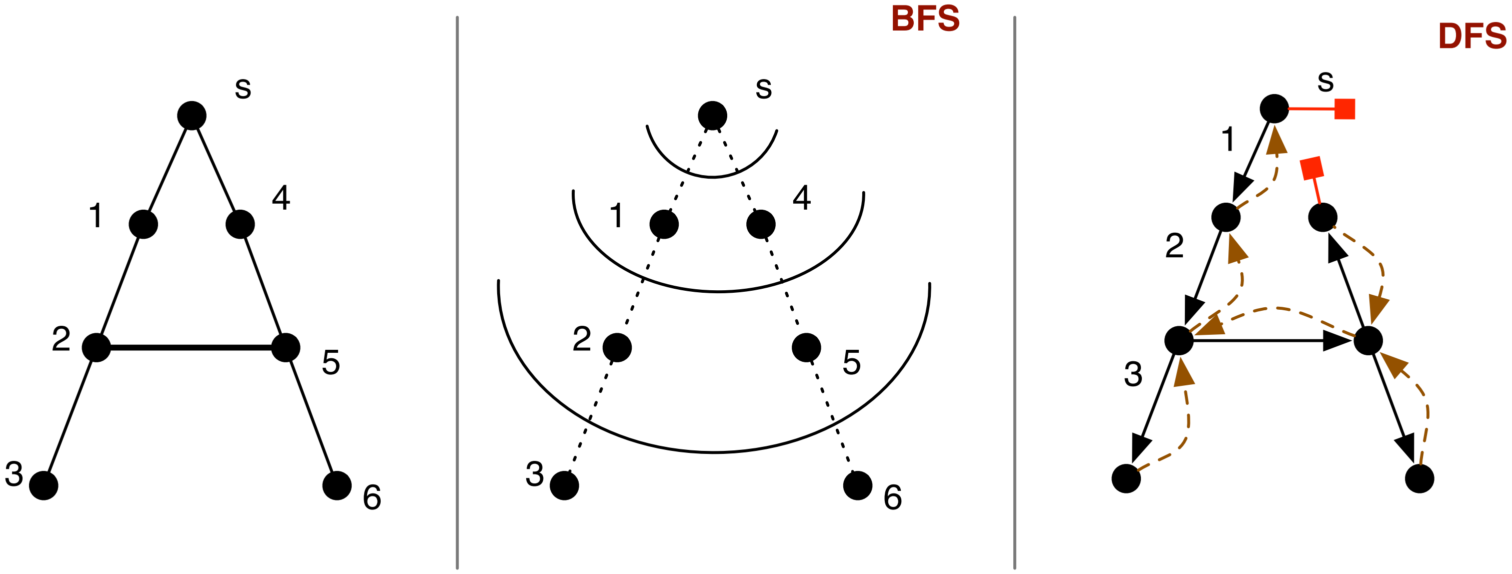
DFS

- Explores vertices one vertex at a time.
  - Increases depth
  - Backtracking when it can’t go deeper
- Can solve/generate
  - reachability
  - connectedness
  - spanning tree
- Not suitable for shortest unweighted path

DFS vs. BFS



DFS vs. BFS



THE DFS ALGORITHM

```
            : fun DFS($G$, $s$) = let
            :     fun DFS′($X$, $v$) =
            :         if ($v \in X$)
  TOUCH $v$ :         then $X$
            :         else let
  ENTER $v$ :             val $X' = X \cup \{v\}$
            :             val $X'' =$ iter DFS′ $X'$ ($N_G(v)$)
   EXIT $v$ :         in $X''$ end
            : in DFS′($\{\}$, $s$) end
```

- Each `iter` does a mapping of the sort $f : \alpha \times \beta \rightarrow \alpha$

```
$S = s_0$
foreach $a \in A$:
    $S = f(S, a)$
return $S$
```

SOME OBSERVATIONS

- `iter` goes sequentially
  - Sets are unordered, ordering depends on implementation!
- When a vertex v is entered (ENTER v) in code
  - it picks the “first” outgoing edge (v, w_1)
  - through `iter` calls $\texttt{DFS}'(X \cup \{v\}, w_1)$
- When $\texttt{DFS}'(X \cup \{v\}, w_1)$ returns
  - All vertices reachable from w_1 are explored
  - Vertex set returned is

$$X_1 = X \cup \{v\} \cup \{\text{All vertices reachable from } w_1\}$$

- `iter` picks next edge (v, w_2) and continues
- When `iter` is done

$$X'' = X \cup \{v\} \cup \{\text{All vertices reachable from } v\}$$

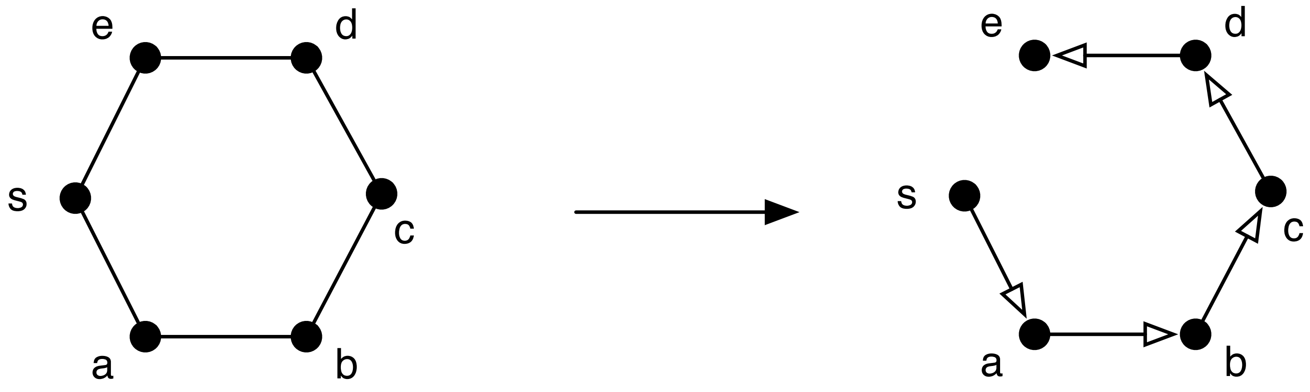
TOUCHING, ENTERING AND EXITING

```
           : fun DFS$(G, s)$ = let
           :   fun DFS$'(X, v)$ =
           :     if $(v \in X)$
TOUCH $v$  :     then $X$
           :     else let
ENTER $v$  :       val $X' = X \cup \{v\}$
           :       val $X'' =$ iter DFS$'$ $X'$ $(N_G(v))$
EXIT $v$   :     in $X''$ end
           : in DFS$'(\{\}, s)$ end
```

- We try to visit a vertex v
- We process v and its outgoing edges.
- We are done with v.

DFS WITH PARALLELISM

- Can we do all outgoing edges in parallel?
  - Yes - if parallel searches never meet up (then we really have a tree!)
  - No - otherwise.



COST OF DFS

LEMMA

For a graph $G = (V, E)$ with m out edges and n vertices:

- `DFS'` will be called at most m times
- There will be at most $\min(n, m)$ "enters".

- $v \in X$ can fail at most m times.
- we make call to `DFS'`, when we have an edge (total m times)
  - But we can enter a vertex a most once per `DFS'`
- So number of enters $\leq \min(n, m)$

COST OF DFS

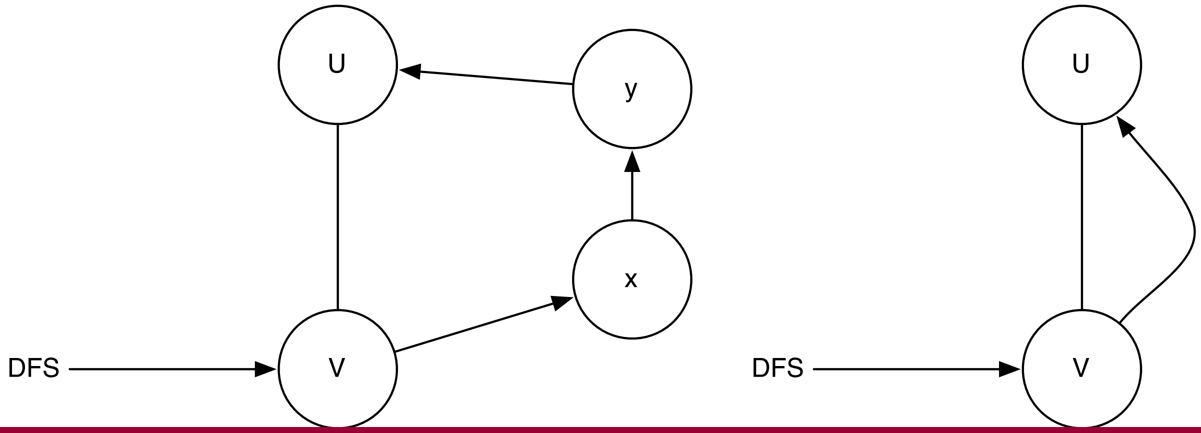
COROLLARY

The `DFS` algorithm on a graph with m out edges, and n vertices, and using the tree-based cost specification for sets, runs in $O(m \log n)$ work and span.

- Using ST sequences reduces work and span to $O(m)$

CYCLE DETECTION IN UNDIRECTED GRAPHS

- `DFS`' arrives at *v* a second time and this time from *u*. What can we conclude?
  - There must be two paths between *u* and *v*! (Why?)
- Not really! In undirected graphs cycles should have length at least 3.



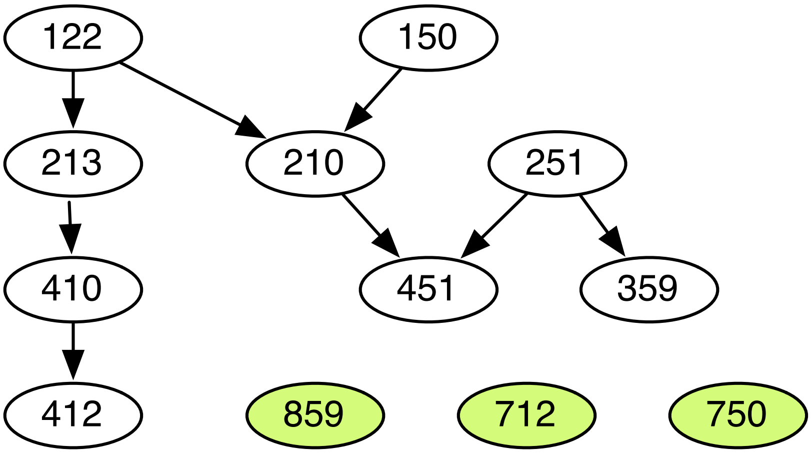
CYCLE DETECTION IN UNDIRECTED GRAPHS

```
          : fun $undirectedCycle(G, s) =$ let
          :   fun DFS' $\underline{p}$ $((X, \underline{C}), v) =$
          :     if $(v \in X)$
TOUCH v   :       then $(X, \underline{true})$
          :       else let
ENTER v   :         val $X' = X \cup \{v\}$
          :         val $(X'', C') = iter\ (\underline{DFS'\ v})\ (X', C)\ (\underline{N_G(v) \setminus \{p\}})$
EXIT v    :       in $(X'', C')$ end
          : in DFS' $\underline{s}$ $(\{\}, \underline{false}), s)$ end
```

- C keeps tracks of cycles.
- p is the parent – removed from neighbors and curried!

TOPOLOGICAL SORTING

- Order the vertices so that the ordering respects reachability.
  - If *u* is reachable from *v*, *v* must come earlier in the ordering.



PARTIAL ORDERS

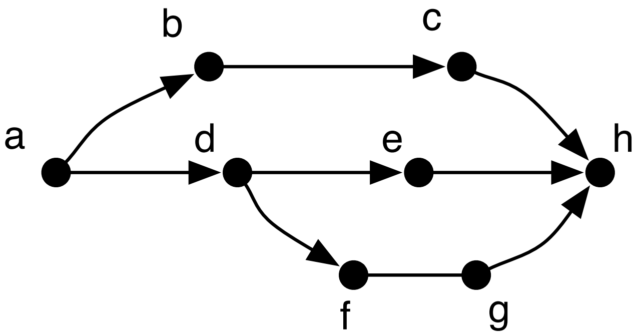
- A DAG defines a partial order on the vertices.
- For vertices $a, b \in V$, $a \leq_p b$ if and only if there is a directed path from a to b
- Partial order is a relation $\leq_p$ that obeys
  1. reflexivity — $a \leq_p a$,
  2. antisymmetry — if $a \leq_p b$ and $b \leq_p a$, then $b = a$, and
  3. transitivity — if $a \leq_p b$ and $b \leq_p c$ then $a \leq_p c$.

TOPOLOGICAL SORT

- If $\leq_t$ is the total ordering then

$$a \leq_p b \Rightarrow a \leq_t b$$

but not reverse is not necessarily true!



$$a \leq_t b \leq_t c \leq_t d \leq_t e \leq_t f \leq_t g \leq_t h$$

TOPOLOGICAL SORT WITH DFS

- Augment with a new source vertex s

$$G = (V, E) \rightarrow G' = (V \cup \{s\}, E \cup \{(s, v) : v \in V\})$$

- Why do we need to do this?

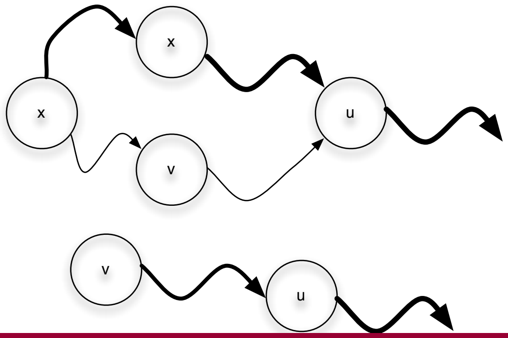
```
        : fun $topSort(G = (V, E))$ = let
        :     val $s$ = a new vertex
        :     val $G' = (V \cup \{s\}, E \cup \{(s, v) : v \in V\})$
        :     fun $DFS'((X, \underline{L}), v)$ =
        :         if $(v \in X)$
TOUCH v :         then $(X, \underline{L})$
        :         else let
ENTER v :             val $X' = X \cup \{v\}$
        :             val $(X'', L') = iter\ DFS'\ (X', L)\ (N_{G'}(v))$
 EXIT v :         in $(X'', \underline{v :: L'})$ end

        : in $DFS'((\{\}, \underline{[]}), s)$ end
```

TOPOLOGICAL SORT WITH DFS

THEOREM

On a DAG when exiting a vertex *v* in `DFS` all vertices reachable from *v* have already exited.

- Assume *u* is reachable from *v*.
  - *u* is entered before *v*. *u* must exit before *v* is entered (otherwise there is a cycle!)
  - *v* is entered before *u*. *u* will exit first.



CYCLE DETECTION IN DIRECTED GRAPHS

- Important preprocessing step in Topological Sort
  - Topological sort will return garbage when graph has cycles.

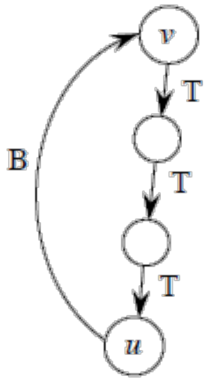
- We augment the graph with a node *s* with an edge to every other vertex the graph.
  - This can not add cycles. Nothing comes into *s*.

CYCLE DETECTION IN DIRECTED GRAPHS

```
          : fun $directedCycle(G = (V, E)) =$ let
          :     val $s =$ a new vertex
          :     val $G' = (V \cup \{s\}, E \cup \{(s, v) : v \in V\})$

          :     fun $\mathrm{DFS}'((X,\ \underline{Y},\ \underline{C}),\ v) =$
          :         if $(v \in X)$
 TOUCH v  :         then $(X,\ Y,\ \underline{v \overset{?}{\in} Y})$
          :         else let
 ENTER v  :             val $X' = X \cup \{v\}$
          :             val $Y' = \underline{Y \cup \{v\}}$
          :             val $(X'', Y'', C') = iter\ \mathrm{DFS}'\ (X', Y', C)\ (N_{G'}(v)$
 EXIT v   :         in $(X'',\ \underline{Y'' \setminus \{v\}},\ C')$ end

          :     val $(\_, \_, C) = \mathrm{DFS}'((\{\},\ \underline{\{\}},\ \underline{false}),\ s)$
          : in $C$ end
```

BACK EDGES IN A DFS SEARCH

- A back edge goes from a vertex v to an ancestor u in the DFS tree.



THEOREM

A directed graph $G = (V, E)$ has a cycle if and only if for $G' = (V \cup \{s\}, E \cup \{(s, v) : v \in V\})$ a DFS from s has a back edge.

GENERALIZING DFS

- All DFS code seem very much alike.
- They do work on
  - Touching,
  - Entering, and
  - Exiting
- We need to keep some state σ around
  - and update it appropriately!

$$\begin{aligned}
\Sigma_0 &: \alpha \\
\mathtt{touch} &: \alpha \times \mathtt{vertex} \times \mathtt{vertex} \rightarrow \alpha \\
\mathtt{enter} &: \alpha \times \mathtt{vertex} \times \mathtt{vertex} \rightarrow \alpha \\
\mathtt{exit} &: \alpha \times \mathtt{vertex} \times \mathtt{vertex} \rightarrow \alpha
\end{aligned}$$

GENERIC DFS ALGORITHM

```
fun $\texttt{DFS}(G, \Sigma_0, s) =$ let
    fun $\texttt{DFS}'\ p\ ((X, \Sigma), v) =$
        if $(v \in X)$ then $(X, \underline{\texttt{touch}}(\Sigma, v, p))$
        else let
            val $\Sigma' = \underline{\texttt{enter}}(\Sigma, v, p)$
            val $(X', \Sigma'') = \texttt{iter}\ (\texttt{DFS}'\ p)\ (X \cup \{v\}, \Sigma')\ N_G^+(v)$
            val $\Sigma''' = \underline{\texttt{exit}}(\Sigma, \Sigma'', v, p)$
        in $(X', \Sigma''')$ end
in $\texttt{DFS}'\ s\ ((\{\}, \underline{\Sigma_0}), s)$ end
```

UNDIRECTED CYCLE DETECTION

```
fun $\texttt{DFS}(G, \underline{\Sigma_0}, s)$ = let
   fun $\texttt{DFS}'\ p\ ((X, \Sigma), v) =$
      if $(v \in X)$ then $(X, \underline{\texttt{touch}}(\Sigma, v, p))$
      else let
         val $\Sigma' = \underline{\texttt{enter}}(\Sigma, v, p)$
         val $(X', \Sigma'') = \texttt{iter}\ (\texttt{DFS}'\ p)\ (X \cup \{v\}, \Sigma')\ N_G^+(v)$
         val $\Sigma''' = \underline{\texttt{exit}}(\Sigma, \Sigma'', v, p)$
      in $(X', \Sigma''')$ end
in $\texttt{DFS}'\ s\ ((\{\}, \underline{\Sigma_0}), s)$ end
```

```
$\Sigma_0 = ([s],\ \texttt{false}) : \texttt{vertex list} \times \texttt{bool}$
fun $\texttt{touch}((L\ \texttt{as}\ h :: T,\ \texttt{fl}),\ v,\ p) = (L,\ h \neq p)$
fun $\texttt{enter}((L,\ \texttt{fl}),\ v,\ p) = (v :: L,\ \texttt{fl})$
fun $\texttt{exit}((L\ \texttt{as}\ h :: T,\ \texttt{fl}),\ v,\ p) = (T,\ \texttt{fl})$
```

TOPOLOGICAL SORT

```
fun DFS$(G, \Sigma_0, s) =$ let
  fun DFS$'$ $p$ $((X, \Sigma), v) =$
    if $(v \in X)$ then $(X, \underline{touch}(\Sigma, v, p))$
    else let
      val $\Sigma' = \underline{enter}(\Sigma, v, p)$
      val $(X', \Sigma'') = iter$ (DFS$'$ $p$) $(X \cup \{v\}, \Sigma')$ $N_G^+(v)$
      val $\Sigma''' = \underline{exit}(\Sigma, \Sigma'', v, p)$
    in $(X', \Sigma''')$ end
in DFS$'$ $s$ $((\{\}, \underline{\Sigma_0}), s)$ end
```

$\Sigma_0 = [] : vertex\ list$
fun $touch(L,\ v,\ p) = L$
fun $enter(L,\ v,\ p) = L$
fun $exit(L,\ v,\ p) = v :: L$

DIRECTED CYCLE DETECTION

```
fun $\mathtt{DFS}(G, \Sigma_0, s) =$ let
  fun $\mathtt{DFS}'\ p\ ((X, \Sigma), v) =$
    if $(v \in X)$ then $(X, \underline{\mathtt{touch}}(\Sigma, v, p))$
    else let
      val $\Sigma' = \underline{\mathtt{enter}}(\Sigma, v, p)$
      val $(X', \Sigma'') = \mathtt{iter}\ (\mathtt{DFS}'\ p)\ (X \cup \{v\}, \Sigma')\ N_G^+(v)$
      val $\Sigma''' = \underline{\mathtt{exit}}(\Sigma, \Sigma'', v, p)$
    in $(X', \Sigma''')$ end
in $\mathtt{DFS}'\ s\ ((\{\}, \underline{\Sigma_0}), s)$ end
```

$\Sigma_0 = (\{\},\ \mathtt{false}) : \mathtt{Set} \times \mathtt{bool}$
fun $\mathtt{touch}((S,\ \mathtt{fl}),\ v,\ p) = (S,\ v \overset{?}{\in} S)$
fun $\mathtt{enter}((S,\ \mathtt{fl}),\ v,\ p) = (S \cup \{v\},\ \mathtt{fl})$
fun $\mathtt{exit}((S,\ \mathtt{fl}),\ v,\ p) = (S \setminus \{v\},\ \mathtt{fl})$

DFS WITH ST SEQUENCES

```
fun DFS($G$ : (int seq) seq,  $s$ : int) =
let
   fun DFS′ $p$ (($X$ : bool stseq, $\Sigma$),  $v$ : int) =
      if ($X[v]$) then ($X$, touch($\Sigma, v, p$))
      else let
         val $X' =$ update($v$, true, $X$)
         val $\Sigma' =$ enter($\Sigma, v, p$)
         val $(X'', \Sigma'') =$ iter (DFS′$v$) $(X', \Sigma')$ $(G[v])$
      in ($X''$, exit($\Sigma'', v, p$))
   val $X_{init} =$ stSeq.fromSeq($\langle$ false : $v \in \langle 0, \ldots, |G| - 1 \rangle \rangle$)
in
   stSeq.toSeq(DFS′(($X_{init}, \Sigma_0$),  $s$))
end
```

- $O(m)$ work and span.

15-210
PARALLEL AND SEQUENTIAL ALGORITHMS AND DATA STRUCTURES

LECTURE 13

SHORTEST WEIGHTED PATHS

SYNOPSIS

- Representing weighted graphs.
- Priority-first Search
- Shortest weighted paths
- Dijkstra’s Algorithm

WEIGHTED GRAPH REPRESENTATION

- $G = (V, E, w)$ where $w\colon E \to \texttt{eVal}$
- `eVal` is a set (type) of possible values
  - Typically real numbers, but could be anything!
- Table of $(\mathit{edge} \mapsto \mathit{weight})$.

$$W = \{(0,1) \mapsto 0.7,\ (1,2) \mapsto -2.0,\ (0,2) \mapsto 1.5\}$$

- We could use $\mathit{find}\ W\ e$ to find $w(e)$.

WEIGHTED GRAPH REPRESENTATION

- Table of $(\mathit{vertex} \mapsto \text{table of } (\mathit{vertex} \mapsto \mathit{weight}))$

$$G = \{0 \mapsto \{1 \mapsto 0.7,\ 2 \mapsto 1.5\},\ 1 \mapsto \{2 \mapsto -2.0\},\ 2 \mapsto \{\}\}$$

- With one lookup, we can get to the neighbors and weights.
- We will mostly use this representation.

PRIORITY-FIRST SEARCH

- Generalization of BFS and DFS – also called best-first search
- Visits vertices in some priority order
  - Static - decided ahead of time
  - Dynamic – decided on the fly– while things change during the search
-

PRIORITY-FIRST SEARCH

```
fun pfs(G, s) = let
  fun pfs′(X, F) =
  if (F = {}) then X
  else let
      val M = highest priority vertices in F
      val X′ = X ∪ M
      val F′ = (F ∪ N(M)) \ X′
  in pfs′(X′, F′) end
in pfs′({}, {s}) end
```

PRIORITY-FIRST SEARCH

- Several famous graph algorithms are instances of priority-first search.
  - Dijkstra's Algorithm for single-source shortest paths (SSSP).
  - Prim's Algorithm for minimum spanning trees (MST).
- PFS is a greedy algorithm.
  - It greedily adds vertices from the frontier based on a cost function.
  - It never backs up!

SHORTEST WEIGHTED PATHS

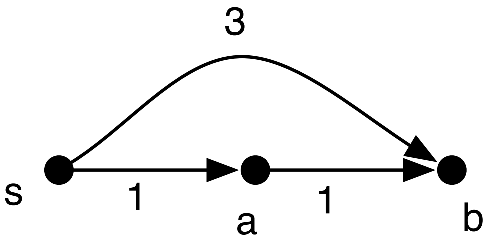
- $G = (V, E, w)$ with $w : E \to \mathcal{R}$
  - $w(u, v) = \infty$ if $(u, v) \notin E$
- The weight of a path is the sum of the weights of the edges on it.

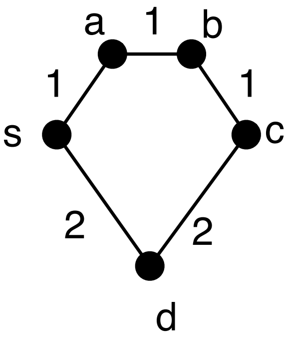
THE SSSP PROBLEM

- Given a graph G and a source vertex s, find the shortest weighted path to every other vertex.
  - $\delta_G(u, v)$ is the weight of the shortest path from u to v

DIJKSTRA'S ALGORITHM

- Dijkstra's Algorithm solves SSSP when the weights are non-negative ($w : E \to \mathcal{R}_+ \cup \{0\}$).
    - Greedy
    - Finds optimal solutions to a nontrivial task.
- Why do we need a new algorithm? Why not use BFS?





OBSERVATIONS

- Which vertices can we definitely claim we know the shortest path from s?
  - s itself (Why?)
  - The vertex v nearest to s (Why?)
- In general
  - if we know the shortest path distances to a set of vertices
  - how can we determine the shortest path to another vertex?

DIJKSTRA'S ALGORITHM

- At any point in time we know the exact shortest path weight from s
  - to vertices in $X \subset V$ ($s \in X$), and
  - to some vertex $v \in T (= V \setminus X)$ that is closest to some vertex in X

  based on paths going through only vertices in X.
- Thus, expand X by considering only the nearest neighbors of the vertices visited!
- Define $\delta_{G,X}(s, v)$ to be the shortest path length from s to v in G **that only goes through vertices in** X (except for v)

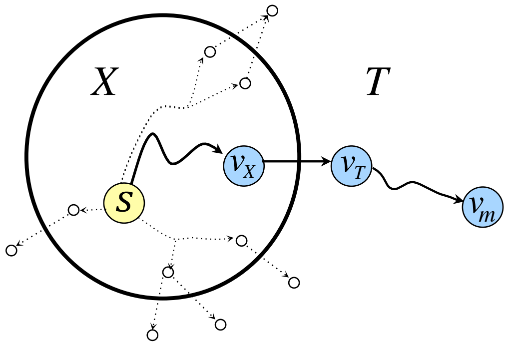
DIJKSTRA'S PROPERTY

- Consider a graph
  - $G = (V, E)$, $w \colon E \to \mathcal{R}_+ \cup \{0\}$,
  - a source vertex $s \in V$
- For any partitioning of the vertices V into X and $T = V \setminus X$ with $s \in X$,

$$\min_{t \in T} \delta_{G,X}(s,t) = \min_{t \in T} \delta_G(s,t) \ .$$

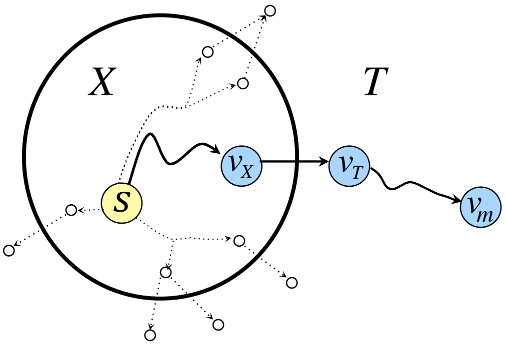
- What is this assertion saying?
  - The actual shortest distance to the vertex in T that is closest to s, has to go through vertices in X!

DIJKSTRA'S PROPERTY



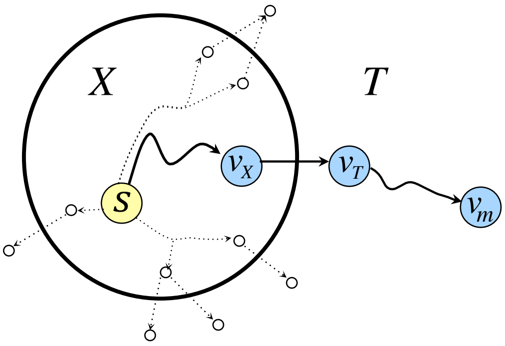
- Consider
  - a vertex $v_m \in T$ such that $\delta_G(s, v_m) = \min_{t \in T} \delta_G(s, t)$, and
  - a shortest path from s to v_m in G.
- The path must cross from X to T at some point using some edge (v_X, v_T).
  - (Prefix) Subpaths of shortest paths are also shortest paths! (Why?)

DIJKSTRA'S PROPERTY



- The subpath from s to v_T is the shortest path to v_T
- Since edges weights are ≥ 0 $\delta_G(s, v_T) \leq \delta_G(s, v_m)$
  - It could be that $v_T = v_m$.
- Also, $\delta_{G,X}(s, v_T) = \delta_G(s, v_T)$ (Why?)

DIJKSTRA'S PROPERTY



$$\min_{t\in T}\delta_{G,X}(s,t)\le\delta_{G,X}(s,v_T)=\delta_G(s,v_T)\le\delta_G(s,v_m)=\min_{t\in T}\delta_G(s,t)\ .$$

But

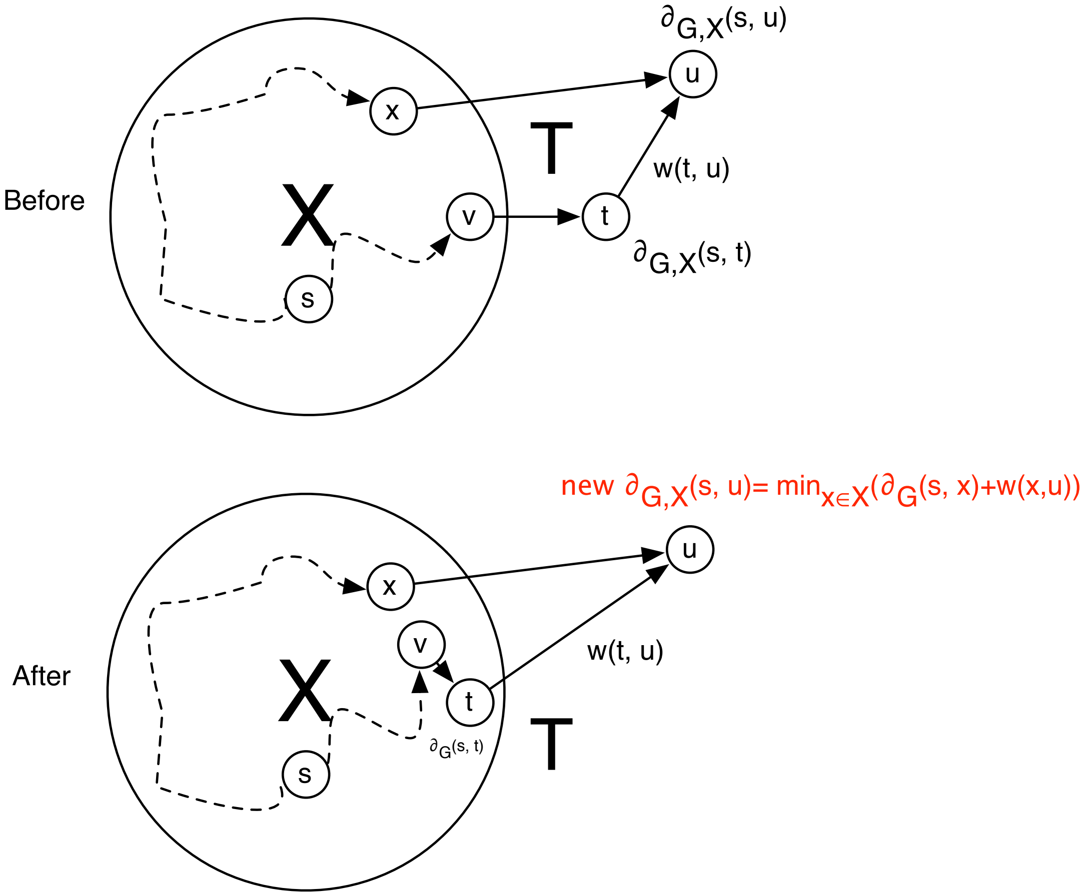
$$\min_{t\in T}\delta_{G,X}(s,t)\ge\min_{t\in T}\delta_G(s,t)\ \text{Why?}$$

$$\Rightarrow\min_{t\in T}\delta_{G,X}(s,t)=\min_{t\in T}\delta_G(s,t)\ .$$

DIJKSTRA'S PROPERTY

- We can find the shortest path to a node in $T = V \setminus X$, by just considering neighbors of X.
- Pick a vertex $t \in T$ that minimizes priority $\delta_{G,X}(s, t)$.
- At that point
    - $\delta_{G,X}(s, t)$ becomes $\delta_G(s, t)$ - we now know the exact shortest path length to t.
    - $X = X \cup \{t\}$, that is, t is now visited.
    - $T = T \setminus \{t\}$
    - $\delta_{G,X}(s, u)$ where $u \in T$ and $(t, u) \in E$ must be updated. (Why?/How?)

DIJKSTRA'S PROPERTY – UPDATES



new $\partial_{G,X}(s, u) = \min_{x \in X}(\partial_G(s, x) + w(x,u))$

DIJKSTRA'S ALGORITHM

- Given a weighted graph $G = (V, E, w)$ and a source s, Dijkstra's algorithm is priority first search on G
  - starting at s, with $d(s) = 0$ (and $d(v) = \infty, v \neq s$)
  - using priority $P(v) = \min_{t \in V}(d(t) + w(t, v))$ (to be minimized)
  - setting $d(v) = P(v)$ when v is visited.

DIJKSTRA'S ALGORITHM

LEMMA

- When Dijkstra's Algorithm returns $d(v) = \delta_G(s, v)$ for all reachable v,

- Base case is true: $d(s) = 0$.
- Assume true for $|X| = i$, then add vertex that minimizes $P(v) = \delta_{G,X}(s, v)$.
- By Dijkstra's Property we know $\min_{v \in T} \delta_{G,X}(s, v) = \min_{v \in T} \delta_G(s, v)$
- So $d(v) = \delta_G(s, v)$ for $|X| = i + 1$.

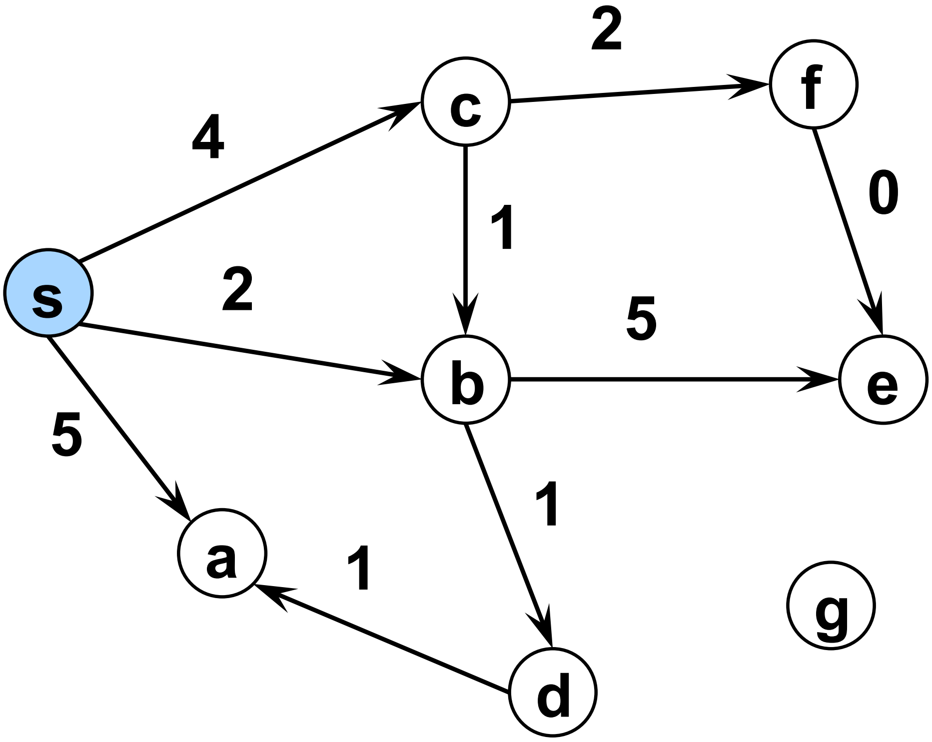
DIJKSTRA'S ALGORITHM

```
fun $\mathtt{dijkstra}(G, s) =$
let
   fun $\mathtt{dijkstra}'(X,\ Q) =$
      case $\boxed{\mathtt{PQ.deleteMin}}(Q)$ of
           $(NONE,\ \_) \Rightarrow X$
         | $(SOME(d, v), Q') \Rightarrow$
           if $(\boxed{(v,\ \_) \in X})$ then $\mathtt{dijkstra}'(X,\ Q')$
           else let
              val $X' = \boxed{X \cup \{(v, d)\}}$
              fun $\mathtt{relax}\ (Q, (u, w)) = \boxed{\mathtt{PQ.insert}}(d + w, u)\ \ Q$
              val $Q'' = \boxed{\mathtt{iter}}\ \ \mathtt{relax}\ \ Q'\ \ \boxed{N_G(v)}$
           in $\mathtt{dijkstra}'(X',\ Q'')$ end
in
   $\mathtt{dijkstra}'(\{\},\ \boxed{\mathtt{PQ.insert}}\ \ (0, s)\ \ \{\})$
end
```

DIJKSTRA VARIANTS

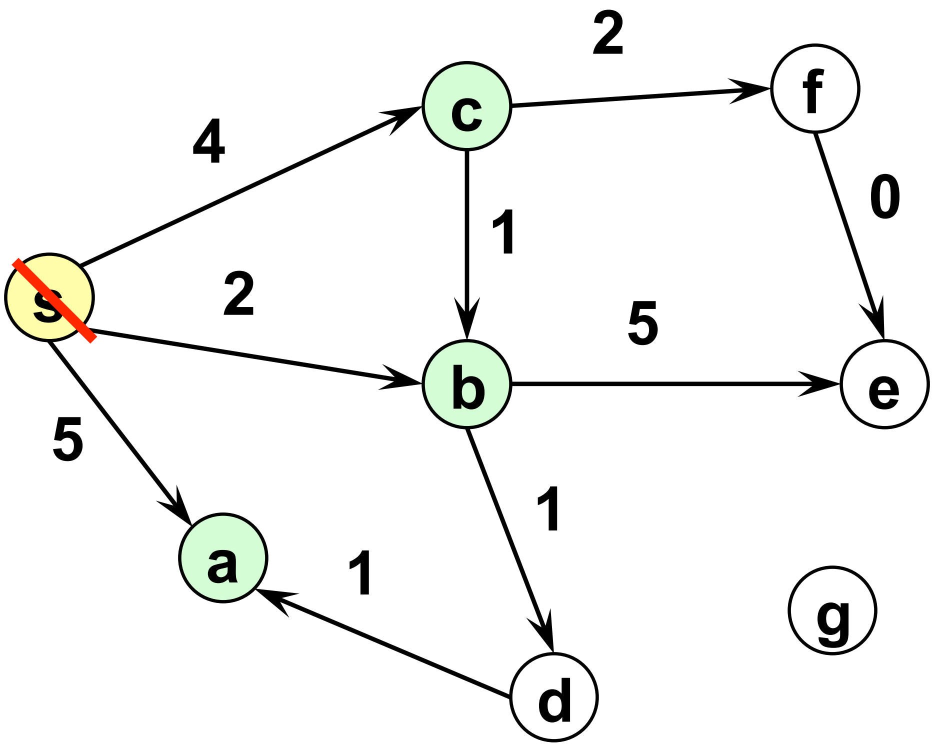
- Update the neighbors in the priority queue instead of adding duplicates.
  - PQ needs to support `decreaseKey` function.
- Visit all equally closest vertices in parallel (like BFS)
  - Potentially not much parallelism!
  - PQ needs to return all such vertices.

DIJKSTRA IN ACTION



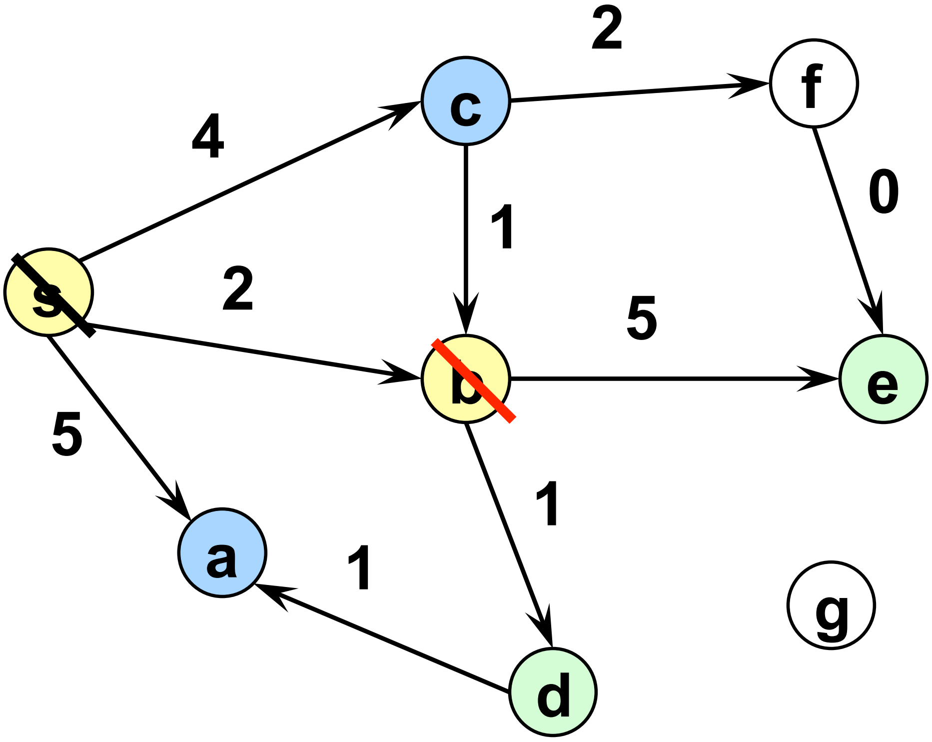
X	Q
	$0 \mapsto s$

DIJKSTRA IN ACTION



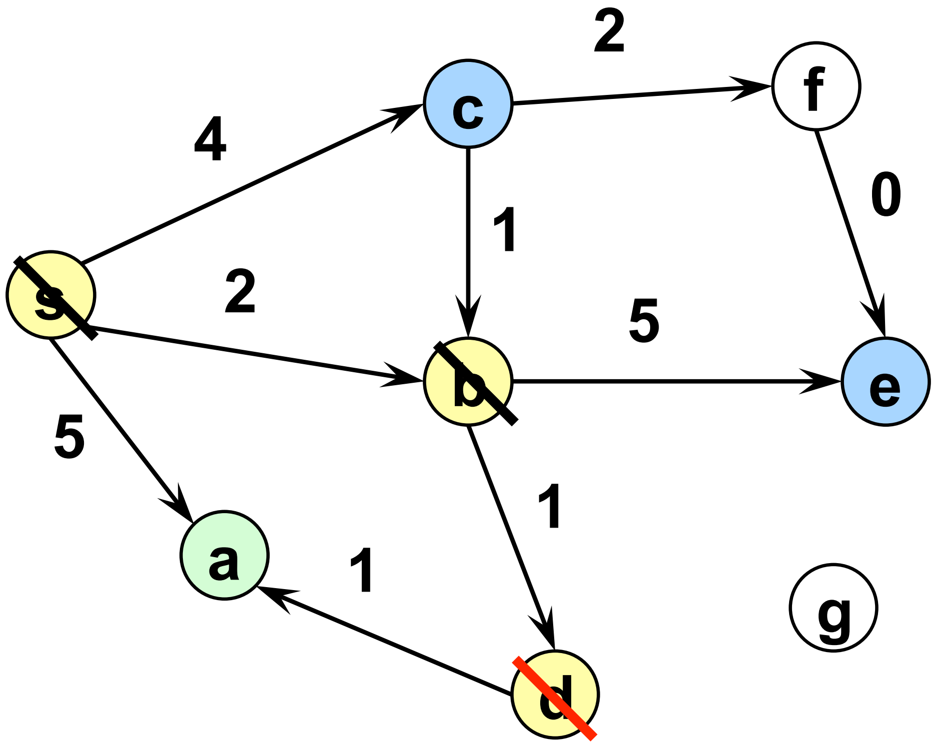
X	Q
$s \mapsto 0$	$2 \mapsto b$
	$4 \mapsto c$
	$5 \mapsto a$

DIJKSTRA IN ACTION



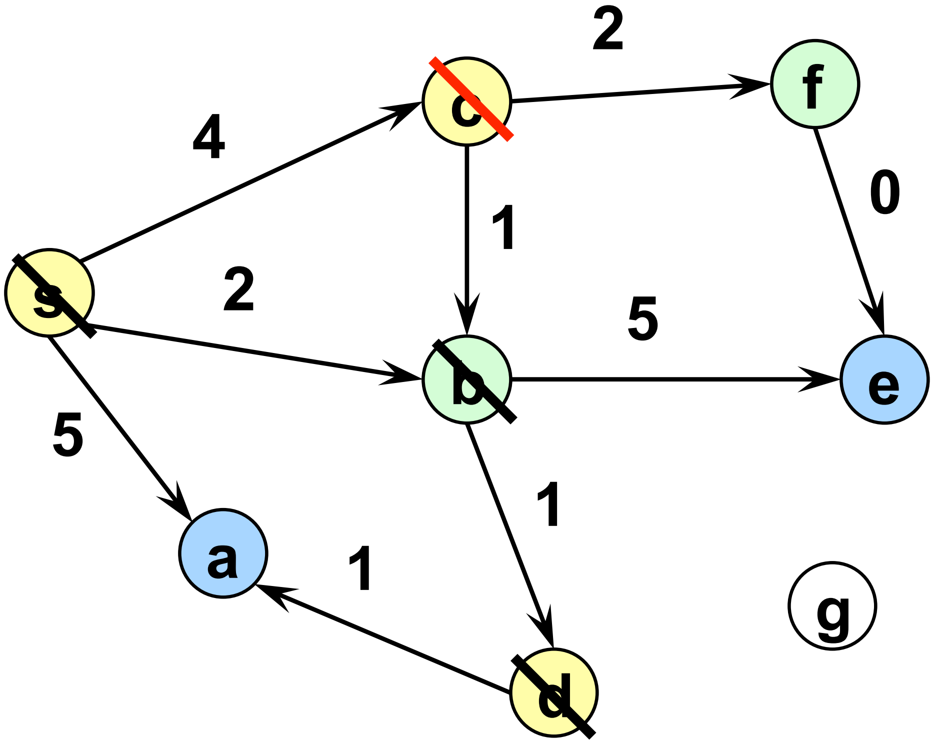
X	Q
s↦0	**3↦d**
b↦2	4↦c
	5↦a
	7↦e

Dijkstra in Action



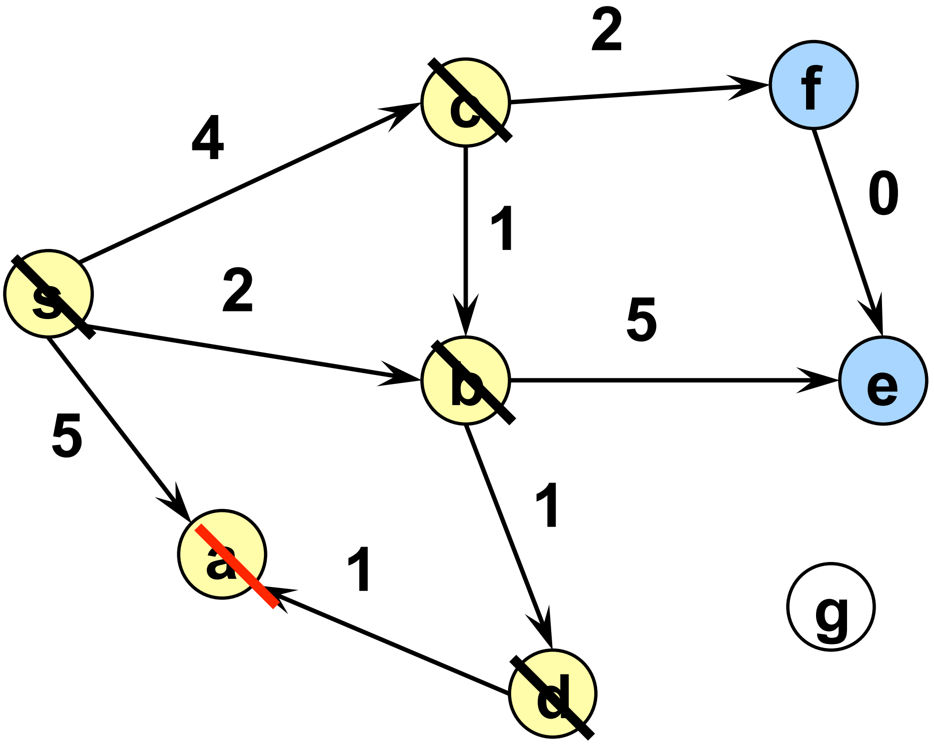
X	Q
s↦0	4↦c
b↦2	**4↦a**
d↦3	5↦a
	7↦e

DIJKSTRA IN ACTION



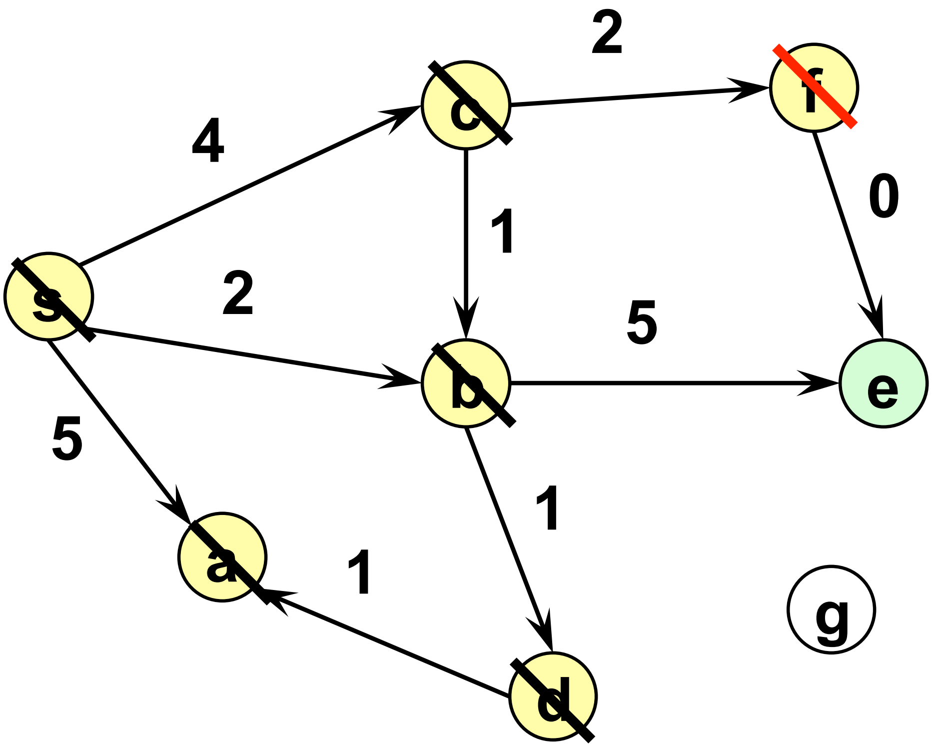
X	Q
s↦0	4↦a
b↦2	5↦a
d↦3	**5↦b**
c↦4	**6↦f**
	7↦e

DIJKSTRA IN ACTION



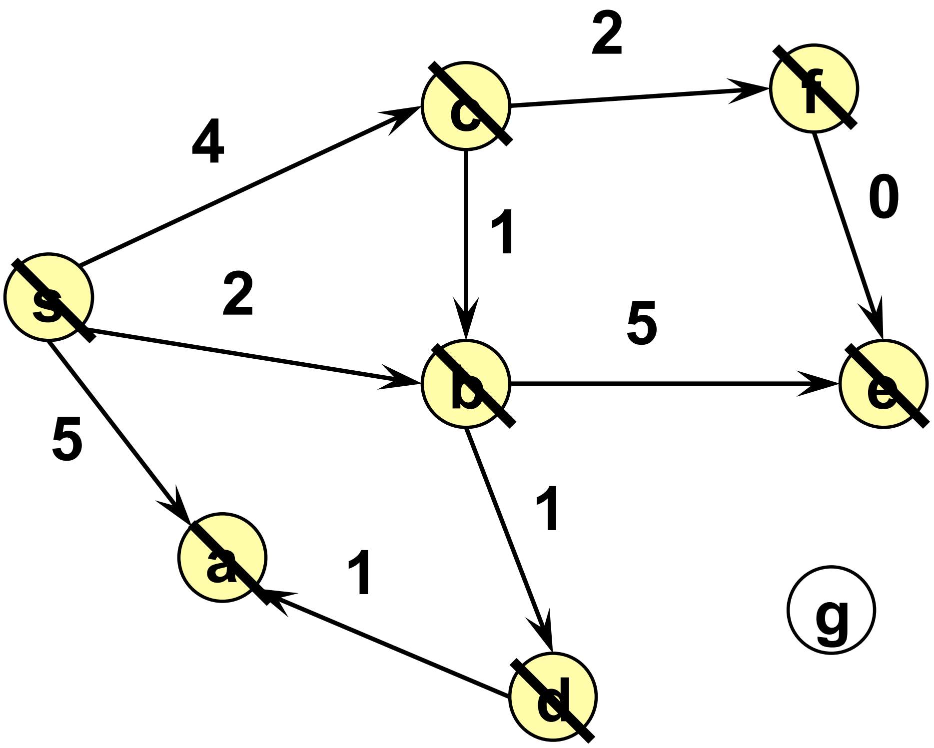
X	Q
s↦0	5↦a
b↦2	5↦b
d↦3	6↦f
c↦4	7↦e
a↦4	

DIJKSTRA IN ACTION



X	Q
s↦0	**6↦e**
b↦2	7↦e
d↦3	
c↦4	
a↦4	
f↦6	

DIJKSTRA IN ACTION



X	Q
s↦0	
b↦2	
d↦3	
c↦4	
a↦4	
f ↦6	
e↦6	

DIJKSTRA'S ALGORITHM

```
fun $\texttt{dijkstra}(G, s) =$
let
  fun $\texttt{dijkstra}'(X,\ Q) =$
    case $\boxed{\texttt{PQ.deleteMin}}(Q)$ of
        $(\mathit{NONE}, \_) \Rightarrow X$
      | $(\mathit{SOME}(d, v), Q') \Rightarrow$
        if $(\boxed{(v, \_) \in X})$ then $\texttt{dijkstra}'(X,\ Q')$
        else let
          val $X' = \boxed{X \cup \{(v, d)\}}$
          fun $\texttt{relax}\ (Q, (u, w)) = \boxed{\texttt{PQ.insert}}(d + w, u)\ \ Q$
          val $Q'' = \boxed{\texttt{iter}}\ \ \texttt{relax}\ \ Q'\ \ \boxed{N_G(v)}$
        in $\texttt{dijkstra}'(X',\ Q'')$ end
in
  $\texttt{dijkstra}'(\{\},\ \boxed{\texttt{PQ.insert}}\ \ (0, s)\ \ \{\})$
end
```

COST ANALYSIS

- Priority Queue with $O(\log n)$ work and span.
- Graphs: tree-based table or arrays
- Table of distances: tree-based table, array sequence, or ST sequence.

Operation	Line	# of calls	PQ	Tree Table	Array	ST Array
`deleteMin`	4	$O(m)$	$O(\log m)$	-	-	-
`insert`	10	$O(m)$	$O(\log m)$	-	-	-
Priority Q total			$O(m \log m)$	-	-	-
`find`	7	$O(m)$	-	$O(\log n)$	$O(1)$	$O(1)$
`insert`	9	$O(n)$	-	$O(\log n)$	$O(n)$	$O(1)$
Distances total			-	$O(m \log n)$	$O(n^2)$	$O(m)$
$N_G(v)$	11	$O(n)$	-	$O(\log n)$	$O(1)$	-
`iter`	11	$O(m)$	-	$O(1)$	$O(1)$	-
Graph access total			-	$O(m + n \log n)$	$O(m)$	-

- Using a tree table work is $O(m \log n)$.

SUMMARY

- Representing weighted graphs.
- Priority-first Search
- Shortest weighted paths
- Dijkstra's Algorithm

15-210
PARALLEL AND SEQUENTIAL ALGORITHMS AND DATA STRUCTURES

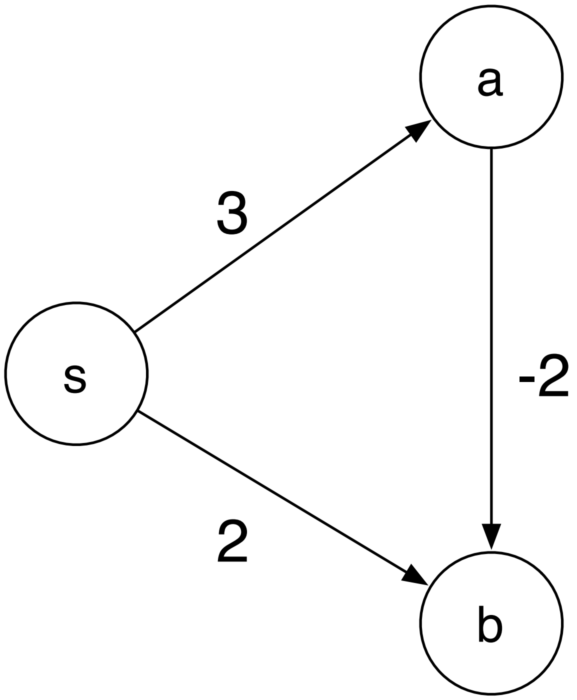
LECTURE 14

SHORTEST WEIGHTED PATHS–II

SYNOPSIS

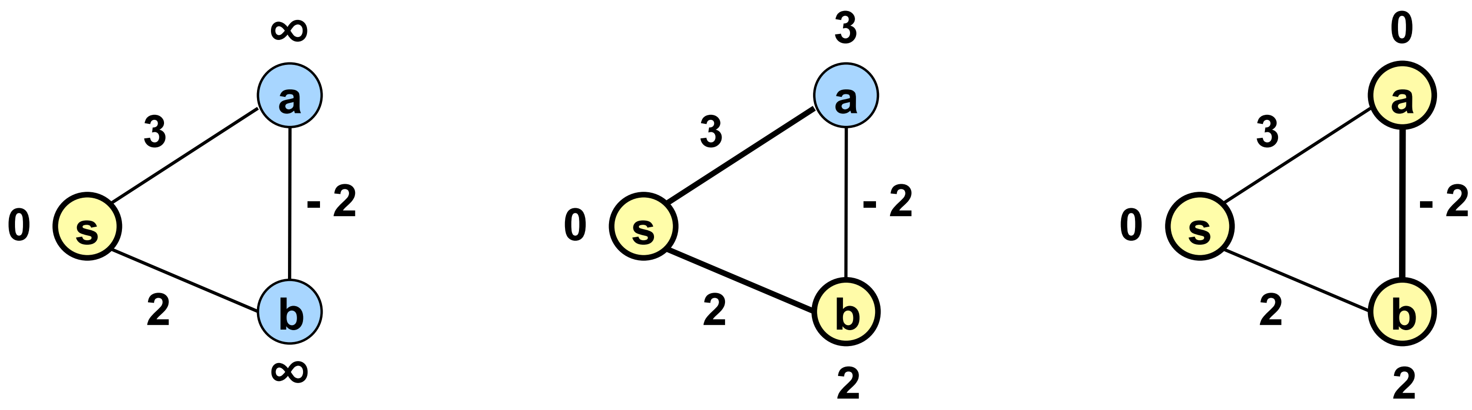
- Graphs with negative edge weights.
- Bellman Ford Algorithm
- Cost Analysis

GRAPHS WITH NEGATIVE WEIGHTS



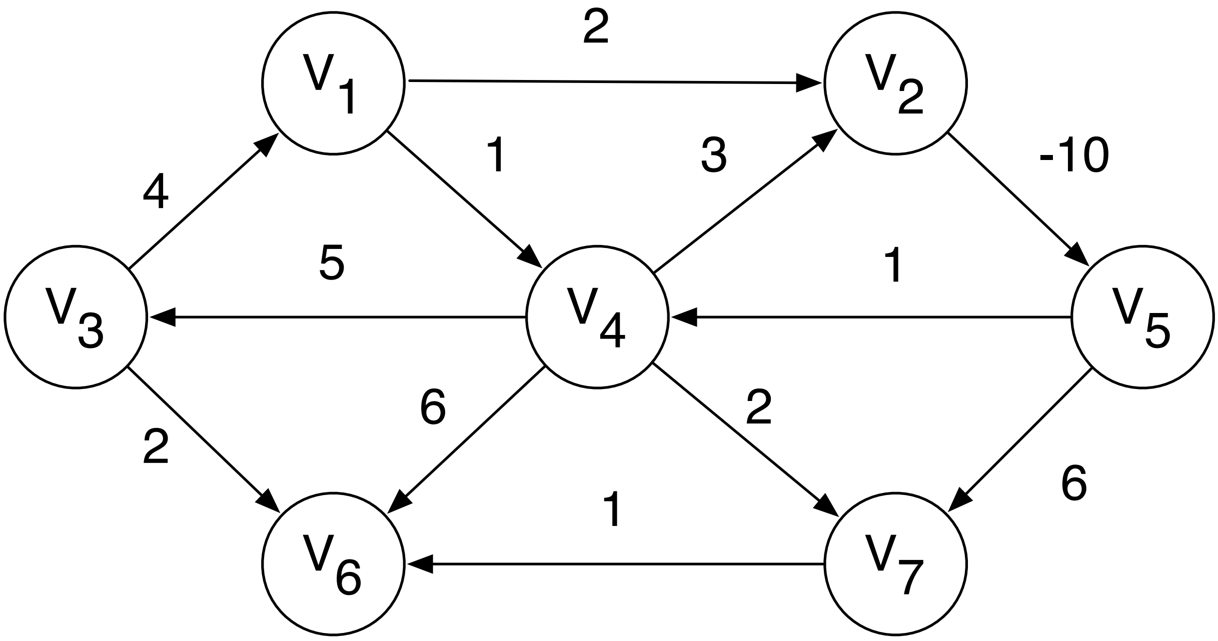
- What is a problem with this graph?

GRAPHS WITH NEGATIVE WEIGHTS



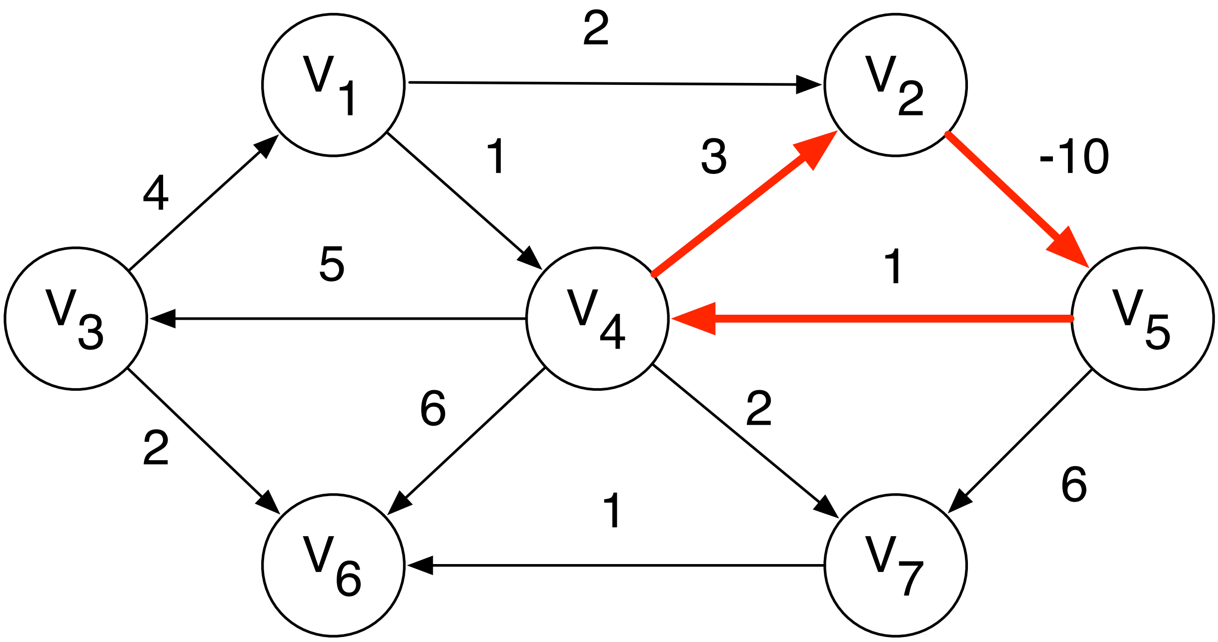
- Dijkstra fails! (Why?)

GRAPHS WITH NEGATIVE WEIGHTS



- What is a problem with this graph?

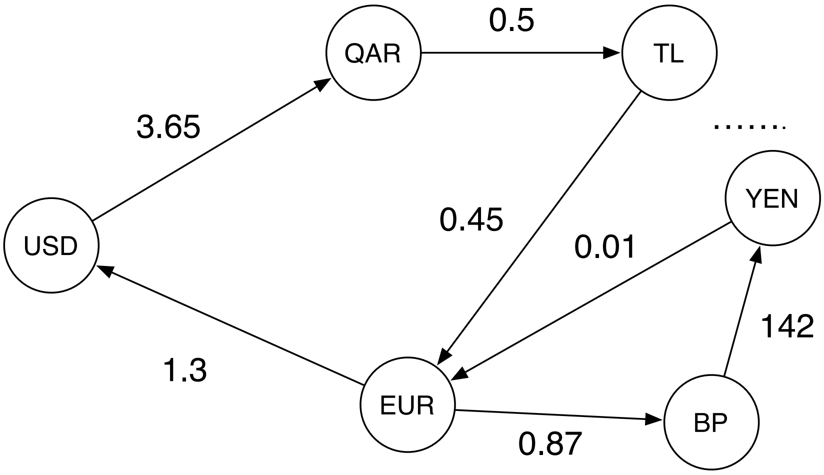
GRAPHS WITH NEGATIVE WEIGHTS



- Negative cost cycle!
- There is no shortest path from v_3 to v_5

GRAPHS WITH NEGATIVE WEIGHTS

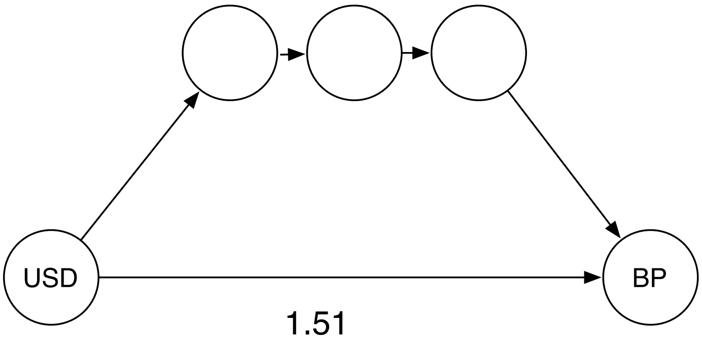
- Currency Exchange Arbitrage



- 100 USD → 365 QAR → 177.5 TL → 80.68 EUR → 104.9 USD
  - You just made 5 USD out of thin air!

GRAPHS WITH NEGATIVE WEIGHTS

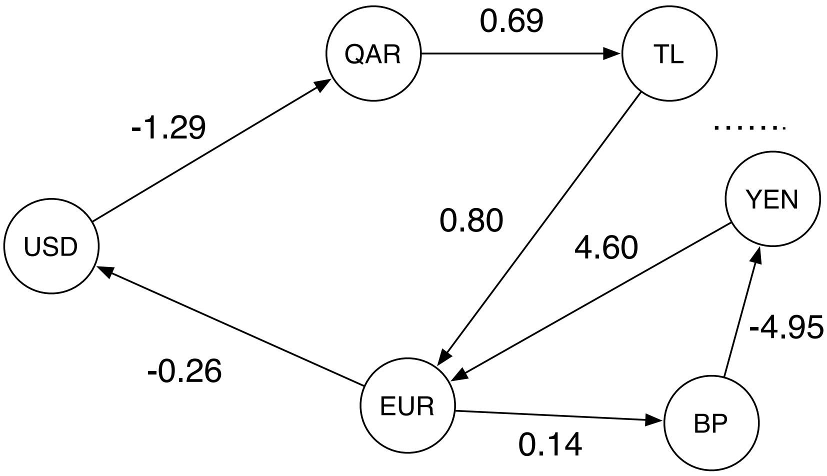
- I have USDs but I want to buy BPs.
  - I can buy directly, or
  - I can buy through some intermediate currencies!



- Which way will get me more BPs?
- I need to do this fast!

SHORTEST PATHS

- How does this problem relate to the shortest problem?
  - Where are the negative weights?



- Weights are $-$ log of the exchange rates!

SHORTEST PATHS WITH NEGATIVE WEIGHTS

- Define $\delta_G^l(s,t)$ the shortest weighted path from s to t using at most l edges.
    - so the unweighted path length is l!
- Base cases:
    - $\delta_G^0(s,s) = 0$
    - $\delta_G^0(s,v) = \infty$ for all $v \neq s$.
- Induction

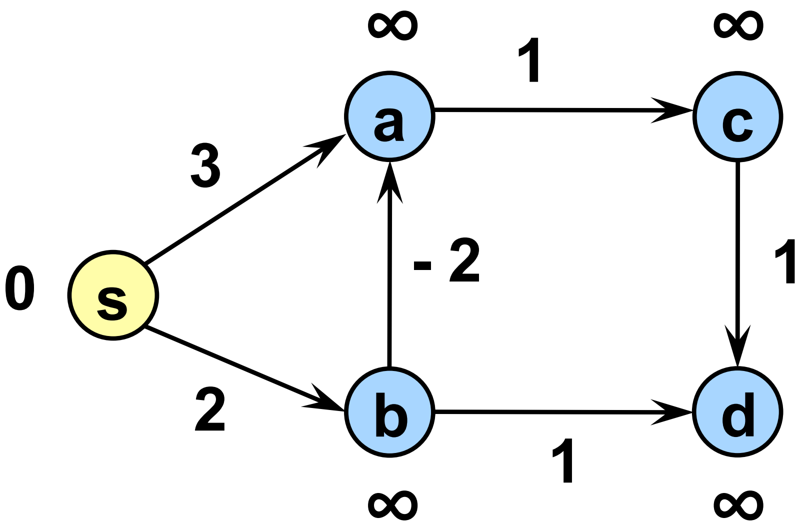
$$\delta^{k+1}(v) = \min_{x \in N^-(v)} \left(\delta^k(x) + w(x,v) \right) .$$

- Minimum of $\delta^k(x) + w(x,v)$ over the in-neighbors.

THE BELLMAN FORD ALGORITHM

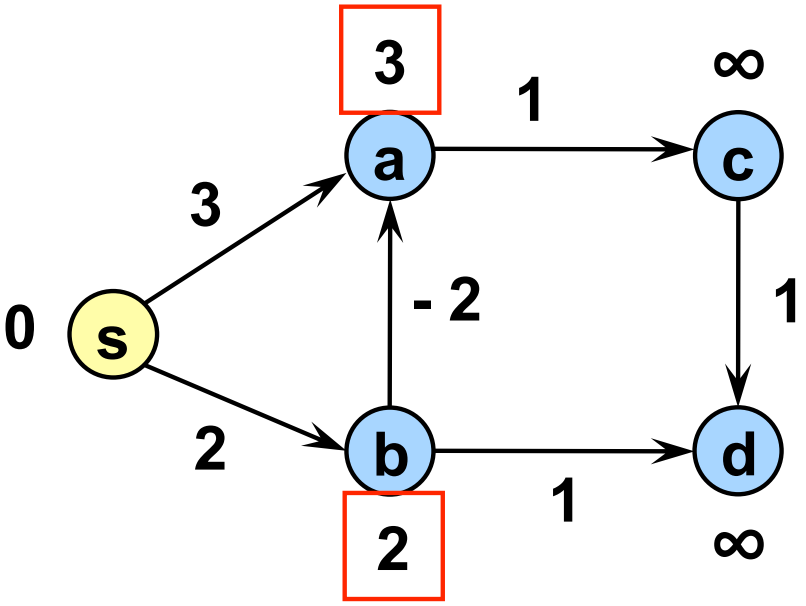
```
fun BellmanFord$(G = (V, E), s) =$
let
    fun $BF(D, k) =$
        let
            val $D' = \{v \mapsto \min_{u \in N_G^-(v)}(D_u + w(u, v)) : v \in V\}$
        in
            if $(k = |V|)$ then $\perp$
            else if (all$\{D_v = D'_v : v \in V\}$) then $D$
            else $BF(D', \; k + 1)$
        end

    val $D = \{v \mapsto$ if $v = s$ then $0$ else $\infty : v \in V\}$
in $BF(D, 0)$ end
```

HOW BELLMAN FORD ALGORITHM WORKS



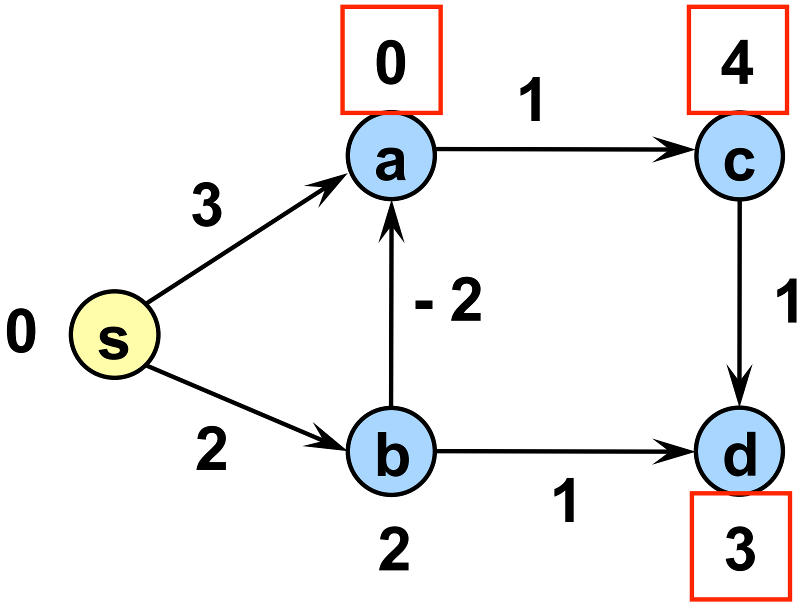
path lengths = 0

HOW BELLMAN FORD ALGORITHM WORKS



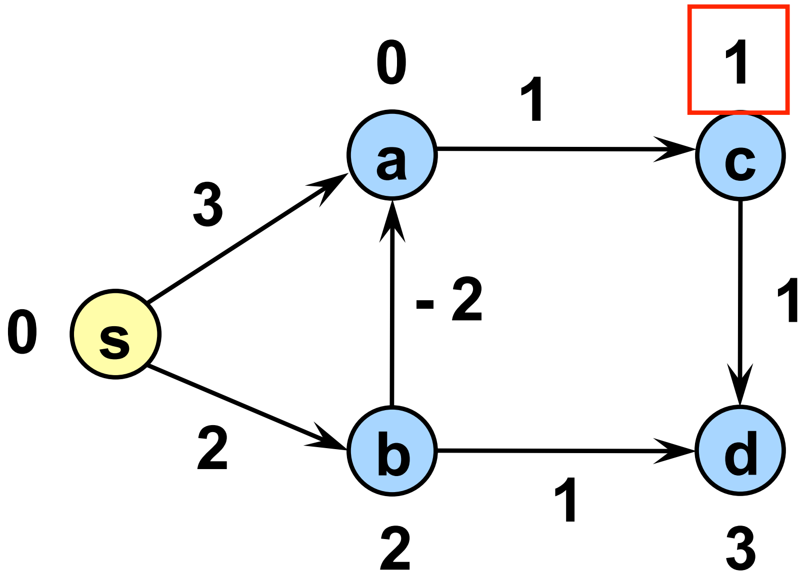
path lengths ≤ 1

HOW BELLMAN FORD ALGORITHM WORKS



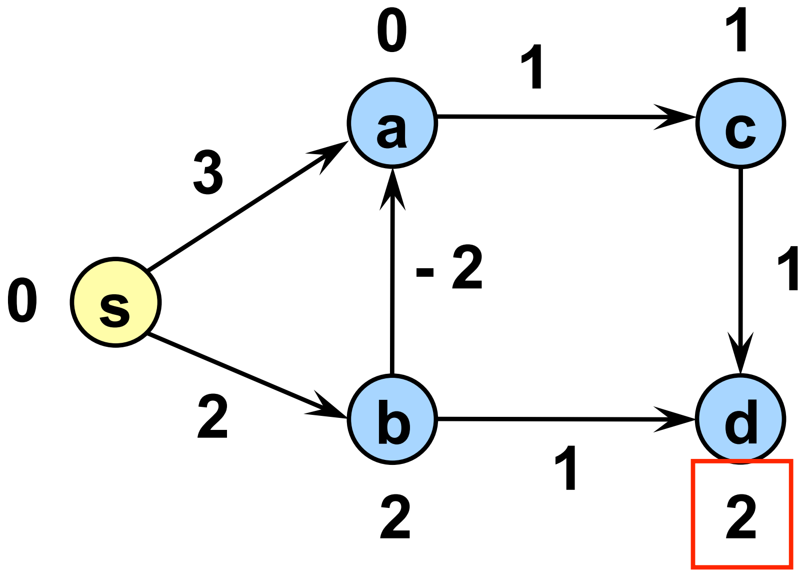
path lengths ≤ 2

HOW BELLMAN FORD ALGORITHM WORKS



path lengths ≤ 3

HOW BELLMAN FORD ALGORITHM WORKS



path lengths ≤ 4

BELLMAN FORD CORRECTNESS

THEOREM

- Given a directed weighted graph $G = (V, E, w)$, $w : E \rightarrow R$, and a source s, the `BellmanFord` algorithm returns the shortest path length from s to every vertex **or** indicates that there is a negative weight cycle in G reachable from s.

BELLMAN FORD CORRECTNESS

- Use induction on the the number of edges k in the path.
- Base case is correct, $D_s = 0$.
- On each step, for all $v \in V \setminus \{s\}$, a shortest path with at most $k + 1$ edges
  - must consist of a path of at most k edges for vertex u
  - followed by a single edge (u, v).
- Taking the minimum combination, gives us the shortest path with at most $k + 1$ edges.

NEGATIVE COST CYCLES

- This can go at most for $n = |V| - 1$ rounds
- If we reach round n, there must be reachable negative cost cycle.
- Otherwise, Bellman Ford will stop earlier with all simple shortest paths.

COST ANALYSIS

- Graph represented as a table.
  - ($\mathbb{R}$ `vtxTable`) `vtxTable`, where first `vtxTable` maps vertices to their in-neighbors

$$G = \{0 \mapsto \{1 \mapsto 0.7,\ 2 \mapsto 1.5\},\ 1 \mapsto \{2 \mapsto -2.0\},\ 2 \mapsto \{\}\}\ .$$

- Graph represented as a sequence of sequences.
  - $((\mathit{int} \times \texttt{eVal})\ \texttt{seq})\ \texttt{seq}$

BELLMAN - FORD ALGORITHM (AGAIN)

```
1   fun  BellmanFord$(G = (V, E), s) =$
2   let
3       fun  $BF(D, k) =$
4           let
5               val  $D' = \{v \mapsto \min_{u \in N_G^-(v)}(D_u + w(u, v)) : v \in V\}$
6           in
7               if  $(k = |V|)$  then  $\perp$
8               else  if  (all$\{D_v = D'_v : v \in V\}$)  then  $D$
9               else  $BF(D',\ k + 1)$
10          end
11      val  $D = \{v \mapsto$ if  $v = s$  then  $0$  else  $\infty : v \in V\}$
12  in  $BF(D, 0)$  end
```

- Line 5 is tabulate over the vertices
- Line 8 is tabulate with a reduction over the vertices

COST ANALYSIS

$$\textbf{val } D' = \{v \mapsto \min_{u \in N_G^-(v)}(D_u + w(u,v)) : v \in V\}$$

- Sum work and max span over vertices.
- $n = |V|$ and $m = |E|$
- For each vertex we have the following costs:
    - Find the neighbors `find G v`: $O(\log n)$ work and span.
    - Map over neighbors – find distance D_u and add: $O(\log n)$ work and span for each u in the in-neigborhood.
    - Min reduce: $O(1 + N_G(v)|)$ work and $O(\log |N_G(v)|)$ span.

WORK PER STAGE-1

val $D' = \{v \mapsto \min_{u \in N_G^-(v)}(D_u + w(u,v)) : v \in V\}$

Operation	**Over one vertex v**	**Over graph G**
Find	$O(\log n)$	$O(n \log n)$
Map	$O(1 + \lvert N_G^-(v)\rvert \log n)$	$O(n + m \log n)$
Min Reduce	$O(1 + \lvert N_G^-(v)\rvert)$	$O(n + m)$

- Total work is $O((n+m)\log n)$ and assuming $m > n$, $O(m \log n)$

SPAN PER STAGE-1

val $D' = \{v \mapsto \min_{u \in N_G^-(v)}(D_u + w(u,v)) : v \in V\}$

Operation	Over one vertex v	Over graph G
Find	$O(\log n)$	$O(\log n)$
Map	$O(1 + \log n)$	$O(1 + \log n)$
Min Reduce	$O(\log \lvert N_G^-(v) \rvert)$	$O(\log n)$

- Total span is $O(\log n)$

WORK / SPAN PER STAGE - 2 – TOTAL COST

else if (all $\{D_v = D'_v : v \in V\}$) **then** D

- This involves a tabulate and an and-reduction.
- Work $= O(n \log n)$, Span $= O(\log n)$

- n sequential calls to BF, so total costs are:

$$\begin{aligned} W(n,m) &= O(n \cdot m \log n) \\ S(n,m) &= O(n \log n) \end{aligned}$$

COSTS WITH ST SEQUENCES

- We use IL (integer labeled) graphs.
- `find` → `nth`: $O(1)$ work.
- Similar improvements for looking up neighbors and distance table.

$$
\begin{aligned}
W(n,m) &= O(nm) \\
S(n,m) &= O(n)
\end{aligned}
$$

SUMMARY

- Graphs with negative edge weights.
- Bellman Ford Algorithm
- Analysis

15-210
PARALLEL AND SEQUENTIAL ALGORITHMS AND DATA STRUCTURES

LECTURE 15

PROBABILITY AND RANDOMIZED ALGORITHMS

SYNOPSIS

- Overview of Discrete Probability
- Finding the two largest elements
- Find the k^{th} smallest element.

RANDOMIZED ALGORITHMS

- Exploit randomness during computation
  - Pivot selection in Quicksort
  - Average case analysis
  - Primality testing
- Question: How many comparisons are needed to find the *second* largest number on a sequence of n numbers?
  - Naive algorithm: $2n - 3$ comparisons
  - Divide and Conquer algorithm: $3n/2$ comparisons
  - Simple randomized algorithm: $n - 1 + 2\log n$ comparisons *on the average*.

OVERVIEW OF DISCRETE PROBABILITY

- **Probabilistic Experiment**: outcome is probabilistic.
- **Sample Space (Ω)**: arbitrary and possibly countably infinite set of possible outcomes.
  - Tossing a coin
  - Throwing a die/pair of dice.
- **Primitive Event**: Any one of the elements of Ω.
- **Event**: Any subset of Ω
  - First die is a 5
  - Dice sum to 7
  - Any die is even.

PROBABILITY FUNCTION

- Probability Function: $\Omega \to [0, 1]$

$$\sum_{e \in \Omega} \mathbf{Pr}[e] = 1$$

- Probability of an event A:

$$\sum_{e \in A} \mathbf{Pr}[e]$$

  - Probability of “first die is 4”?
  - Probability of “dice sum to to 4”?

RANDOM VARIABLES

- **Random Variable**: $X : \Omega \to \Re$
  - X is the sum of the two die rolls
- **Indicator Random Variable**: $Y : \Omega \to \{0, 1\}$
  - Y is 1 if the dice are the same, 0 otherwise
  - Y is 1 if the total is larger than 7, 0 otherwise
- For $a \in \Re$, the event "$X = a$" is the set

$$\{\omega \in \Omega \mid X(\omega) = a\}$$

EXPECTATION

- The expectation of a random variable

$$\mathop{\mathbf{E}}_{\Omega,\mathbf{Pr}[]}[X] = \sum_{e \in \Omega} X(e) \cdot \mathbf{Pr}[e] \,.$$

- The expectation of an *indicator* random variable:

$$\mathbf{E}[Y] = \sum_{e \in \Omega, p(e) = \mathtt{true}} \mathbf{Pr}[e] = \sum_{e \in \Omega} \mathbf{Pr}[\{e \in \Omega \mid p(e)\}] \,.$$

  - $p : \Omega \to \mathtt{bool}$

INDEPENDENCE

- Events A and B are independent if the occurence of one does not affect the probability of the other

$$\mathbf{Pr}[A \cap B] = \mathbf{Pr}[A] \cdot \mathbf{Pr}[B]$$

  - $A = \{(d_1, d_2) \in \Omega \mid d_1 = 1\}$ and $B = \{(d_1, d_2) \in \Omega \mid d_2 = 1\}$ are independent.
  - $C = \{(d_1, d_2) \in \Omega \mid d_1 + d_2 = 4\}$ is NOT independent of A (Why?)

INDEPENDENCE

- Events $A_1, \ldots, A_k$ are *mutually independent* if and only if for any non-empty subset $I \subseteq \{1, \ldots, k\}$,

$$\mathbf{Pr}[\bigcap_{i \in I} A_i] = \prod_{i \in I} \mathbf{Pr}[A_i].$$

- Random variable X and Y are independent if fixing one does NOT affect the probability distribution of the other.
  - $X =$ "value of the first die" is independent of $Y =$ "value of the second die".
  - X is NOT independent of $Z =$ "sum of the dice"

LINEARITY OF EXPECTATIONS

- Important Theorem: given two random variables X and Y

$$\mathbf{E}\left[X\right] + \mathbf{E}\left[Y\right] = \mathbf{E}\left[X + Y\right]$$

- Easy to show!

$$\sum_{e\in\Omega} \mathbf{Pr}[e]X(e) + \sum_{e\in\Omega} \mathbf{Pr}[e]Y(e) = \sum_{e\in\Omega} \mathbf{Pr}[e](X(e) + Y(e))$$

- Expected sum of two dice
    - Consider 36 outcomes and take average
    - Sum expectations for each dice ($3.5 + 3.5 = 7$)

LINEARITY OF EXPECTATIONS

- In general, for a binary function f the equality

  $$f(\mathbf{E}[X], \mathbf{E}[Y]) = \mathbf{E}[f(X, Y)]$$

  is **not** true in general.
  - $\max(\mathbf{E}[X], \mathbf{E}[Y]) \neq \mathbf{E}[\max(X, Y)]$
  - What is $\mathbf{E}[\max(X, Y)]$?
- $\mathbf{E}[X] \times \mathbf{E}[Y] = \mathbf{E}[X \times Y]$ is true if X and Y are independent.

EXAMPLES

- Toss n coins with probability of heads, p. What is the expected value of X, the number of heads?

$$
\begin{aligned}
\mathbf{E}[X] &= \sum_{k=0}^{n} k \cdot \mathbf{Pr}[X = k] \\
&= \sum_{k=1}^{n} k \cdot p^k (1-p)^{n-k} \binom{n}{k} \text{(Why?)} \\
&= \sum_{k=1}^{n} k \cdot \frac{n}{k} \binom{n-1}{k-1} p^k (1-p)^{n-k} \qquad \left[\text{because } \binom{n}{k} = \frac{n}{k}\binom{n-1}{k-1}\right] \\
&= n \sum_{k=1}^{n} \binom{n-1}{k-1} p^k (1-p)^{n-k}
\end{aligned}
$$

EXAMPLES

- Toss n coins with probability of heads, p. What is the expected value of X, the number of heads?

$$\begin{aligned}
\mathbf{E}[X] &= \sum_{k=0}^{n} k \cdot \mathbf{Pr}[X = k] \\
&\cdots \\
&= n \sum_{j=0}^{n-1} \binom{n-1}{j} p^{j+1} (1-p)^{n-(j+1)} && \text{[because } k = j + 1 \text{]} \\
&= n \cdot p \sum_{j=0}^{n-1} \binom{n-1}{j} p^{j} (1-p)^{(n-1)-j} \\
&= n \cdot p \cdot (p + (1-p))^{n-1} && \text{[Binomial Theorem]} \\
&= n \cdot p
\end{aligned}$$

EXAMPLES

- Toss n coins with probability of heads, p. What is the expected value of X, the number of heads?
- Using linearity of expectations.
  - $X_i = \mathbb{I}\{i\text{-th coin turns up heads}\}$
  - $X = \sum_{i=1}^{n} X_i$

$$\mathbf{E}[X] = \mathbf{E}\left[\sum_{i=1}^{n} X_i\right] = \sum_{i=1}^{n} \mathbf{E}[X_i] = \sum_{i=1}^{n} p = n \cdot p$$

  - because $\mathbf{E}[X_i] = p$.

EXAMPLES

- A coin has a probability p of coming up heads. What is the expected value of Y representing the number of flips until we see a head?
- Write a recurrence!
  - With probability p, we'll get a head and we are done,
  - With probability $1 - p$, we'll get a tail and we'll go back to square one

$$
\begin{aligned}
\mathbf{E}[Y] &= p \cdot 1 + (1 - p)\Big(1 + \mathbf{E}[Y]\Big) \\
&= 1 + (1 - p)\,\mathbf{E}[Y] \implies \mathbf{E}[Y] = 1/p.
\end{aligned}
$$

FINDING THE TOP TWO ELEMENTS

```
fun $max2(S)$ = let
  fun $replace((m_1, m_2), v)$ =
    if $v \leq m_2$ then $(m_1, m_2)$
    else if $v \leq m_1$ then $(m_1, v)$
    else $(v, m_1)$

  val $start$ = if $S_1 \geq S_2$ then $(S_1, S_2)$ else $(S_2, S_1)$

in $iter\ replace\ start\ S\langle 3, \ldots, n\rangle$
end
```

- We will do exact analysis.
- $1 + 2(n - 2) = 2n - 3$ comparisons in the worst case. (Why?)
- A Divide and Conquer algorithm gives $3n/2 - 2$

WORST CASE ANALYSIS

```
fun max2($S$) = let
  fun replace(($m_1, m_2$), $v$) =
    if $v \leq m_2$ then ($m_1, m_2$)
    else if $v \leq m_1$ then ($m_1, v$)
    else ($v, m_1$)

  val start = if $S_1 \geq S_2$ then ($S_1, S_2$) else ($S_2, S_1$)

in iter replace start $S\langle 3, \ldots, n\rangle$
end
```

- An already sorted sequence (e.g., $\langle 1, 2, 3, \ldots, n\rangle$) will need exactly $2n - 3$ comparisons.
- But this happens with $1/n!$ chance!

A RANDOMIZED ALGORITHM

- The worst-case analysis is overly pessimistic.
- Consider the following variant:
- On input of a sequence S of n elements:
  1. Let $T = \texttt{permute}(S, \pi)$, where π is a random permutation (i.e., we choose one of the $n!$ permutations).
  2. Run the naïve algorithm on T.
- No need to really generate the permutation!
  - Just pick an unprocessed element randomly until all elements are processed.
  - It is convenient to model this by one initial permutation!

ANALYSIS

```
1 fun $max2(S)$ = let
2    fun $replace((m_1, m_2), v)$ =
3       if $v \le m_2$ then $(m_1, m_2)$
4       else if $v \le m_1$ then $(m_1, v)$
5       else $(v, m_1)$

6    val $start$ = if $S_1 \ge S_2$ then $(S_1, S_2)$ else $(S_2, S_1)$

7 in $iter\ replace\ start\ S\langle 3, \ldots, n\rangle$
8 end
```

- $X_i = 1$ if T_i is compared in Line 4, 0 otherwise.
- Y is the number of comparisons

$$Y = \underbrace{1}_{\text{Line 6}} + \underbrace{n-2}_{\text{Line 3}} + \underbrace{\sum_{i=3}^{n} X_i}_{\text{Line 4}};$$

ANALYSIS

- This expression in true regardless of the random choice we're making.
- We're interested in computing the expected value of Y.
- By linearity of expectation,

$$\begin{aligned}
\mathbf{E}\left[Y\right] &= \mathbf{E}\left[1 + (n-2) + \sum_{i=3}^{n} X_i\right] \\
&= 1 + (n-2) + \sum_{i=3}^{n} \mathbf{E}\left[X_i\right].
\end{aligned}$$

ANALYSIS

- Problem boils down to computing $\mathbf{E}[X_i]$, for $i = 3, \ldots, n$!
- What is the probability that $T_i > m_2$?
  - $T_i > m_2$ holds when T_i is either the largest or the second largest in $\{T_1, \ldots, T_i\}$
- So, what is the probability that T_i is one of the two largest elements in a randomly permuted sequence of length i?
  - $\frac{1}{i} + \frac{1}{i} = \frac{2}{i}$
- $\mathbf{E}[X_i] = 1 \cdot \frac{2}{i} = 2/i$

ANALYSIS

$$\begin{aligned}
\mathbf{E}[Y] &= 1 + (n-2) + \sum_{i=3}^{n} \mathbf{E}[X_i] \\
&= 1 + (n-2) + \sum_{i=3}^{n} \frac{2}{i} \\
&= 1 + (n-2) + 2\left(\tfrac{1}{3} + \tfrac{1}{4} + \ldots \tfrac{1}{n}\right) \\
&= n - 4 + 2\left(1 + \tfrac{1}{2} + \tfrac{1}{3} + \tfrac{1}{4} + \ldots \tfrac{1}{n}\right) \\
&= n - 4 + 2H_n
\end{aligned}$$

- H_n is the n^{th} Harmonic number
- $H_n \leq 1 + \log_2 n$
- $\mathbf{E}[Y] \leq n - 2 + 2\log_2 n$

FINDING THE k^{th} SMALLEST ELEMENT

- **Input:** a sequence of n numbers (not necessarily sorted)
- **Output:** the k^{th} smallest value in S (i.e., `(nth (sort S) k)`).
- *Requirement:* $O(n)$ expected work and $O(\log^2 n)$ span.

- We can't really sort the sequence!

FINDING THE k^{th} SMALLEST ELEMENT

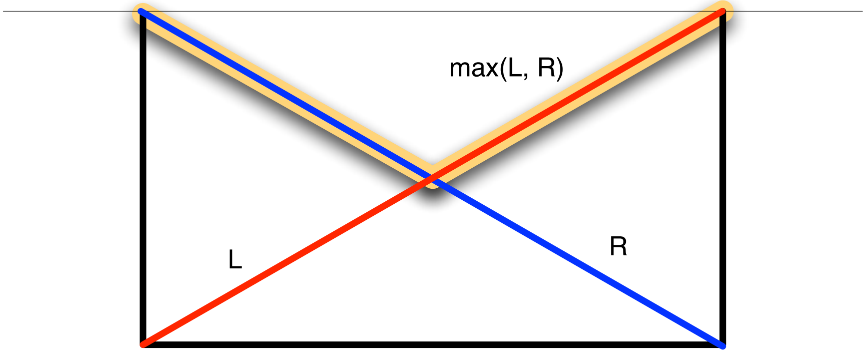
```
fun kthSmallest($(k, S)$) = let
        val $p$ = a value from $S$ picked uniformly at random
        val $L = \langle x \in S \mid x < p \rangle$
        val $R = \langle x \in S \mid x > p \rangle$
    in if $(k < |L|)$ then kthSmallest($(k, L)$)
        else if $(k < |S| - |R|)$ then $p$
        else kthSmallest($(k - (|S| - |R|), R)$)
```

- Let $X_n = \max\{|L|, |R|\}$

$$
\begin{aligned}
W(n) &= W(X_n) + O(n) \\
S(n) &= S(X_n) + O(\log n)
\end{aligned}
$$

FINDING THE k^{th} SMALLEST ELEMENT

- We want to find $\mathbf{E}[X_n]$?



$$\mathbf{E}[X_n] = \sum_{i=1}^{n-1} \max\{i, n-i\} \cdot \tfrac{1}{n} \leq \sum_{j=n/2}^{n-1} \frac{2}{n} \cdot j \leq \frac{3n}{4}$$

FINDING THE k^{th} SMALLEST ELEMENT

- $\mathbf{E}[X_n] \leq \frac{3n}{4} \Rightarrow$ geometrically decreasing sum $\Rightarrow O(n)$ work.
- What is $\mathbf{Pr}[X_n \leq \frac{3}{4}n]$?
- Since $|R| < n - |L|$,

$$X_n \leq \frac{3}{4}n \Leftrightarrow n/4 < |L| \leq 3n/4$$

and the probability is

$$\frac{3n/4 - n/4}{n} = \frac{n/2}{n} = \frac{1}{2}$$

FINDING THE k^{th} SMALLEST ELEMENT

$$\overline{W}(n) = \sum_i \mathbf{Pr}[X_n = i] \cdot \overline{W}(i) + c \cdot n$$

Using stepwise approximation

$$\leq \mathbf{Pr}[X_n \leq \tfrac{3n}{4}]\overline{W}(3n/4) + \mathbf{Pr}[X_n > \tfrac{3n}{4}]\overline{W}(n) + c \cdot n$$

$$= \tfrac{1}{2}\overline{W}(3n/4) + \tfrac{1}{2}\overline{W}(n) + c \cdot n$$

$$\implies (1 - \tfrac{1}{2})\overline{W}(n) = \tfrac{1}{2}\overline{W}(3n/4) + c \cdot n$$

$$\implies \overline{W}(n) \leq \overline{W}(3n/4) + 2c \cdot n$$

- Root Dominated hence solves to $O(n)$.

FINDING THE k^{th} SMALLEST ELEMENT

$$S(n) = S(X_n) + O(\log n)$$

$$\begin{aligned}
\overline{S}(n) &=\leq \sum_i \mathbf{Pr}[X_n = i] \cdot \overline{S}(i) + c \log n \\
&\leq \mathbf{Pr}[X_n \leq \tfrac{3n}{4}]\overline{S}(3n/4) + \mathbf{Pr}[X_n > \tfrac{3n}{4}]\overline{S}(n) + c \cdot \log n \\
&\leq \tfrac{1}{2}\overline{S}(3n/4) + \tfrac{1}{2}\overline{S}(n) + c \cdot \log n \\
&\implies (1 - \tfrac{1}{2})\overline{S}(n) \leq \tfrac{1}{2}\overline{S}(3n/4) + c \log n \\
&\implies \overline{S}(n) \leq \overline{S}(3n/4) + 2c \log n
\end{aligned}$$

- This solves to $O(\log^2 n)$.

15-210
PARALLEL AND SEQUENTIAL ALGORITHMS AND DATA STRUCTURES

LECTURE 16

GRAPH CONTRACTION

SYNOPSIS

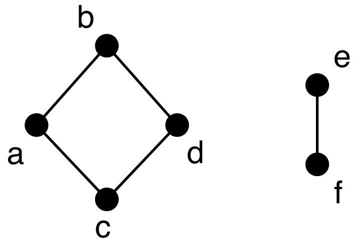
- Graph Contraction
- Finding Connected Components
- Edge Contraction
- Star Contraction

MOTIVATION

- Most graph search algorithms were either
  - sequential, or
  - had span dependent on the diameter.
- Can we make these algorithms more parallel?
  - Polylogarithmic span: span is bounded by a polynomial in $\log n$
- We will look at contraction as a way to build parallel algorithms for some graph problems:
  - Graph Connectivity
  - Spanning Trees

GRAPH CONNECTIVITY

- Two vertices in an undirected graph are connected if there is a path between them.
- A graph is connected if all pairs of vertices are connected.
- The graph connectivity problems partitions a graph into its maximal connected subgraphs.



has two connected subgraphs: $\{a, b, c, d\}$ and $\{e, f\}$

GRAPH CONNECTIVITY

- BFS or DFS
  - Identify vertices of a connected component
  - Identify *all* connected components!
- BFS could be parallel but has span $\propto$ diameter d
- Each connected component needs to be done sequentially!

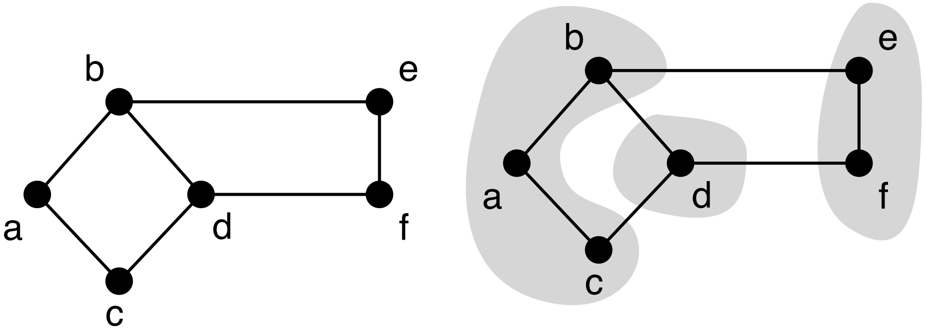
GRAPH CONTRACTION

- Problem → Smaller Problem
- Shrink the size of the graph and solve the connectivity problem on the small graph.
    - Different components can be handled in parallel!
- Applicable to other problems
    - Spanning Trees
    - Minimum Spanning Trees

GRAPH CONTRACTION

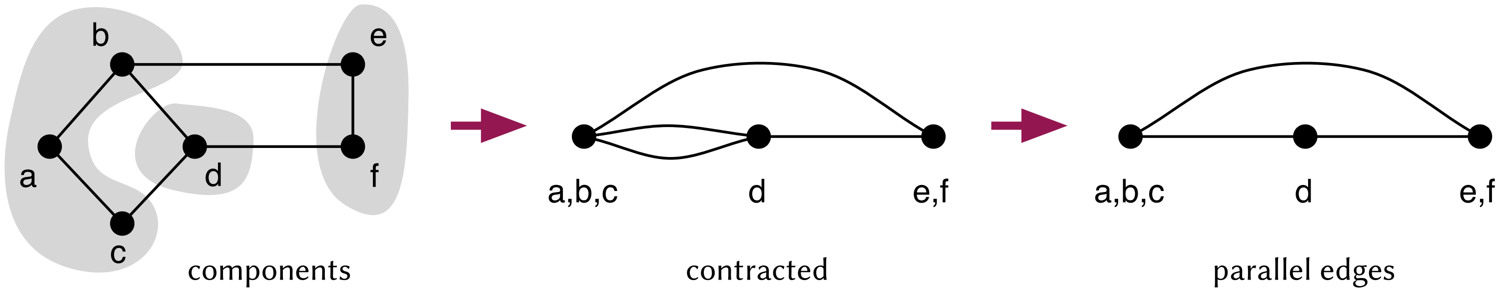
$$\texttt{contract} : \texttt{graph} \rightarrow \texttt{partition}$$

- Takes a graph $G(V, E)$ and returns a partitioning of V into connected subgraphs.
  - Not necessarily maximally connected subgraphs (yet)
  - But vertices in a partition are connected.



$$\{\{a, b, c\}, \{d\}, \{e, f\}\}$$

GRAPH CONTRACTION

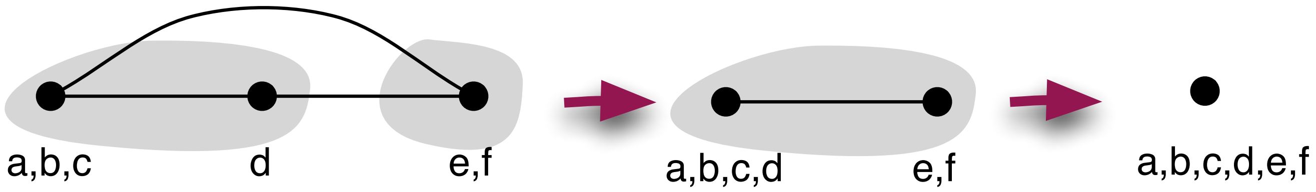


components identified

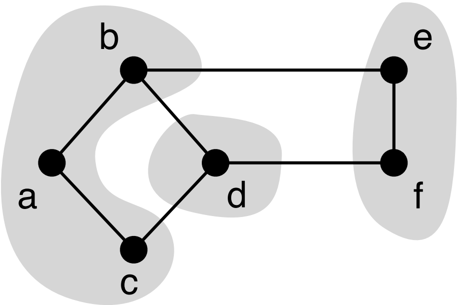
contracted

parallel edges removed

- If the graph contracts on each round, eventually each maximal connected component will shrink down to a single vertex!



REPRESENTING PARTITIONS



$$\downarrow$$

$$\{\{a, b, c\}, \{d\}, \{e, f\}\}$$

$$\downarrow$$

$$(\{a, d, e\}, \{a \mapsto a, b \mapsto a, c \mapsto a, d \mapsto d, e \mapsto e, f \mapsto e\})$$

COUNTING COMPONENTS

```
fun numComponents((V, E), i) =
if |E| = 0 then |V|
else let
    val (V', P) = contract((V, E), i)
    val E' = {(P[u], P[v]) : (u, v) ∈ E | P[u] ≠ P[v]}
in
    numComponents((V', E'), i + 1)
end
```

- Ignore i for the time being!
- V' is the set of representative vertices
- P maps every $v \in V$ to a $v' \in V'$.
- E' is the set of edges in the contracted graph.
  - Self-loops are removed!

COMPUTING COMPONENTS

```
fun $components((V, E),\ i) =$
if $|E| = 0$ then $\{v \mapsto v : v \in V\}$
else let
    val $(V', P) = contract((V, E),\ i)$
    val $E' = \{(P[u], P[v]) : (u, v) \in E \mid P[u] \neq P[v]\}$
    val $P' = components((V', E'),\ i + 1)$
in
    $\{v \mapsto P'[P[v]] : v \in V\}$
end
```

COMPUTING COMPONENTS

```
fun $\mathtt{components}((V, E),\ i) =$
if $|E| = 0$ then $\{v \mapsto v : v \in V\}$
else let
    val $(V', P) = \mathtt{contract}((V, E),\ i)$
    val $E' = \{(P[u], P[v]) : (u, v) \in E \mid P[u] \neq P[v]\}$
    val $P' = \mathtt{components}((V', E'),\ i + 1)$
in
    $\{v \mapsto P'[P[v]] : v \in V\}$
end
```

- Base case: Every vertex maps to itself!

COMPUTING COMPONENTS

```
fun  $components((V, E),\ i) =$
if  $|E| = 0$  then  $\{v \mapsto v : v \in V\}$
else  let
    val  $(V', P) = contract((V, E),\ i)$
    val  $E' = \{(P[u], P[v]) : (u, v) \in E \mid P[u] \neq P[v]\}$
    val  $P' = components((V', E'),\ i + 1)$
in
    $\{v \mapsto P'[P[v]] : v \in V\}$
end
```

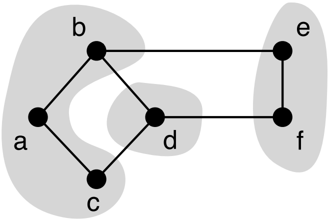
- (Recursively) find components of the contracted graph

COMPUTING COMPONENTS

```
fun  components((V, E),  i) =
if  |E| = 0  then  {v ↦ v : v ∈ V}
else  let
    val  (V′, P) = contract((V, E),  i)
    val  E′ = {(P[u], P[v]) : (u, v) ∈ E | P[u] ≠ P[v]}
    val  P′ = components((V′, E′),  i + 1)
in
    {v ↦ P′[P[v]] : v ∈ V}
end
```

- Map each vertex to the representative vertex of its partition!

COMPUTING COMPONENTS



```
1 fun $\mathtt{components}((V, E),\ i) =$
2 if $|E| = 0$ then $\{v \mapsto v : v \in V\}$
3 else let
4     val $(V', P) = \mathtt{contract}((V, E),\ i)$
5     val $E' = \{(P[u], P[v]) : (u, v) \in E \mid P[u] \neq P[v]\}$
6     val $P' = \mathtt{components}((V', E'),\ i + 1)$
7 in
8     $\{v \mapsto P'[P[v]] : v \in V\}$
9 end
```

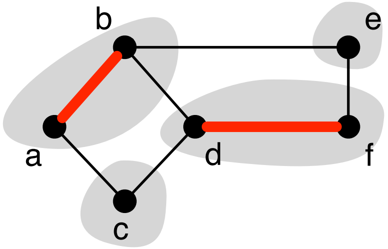
- After 4: $V' = \{a, d, e\}$
  $P = \{a \mapsto a, b \mapsto a, c \mapsto a, d \mapsto d,\ \ e \mapsto e, f \mapsto e\}$
- After 6: $P' = \{a \mapsto a, d \mapsto a, e \mapsto a\}$
- 8 returns: $\{a \mapsto a, b \mapsto a, c \mapsto a, d \mapsto a, e \mapsto a, f \mapsto a\}$

IMPLEMENTING CONTRACT

- Edge Contraction:Only pairs of vertices connected by an edge are contracted.
- Star Contraction: Vertices around a “center star” collapse to the “star”
- Tree Contraction: disjoint trees within the graph are identified and vetices in a tree are collapsed to the root.
- Parallel
- Reduce graph size (vertices/edges?) by a constant factor every round.
    - Will lead to $O(\log n)$ rounds!.

EDGE CONTRACTION

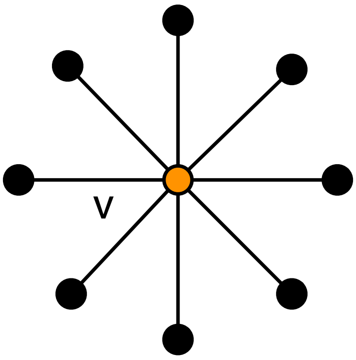
- Find disjoint edges – edges can not share vertices.



- Vertex matching problem
- Can be done in parallel
    - Each edge picks a random priority in $[0, 1]$
    - Any edge which has highest priority for both vertices gets selected.
- It turns out this has some problems!

EDGE CONTRACTION

- Consider a graph like



- How many edges can be contracted each round?
- How many rounds are needed to contract to 1 node?
- Not very parallel!

STAR CONTRACTION

- Star subgraphs can be contracted in parallel!

- How do we find disjoint stars?

FINDING DISJOINT STARS

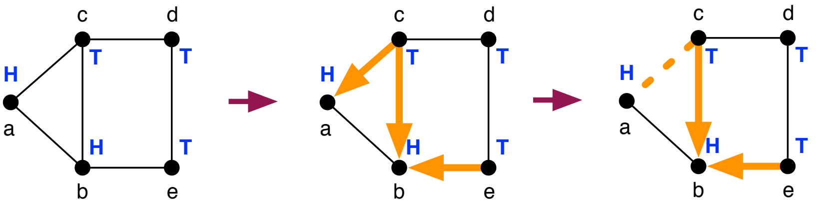
- Each vertex throws a coin
  - Heads → vertex is a star-center
  - Tails → vertex is a *potential satellite* (Why *potential*?)
- Each satellite then selects a center.

RANDOM COIN TOSSES

- Pretend each vertex has a potentially infinite sequence of random coin flips
- $\mathtt{heads}(v, i) : \mathtt{vertex} \times \mathtt{int} \rightarrow \mathtt{bool}$ provides access to these coin tosses.
- This can be implemented with a pseudorandom number generator.

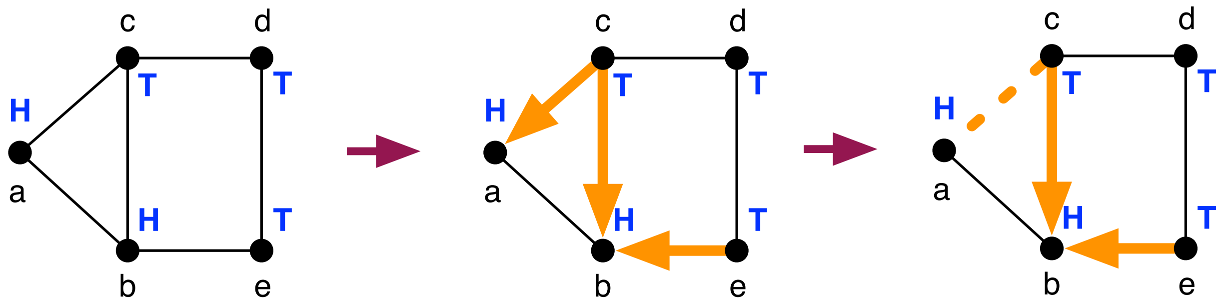
STAR CONTRACTION

```
fun $\texttt{starContract}(G = (V, E), i) =$
let
    % select edges that go from a tail to a head
    val $TH = \{(u, v) \in E \mid \neg\texttt{heads}(u, i) \wedge \texttt{heads}(v, i)\}$
    % make mapping from tails to heads, removing duplicates
    val $P = \cup_{(u,v) \in TH} \{u \mapsto v\}$
    % remove vertices that have been remapped
    val $V' = V \setminus \texttt{domain}(P)$
    % Map remaining vertices to themselves
    val $P' = \{u \mapsto u : u \in V'\} \cup P$
in $(V', P')$ end
```



coin flips (heads(v,i)) find potential centers (TH) compute "hook" edges (P)

STAR CONTRACTION

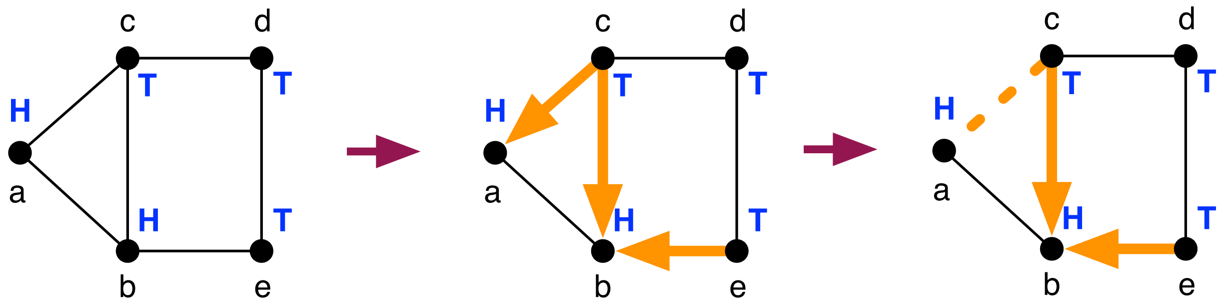


coin flips (heads(v,i)) find potential centers (TH) compute "hook" edges (P)

$$\textbf{val } TH = \{(u, v) \in E \mid \neg \mathtt{heads}(u, i) \wedge \mathtt{heads}(v, i)\}$$

- $TH = \{(c, a), (c, b), (e, b)\}$.

STAR CONTRACTION

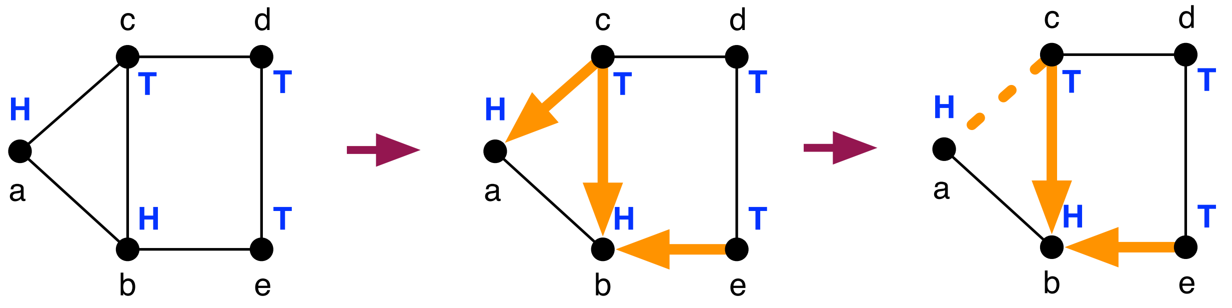


coin flips (heads(v,i)) find potential centers (TH) compute "hook" edges (P)

$$\textbf{val } P = \cup_{(u,v)\in TH} \{u \mapsto v\}$$

- $TH = \{(c, a), (c, b), (e, b)\}$
- $P = \{c \mapsto b, e \mapsto b\}$

STAR CONTRACTION

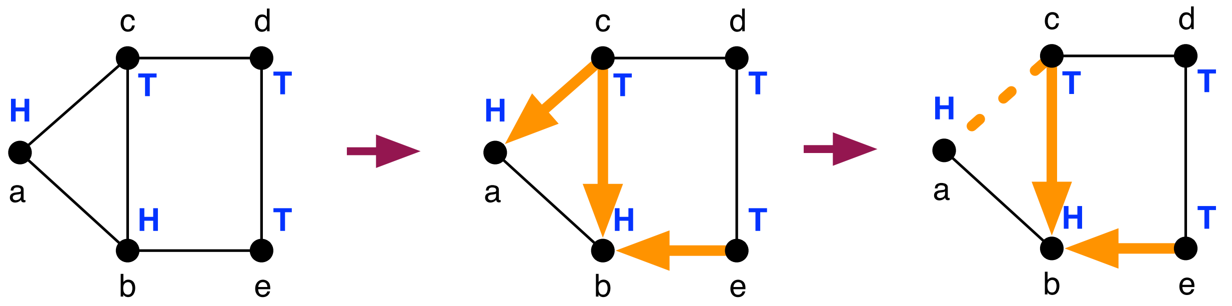


coin flips (heads(v,i)) find potential centers (TH) compute "hook" edges (P)

$$\textbf{val } V' = V \setminus \mathtt{domain}(P)$$

- $P = \{c \mapsto b, e \mapsto b\}$
- $\mathtt{domain}(P) = \{c, e\}$
- $V' = \{a, b, d\}$

STAR CONTRACTION



coin flips (heads(v,i)) find potential centers (TH) compute "hook" edges (P)

val $P' = \{u \mapsto u : u \in V'\} \cup P$

- $P = \{c \mapsto b, e \mapsto b\}$, $V' = \{a, b, d\}$
- $P' = \{a \mapsto a, b \mapsto b, c \mapsto b, d \mapsto d, e \mapsto b\}$

ANALYSIS OF STAR CONTRACTION

LEMMA

For a graph G with n non-isolated vertices, let X_n be the random variable indicating the number of vertices removed by `starContract`$(G, \_)$. Then, $\mathbf{E}\left[X_n\right] \geq n/4$.

- H_v: vertex v comes up heads, T_v: vertex v comes up tails
- R_v: vertex v is removed in contraction
- v has at least one neighbor u.
- $T_v \wedge H_u$ implies R_v
  - If v is a tail, join $u's$ star or some other star.
- $\mathbf{Pr}[R_v] \geq \mathbf{Pr}[T_v]\mathbf{Pr}[H_u] = 1/4$
- Expected total $\geq n/4$

ANALYSIS OF STAR CONTRACTION

```
fun $\mathtt{starContract}(G = (V, E), i) =$
let
    % select edges that go from a tail to a head – $O(m)$ work, $O(1)$ span
    val $TH = \{(u, v) \in E \mid \neg \mathtt{heads}(u, i) \wedge \mathtt{heads}(v, i)\}$
    % make mapping from tails to heads, removing duplicates
    % $O(n)$ work, $O(\log n)$ span
    val $P = \cup_{(u,v) \in TH} \{u \mapsto v\}$
    % remove vertices that have been remapped
    % $O(n)$ work, $O(\log n)$ span
    val $V' = V \setminus \mathtt{domain}(P)$
    % Map remaining vertices to themselves -$O(n)$ work, $O(\log n$ span
    val $P' = \{u \mapsto u : u \in V'\} \cup P$
in $(V', P')$ end
```

- n nodes, m edges
- $O(n + m)$ work, $O(\log n)$ span.

ANALYSIS OF CONNECTIVITY

```
fun numComponents((V, E), i) =
if |E| = 0 then |V|
else let
    val (V', P) = starContract((V, E), i)
    val E' = {(P[u], P[v]) : (u, v) ∈ E | P[u] ≠ P[v]}
in
    numComponents((V', E'), i + 1)
end
```

- $S(n) = S(n') + O(\log n)$
- $n' = n - X_n$ and $\mathbf{E}[X_n] = n/4$, so $\mathbf{E}[n'] = 3n/4$
- $S(n) \in O(\log^2 n)$

ANALYSIS OF CONNECTIVITY

- We can remove a constant fraction of vertices every round.
- For each vertex removed, we remove at least one edge.
- Consider a hypothetical contraction

round	vertices	edges
1	n	m
2	$n/2$	$m - n/2$
3	$n/4$	$m - 3n/4$
4	$n/8$	$m - 7n/8$

- Number of edges does not go below $m - n$.

ANALYSIS OF CONNECTIVITY

$$W(n, m) \leq W(n', m) + O(n + m),$$

- As before, $\mathbf{E}[n'] = 3n/4$, so $\mathbf{E}[W(n, m)] \in O(n + m \log n)$

TREE CONTRACTION

- Identify disjoint trees and contract them.
- For every tree of t vertices contracted, $t - 1$ edges are removed.
- Number of edges also go down geometrically at every round.
- Leads to $O(m)$ work and $O(\log^2 n)$ span.

15-210
PARALLEL AND SEQUENTIAL ALGORITHMS AND DATA STRUCTURES

LECTURE 17

NO LECTURE

SYNOPSIS

- There is no lecture 17

15-210
PARALLEL AND SEQUENTIAL ALGORITHMS AND DATA STRUCTURES

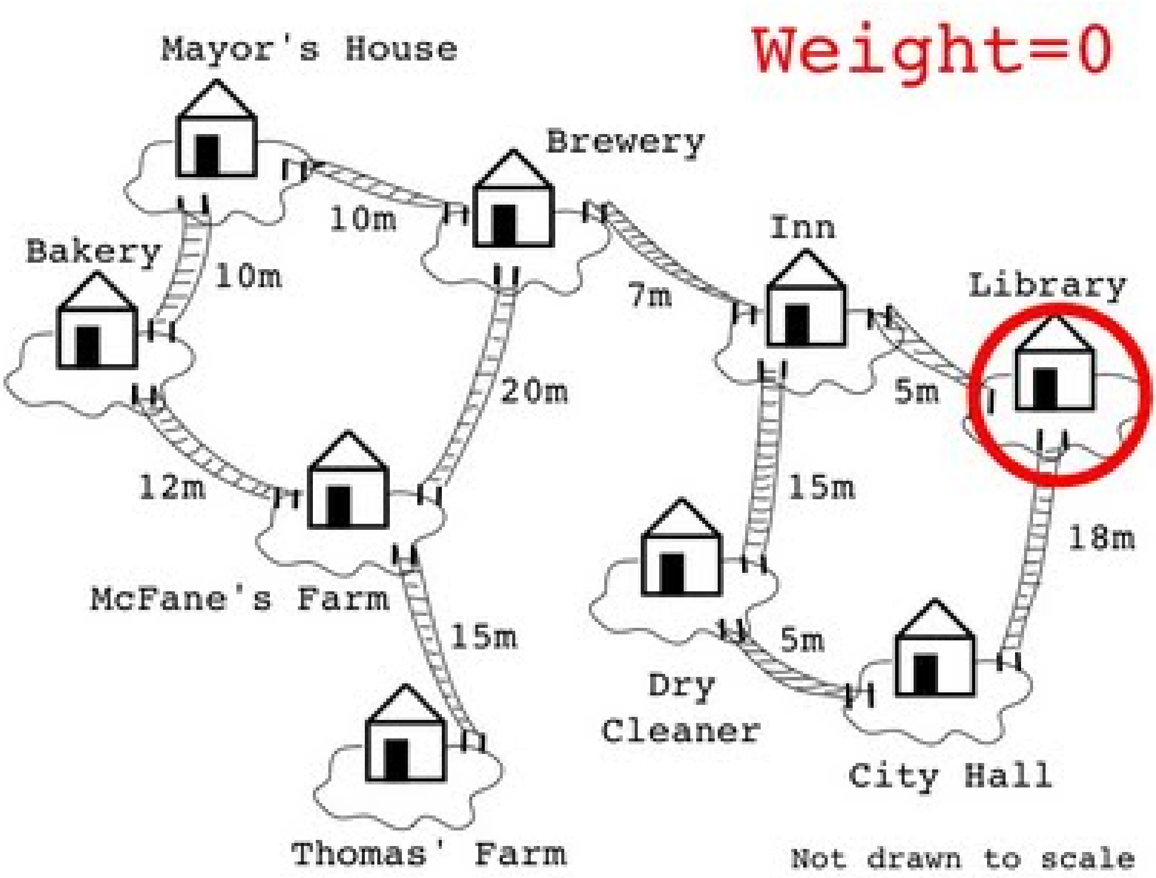
LECTURE 18

MINIMUM SPANNING TREES

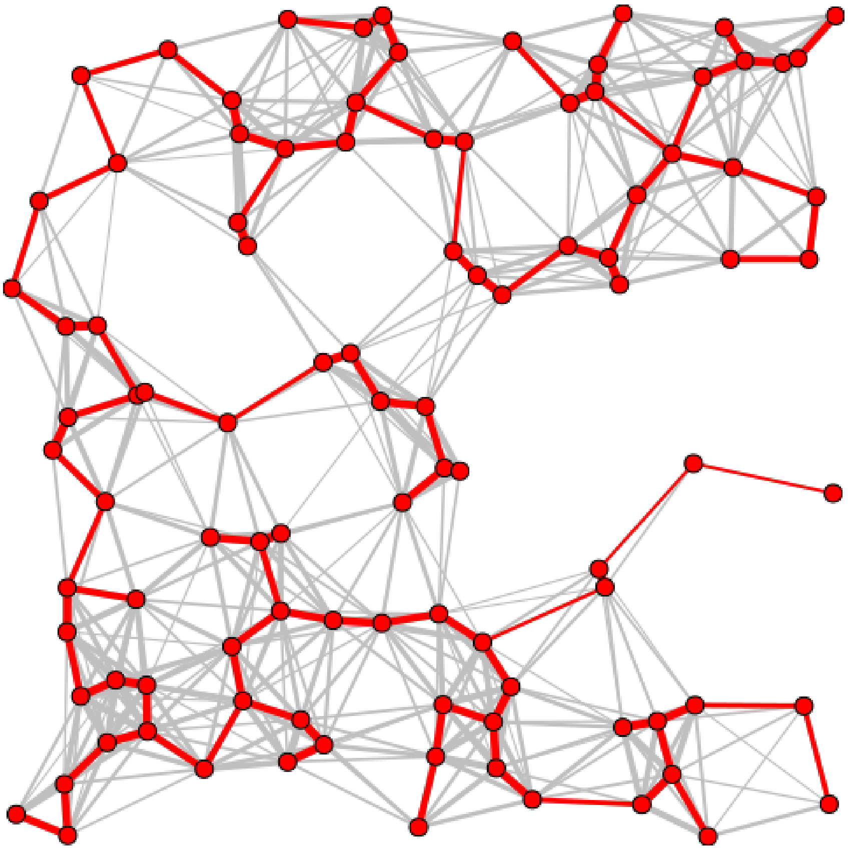
SYNOPSIS

- Minimum Spanning Trees
- Kruskal's and Prim's Algorithms
- Using Star Contraction for MST

MINIMUM SPANNING TREES



MINIMUM SPANNING TREES



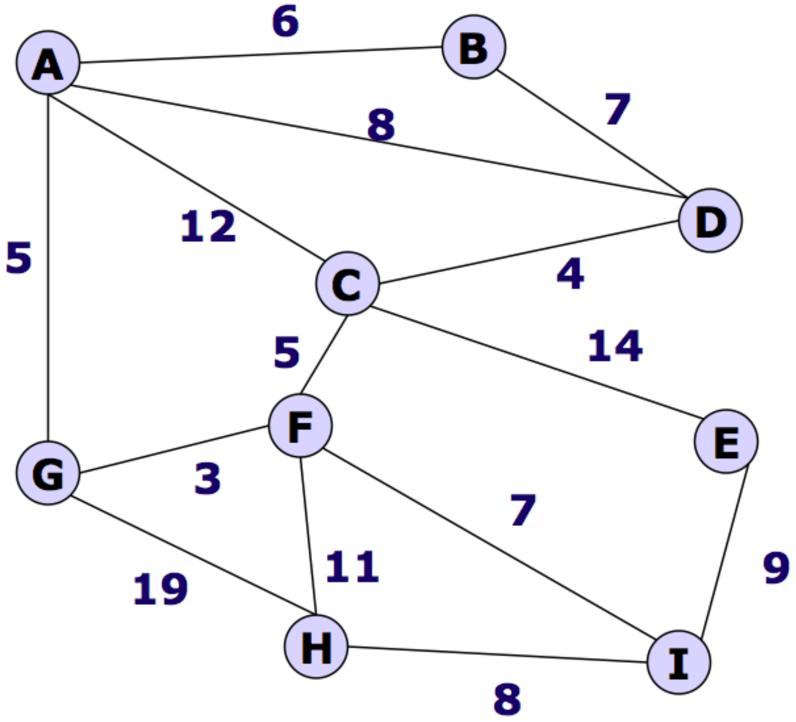
MINIMUM SPANNING TREES

- Given a connected undirected graph $G = (V, E)$
  - Each edge e has $w_e \geq 0$
- Find a spanning tree, T that minimizes

$$w(T) = \sum_{e \in E(T)} w_e.$$

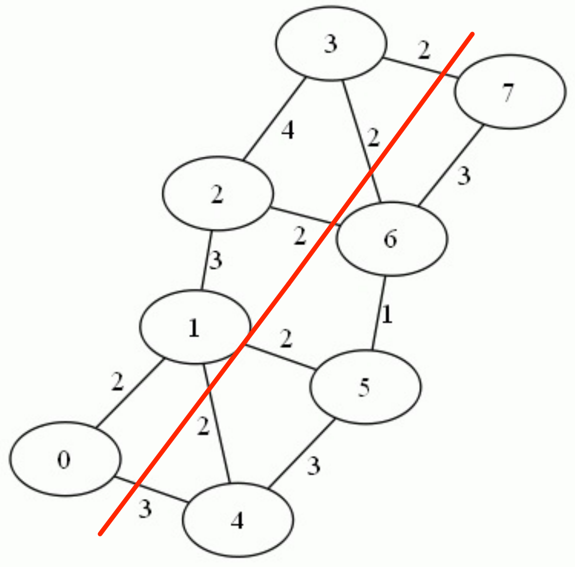
- Sequential algorithms:
  - Kruskal's Algorithm
  - Prim's Algorithm

MINIMUM SPANNING TREES



LIGHT EDGE RULE

- Given $G = (V, E)$, $U \subsetneq V$ partitions the graph into two parts with vertices U and $V \setminus U$.
- The edges between U and $V \setminus U$ are called the cut edges $E(U, \overline{U})$.

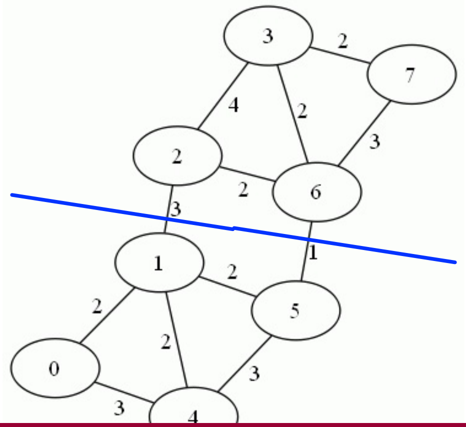


LIGHT EDGE RULE

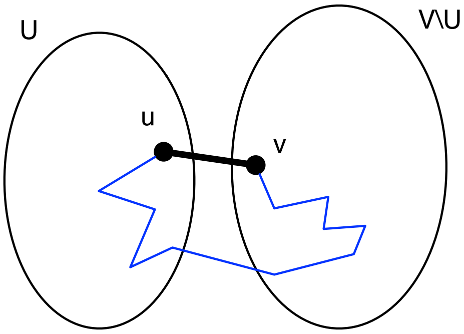
THEOREM

Let $G = (V, E, w)$ be a connected undirected weighted graph with distinct edge weights.

- For any nonempty $U \subsetneq V$
- the minimum weight edge e between U and $V \setminus U$ is in the minimum spanning tree MST(G) of G.



LIGHT EDGE RULE

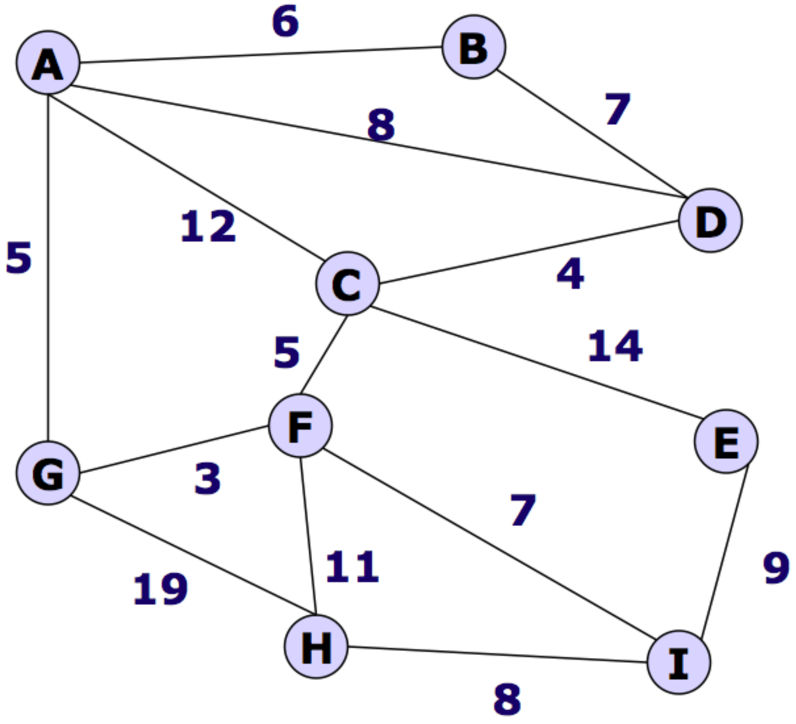


- Assume $e = (u, v)$ is the minimum edge in the cut but not in the MST.
- MST should have at least another edge in the cut.
- Adding e to the path between u and v creates a cycle.
- Removing the max edge from path (blue line) and adding e should give a ST with less weight.
- Original (claimed) MST (through blue line) can not be a

KRUSKAL'S ALGORITHM

- Greedy
- Each vertex is a subtree by itself initially
- Combine the two sub-trees on both sides of the next smallest edge (if they are different)
- Uses the union-find data structure.
- $O(m \log n)$ work and span!

KRUSKAL'S ALGORITHM



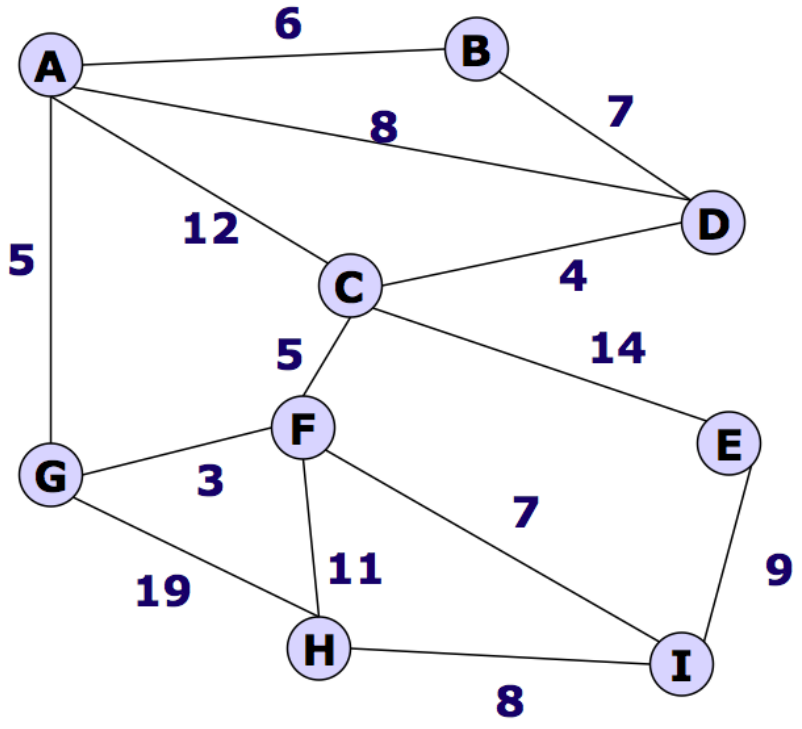
PRIM'S ALGORITHM

- Greedy
- Based on Priority-based Search – Variant of Dijsktra's Algorithm
- Maintain visited X and frontier F vertices.
- Visit the nearest unvisited vertex in the frontier.
- $O(m \log n)$ work and span!

Prim’s Algorithm

```
fun prim($G$) =
let
   fun enqueue $v$ $(Q, (u, w))$ = PQ.insert $(w, (v, u))$ $Q$
   fun proper($X$, $Q$, $T$) =
      case PQ.deleteMin($Q$) of
           (NONE, _) ⇒ $T$
         | (SOME($d, (u, v)$), $Q'$) ⇒
           if $(v \in^{?} X)$ then proper($X$, $Q'$, $T$)
           else let
              val $X' = X \cup \{v\}$
              val $T' = T \cup \{(u, v)\}$
              val $Q'' =$ iter (enqueue $v$) $Q'$ $N_G(v)$
           in proper($X'$, $Q''$, $T'$) end

   val $s$ = an arbitrary vertex from $G$
   val $Q$ = iter (enqueue $s$) {} $N_G(s)$
in
   proper({$s$}, $Q$, {})
end
```

PRIM'S ALGORITHM



PARALLEL MST ALGORITHMS

OBSERVATION

- The minimum weight edge out of every vertex of a weighted graph *G* belongs to its MST.

- Why should this be the case?

- MST can contain other edges!

PARALLEL MST - IDEA #1

- Throw all minimum weight edges into MST
- Tree contract the vertices for all these edges
- Repeat until no edges remain!

- Each rounds removes at least 1/2 of the vertices (Why?)

PARALLEL MST - IDEA #2

- Let *minE* be the set of minimum weight edges.
- Let $H = (V, minE)$ be a subgraph of G
- We apply (modified) star contraction to H
  - The tails hook up through the minimum weight edge!

```
fun minStarContract($G = (V, E), i$) =
let
  val $minE$ = minEdges($G$)
  val $P = \{u \mapsto (v, w) \in minE \mid \neg heads(u, i) \wedge heads(v, i)\}$
  val $V' = V \setminus domain(P)$
in $(V', P)$ end
```

PARALLEL MST - IDEA #2

- Even though we are working with a subgraph, the star contract lemma still applies.

LEMMA

For a graph G with n non-isolated vertices, let X_n be the random variable indicating the number of vertices removed by `minStarContract`(G, r). Then, $\mathbf{E}(X_n) \geq n/4$.

- MST will take expected $O(\log n)$ rounds.

BOOKKEEPING

- As the graph contracts, the end point of each edge changes!
- At the end, the edges may not have the original end points.
- Associate a unique label to each edge initially:
    - ($\mathtt{vertex} \times \mathtt{vertex} \times \mathtt{weight} \times \mathtt{label}$)
    - The end points change but the label does not!

Modified Star Contract

```
1  fun minStarContract($G = (V, E), i$) =
2  let
3     val $minE = \texttt{minEdges}(G)$
4     val $P = \{(u \mapsto (v, w, \ell)) \in minE \mid \neg\texttt{heads}(u, i) \wedge \texttt{heads}(v, i)\}$
5     val $V' = V \setminus \texttt{domain}(P)$
6  in $(V', P)$ end
```

- Line 3: Finds min edge for each vertex.
  - All these belong to the MST
- Line 4: Picks tails and heads, and the creates mapping from tails to heads.
- Line 5: Removes all tail vertices from the vertex set.

THE MST ALGORITHM

```
**fun** $\mathtt{MST}((V, E),\ T,\ i) =$
**if** $|E| = 0$ **then** $T$
**else let**
    **val** $(V', PT) = \mathtt{minStarContract}((V, E), i)$
    **val** $P = \{u \mapsto v : u \mapsto (v, w, \ell) \in PT\} \cup \{v \mapsto v : v \in V'\}$
    **val** $T' = \{\ell : u \mapsto (v, w, \ell) \in PT\}$
    **val** $E' = \{(P[u], P[v], w, l) : (u, v, w, l) \in E \mid P[u] \neq P[v]\}$
**in**
    $\mathtt{MST}((V', E'),\ T \cup T',\ i + 1)$
**end**
```

- Invoked by $\mathtt{MST}(G, \{\}, 1)$.

IMPLEMENTING minEdges (G)

```
fun joinEdges((v_1, w_1, l_1), (v_2, w_2, l_2)) =
    if (w_1 ≤ w_2) then (v_1, w_1, l_1) else (v_2, w_2, l_2)

fun minEdges(E) =
let
    val ET = {u ↦ (v, w, l) : (u, v, w, l) ∈ E}
in
    (merge joinEdges)  {}  ET
end
```

15-210
PARALLEL AND SEQUENTIAL ALGORITHMS AND DATA STRUCTURES

LECTURE 19

QUICKSORT ANALYSIS AND SORTING LOWER BOUNDS

SYNOPSIS

- Quicksort
- Work and Span Analysis of Randomized Quicksort
- Lower Bound for Comparison-based Sorting
- Lower Bound for Merging

QUICKSORT

- Originally invented and analyzed by Hoare in 1960's.
- I strongly urge to watch Jon Bentley on "Three beautiful Quicksorts" at
  - `www.youtube.com/watch?v=QvgYAQzg1z8`.

SEQUENTIAL QUICKSORT

```
int i, j;
for( i = low, j = high - 1; ; )
{
    while( a[ ++i ] < pivot );
    while( pivot < a[ --j ] );
    if( i >= j )
    break;
    swap( a, i, j );
}
// Restore pivot
swap( a, i, high - 1 );
quicksort( a, low, i - 1 ); // Sort small elements
quicksort( a, i + 1, high ); // Sort large elements
```

QUICKSORT

```
fun $quicksort(S) =$
  if $|S| = 0$ then $S$
  else let
      val $p = \textit{pick a pivot from } S$
      val $S_1 = \langle s \in S \mid s < p \rangle$
      val $S_2 = \langle s \in S \mid s = p \rangle$
      val $S_3 = \langle s \in S \mid s > p \rangle$
      val $(R_1, R_3) = (quicksort(S_1) \parallel quicksort(S_3)$
    in
      $append(R_1, append(S_2, R_3))$
    end
```

QUICKSORT

- Each call to Quicksort either makes
  - No recursive calls (base case), or
  - Two recursive calls
- Call tree is a binary
- Depth the call tree determines the span of the algorithm.

PICKING THE PIVOT

- Always pick the first element
  - Worst case $O(n^2)$ work.
  - In practice, almost sorted inputs are not uncommon.
- Pick the median of 3 elements (e.g., first, middle and last elements)
  - could possible divide evenly
  - worst case is still bad
- Pick an element at random
  - we hope this divides evenly in expectation
  - leading to expected $O(n \log n)$ work and $O(\log^2 n)$ span.

PICKING THE PIVOT

- Pick first element
  - Worst case $O(n^2)$ work.
  - Expected $O(n \log n)$ work
    - Averaged over all possible orderings.
  - Work well on the average
  - Slow on some, possibly common, cases.
- Pick a random element
  - Expected worst-case $O(n \log n)$ work.
    - For input in **any** order, the expected work is $O(n \log n)$
  - No input has expected $O(n^2)$ work.
  - With a small probability, we could be unlucky and have $O(n^2)$ work.

RANDOMIZED QUICKSORT

- Assign a uniformly random priority to each number in $[0, 1]$.

```
fun quicksort($S$) =
  if $|S| = 0$ then $S$
  else let
      val $p$ = pick as pivot the highest priority element from $S$
      val $S_1 = \langle s \in S \mid s < p \rangle$
      val $S_2 = \langle s \in S \mid s = p \rangle$
      val $S_3 = \langle s \in S \mid s > p \rangle$
      val $(R_1, R_3) = (\mathtt{quicksort}(S_1) \parallel \mathtt{quicksort}(S_3))$
    in
      $\mathit{append}(R_1, \mathit{append}(S_2, R_3))$
    end
```

- Once the priorities are assigned, the algorithm is deterministic.

RANDOMIZED QUICKSORT

- Count comparisons made!
  - Almost all the work is comparisons.

$$X_n = \# \text{ of comparisons } \texttt{quicksort} \text{ makes on input of size } n$$

- Find $\mathbf{E}[X_n]$ for any input sequence S
- Notation:
  - Let $T = \texttt{sort}(S)$
  - T_i and T_j refer to elements in the final sorted order and $i < j$ and $T_i \leq T_j$.
  - p_i refers to priority chosen for T_i.
  - $A_{i,j} = 1$ if T_i and T_j were ever compared during the sort.

ANALYZING QUICKSORT

- Crucial point is how to model $A_{i,j}$.
- In any one call to `quicksort`, there are three cases:
    - Pivot p is either T_i or $T_j \Rightarrow A_{i,j} = 1$
    - $T_i < p < T_j \Rightarrow T_i \in S_1, T_j \in S_3, A_{i,j} = 0$
    - Either $p < T_i$ or $p > T_j \Rightarrow T_i, T_j \in S_1$ or $T_i, T_j \in S_3$
- If two elements are compared in a `quicksort` call, they will never be compared again in any other call!

ANALYZING QUICKSORT

$$X_n \quad \leq \quad 3\sum_{i=1}^{n}\sum_{j=i+1}^{n} A_{ij}$$

- The non-optimized code compares each element to pivot 3 times.

```
...
        val $S_1 = \langle s \in S \mid s < p \rangle$
        val $S_2 = \langle s \in S \mid s = p \rangle$
        val $S_3 = \langle s \in S \mid s > p \rangle$
...
```

- By linearity of expectation

$$\mathbf{E}\left[X_n\right] \leq 3\sum_{i=1}^{n}\sum_{j=i+1}^{n} \mathbf{E}\left[A_{ij}\right] = 3\sum_{i=1}^{n}\sum_{j=i+1}^{n} \mathbf{Pr}[A_{ij} = 1]$$

ANALYZING QUICKSORT

- Consider first when the pivot is one of T_i, T_{i+1}, ..., T_j
- T_i and T_j are compared $\Leftrightarrow$ p_i or p_j is the highest priority among $\{p_i, p_{i+1}, \ldots, p_j\}$.
  - Assume $T_k, i < k < j$ has higher priority.
  - For any subdivision $\cdots$, $T_i, \cdots$, $T_k, \cdots$, T_j will become a pivot and separate T_i and T_j
  - T_i and T_j will never be compared!

ANALYZING QUICKSORT

$$\begin{aligned}\mathbf{E}[A_{ij}] &= \mathbf{Pr}[A_{ij} = 1] \\ &= \mathbf{Pr}[p_i \text{ or } p_j \text{ is the maximum in } \{p_i, \ldots, p_j\}] \\ &= \frac{2}{j - i + 1} \text{ (Why ?)}\end{aligned}$$

- $j - i + 1$ elements between p_i and p_j and each is equally likely to be the maximum.
- We want either p_i or p_j, hence $\frac{2}{j-i+1}$
- T_i is compared to T_{i+1} with probability 1.

ANALYZING QUICKSORT

$$
\begin{aligned}
\mathbf{E}[X_n] &\le 3 \sum_{i=1}^{n} \sum_{j=i+1}^{n} \mathbf{E}[A_{ij}] \\
&= 3 \sum_{i=1}^{n} \sum_{j=i+1}^{n} \frac{2}{j-i+1} \\
&= 3 \sum_{i=1}^{n} \sum_{k=2}^{n-i+1} \frac{2}{k} \qquad \text{(change variables)} \\
&\le 6 \sum_{i=1}^{n} H_n \\
&\le 6 \cdot n \cdot H_n \in O(n \log n)
\end{aligned}
$$

ANALYZING QUICKSORT

- Indirectly, average work for basic deterministic quicksort is $O(n \log n)$.
  - Just shuffle data randomly and apply the basic algorithm
  - $\equiv$ to picking random priorities

ALTERNATIVE ANALYSIS

- Write a recurrence for the number of comparisons:

$$X(n) = X(Y_n) + X(n - Y_n - 1) + n - 1$$

- Random variable Y_n is the size of S_1.

$$\begin{aligned}
\mathbf{E}\left[X(n)\right] &= \mathbf{E}\left[X(Y_n) + X(n - Y_n - 1) + n - 1\right] \\
&= \mathbf{E}\left[X(Y_n)\right] + \mathbf{E}\left[X(n - Y_n - 1)\right] + n - 1 \\
&= \frac{1}{n}\sum_{i=0}^{n-1}(\mathbf{E}\left[X(i)\right] + \mathbf{E}\left[X(n - i - 1)\right]) + n - 1
\end{aligned}$$

ALTERNATIVE ANALYSIS

$$\begin{aligned} \mathbf{E}[X(n)] &= \frac{1}{n}\sum_{i=0}^{n-1}(\mathbf{E}[X(i)] + \mathbf{E}[X(n-i-1)]) + n - 1 \\ &= \frac{2}{n}\sum_{i=0}^{n-1}\mathbf{E}[X(i)] + n - 1 \end{aligned}$$

- With telescoping, this also solves as $O(n\log n)$

EXPECTED SPAN

- S is split into $L(ess)$, $E(qual)$ and $(g)R(eater)$.
- Let $X_n = \max\{|L|, |R|\}$,
- We use `filter` to partition.

$$S(n) = S(X_n) + O(\log n)$$

EXPECTED SPAN

- Let $\overline{S}(n)$ denote $\mathbf{E}\left[S(n)\right]$
- We bound $\overline{S}(n)$ by considering $\mathbf{Pr}[X_n \leq 3n/4]$ and $\mathbf{Pr}[X_n > 3n/4]$.
- $\mathbf{Pr}[X_n \leq 3n/4] = 1/2$
  - As with `SmallestK`, $1/2$ of the randomly chosen pivots results in larger partition of at most size $3n/4$ elements.

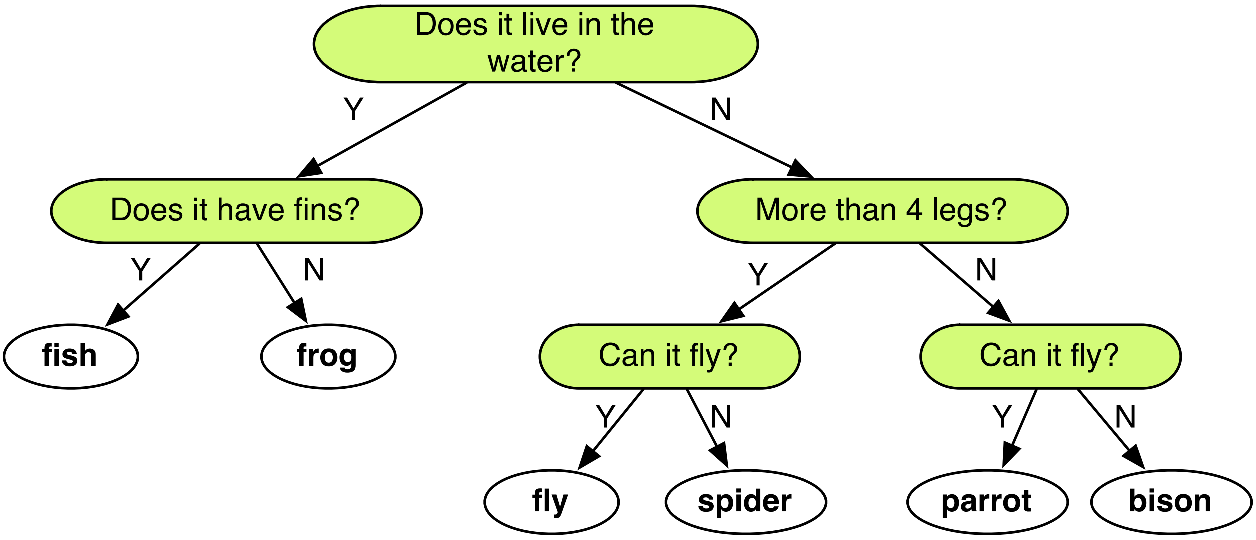
EXPECTED SPAN

$$
\begin{aligned}
\overline{S}(n) &= \sum_i \mathbf{Pr}[X_n = i] \cdot \overline{S}(i) + c \log n \\
&\leq \mathbf{Pr}[X_n \leq \tfrac{3n}{4}]\overline{S}(\tfrac{3n}{4}) + \mathbf{Pr}[X_n > \tfrac{3n}{4}]\overline{S}(n) + c \cdot \log n \\
&\leq \tfrac{1}{2}\overline{S}(\tfrac{3n}{4}) + \tfrac{1}{2}\overline{S}(n) + c \cdot \log n \\
&\Longrightarrow \ (1 - \tfrac{1}{2})\overline{S}(n) \leq \tfrac{1}{2}\overline{S}(\tfrac{3n}{4}) + c \log n \\
&\Longrightarrow \ \overline{S}(n) \leq \overline{S}(\tfrac{3n}{4}) + 2c \log n \\
&\Longrightarrow \ \overline{S}(n) \in O(\log^2 n)
\end{aligned}
$$

LOWER BOUND FOR SORTING

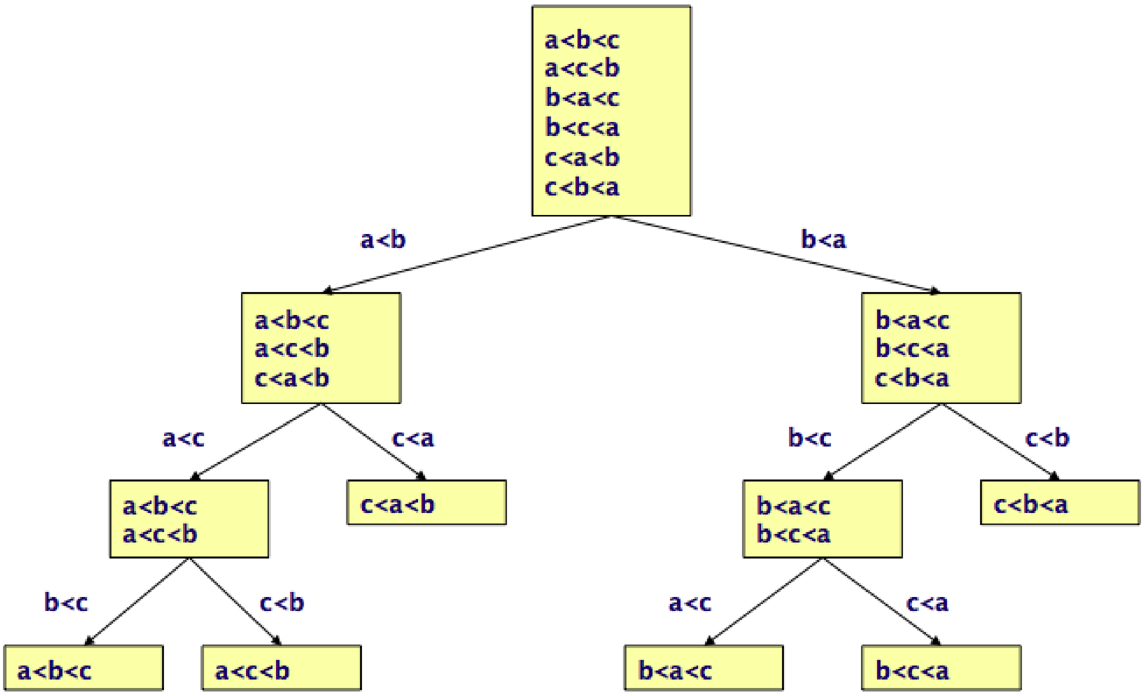
- What is asymptotically the minimum number comparisons any sorting algorithm has to make?
- Lower-bounds apply to problems not to algorithms.
  - Algorithms provide upper bounds!
- We say sorting is $\Omega(n \log n)$
- No (comparison-based) sorting algorithm has work asymptotically lower than $n \log n$.

DECISION TREES



- If there are N outcomes, the number of questions is at least $\log_2 N$.

SORTING AS A DECISION PROBLEM



- For n items, how many possible outcomes can there be?
    - $n!$ ⇒ we need at least $\log_2(n!)$ “questions”.

SORTING AS A DECISION PROBLEM

$$
\begin{aligned}
\log(n!) &= \log n + \log(n-1) + \cdots + \log(n/2) + \cdots + \log 1 \\
&\geq \log n + \log(n-1) + \cdots + \log(n/2) \\
&\geq \frac{n}{2} \cdot \log(n/2) \in \Omega(n \log n)
\end{aligned}
$$

LOWER BOUND FOR MERGING

- We have sorted sequences $A, |A| = n$ and $B, |B| = m$ and $m \leq n$.
  - Assume all elements are unique.
- All interleavings are possible
- We need to choose m positions out of $n + m$ to place the elements of B amongst elements of A.
- This can be done in $\log_2 \binom{n+m}{m}$ ways.

LOWER BOUND FOR MERGING

- $\binom{n}{r} \geq \left(\frac{n}{r}\right)^r$
  - See Lemma in the notes.

$$\log_2 \binom{n+m}{m} \geq \log_2(\frac{n+m}{m})^m = m \log_2(1 + \frac{n}{m})$$

15-210
PARALLEL AND SEQUENTIAL ALGORITHMS AND DATA STRUCTURES

LECTURE 20

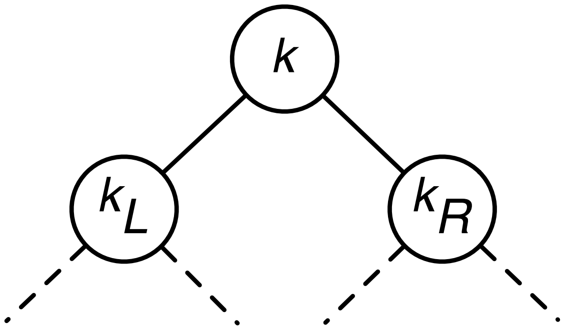
SEARCH TREES I: BSTS SPLIT, JOIN, AND UNION

SYNOPSIS

- Binary Search Trees
- Basic Structural Operations on BSTs
- Basic Operations on BSTs
- Concrete Implementations
- Cost Analysis

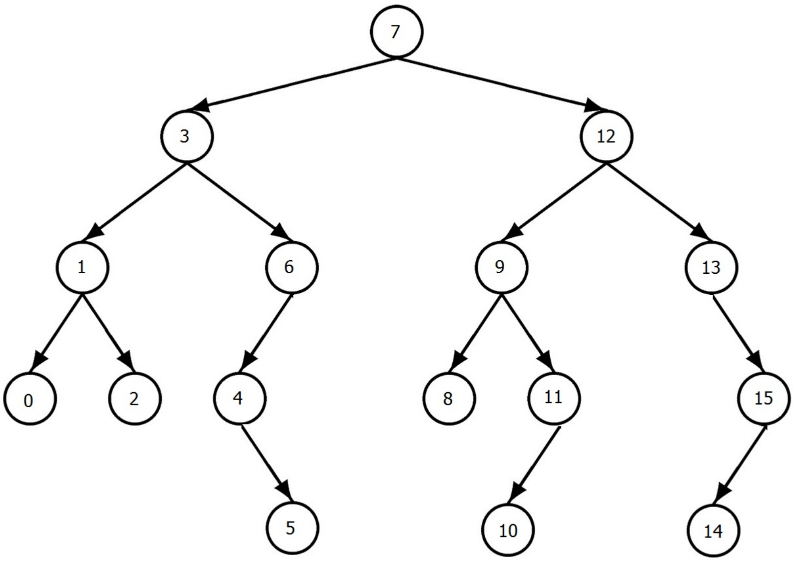
BINARY TREES

- Trees where each node has at most 2 children each of which is a binary tree.
    - Left child / Left subtree
    - Right child / Right subtree



BINARY SEARCH TREES

- Binary trees with the “search” property
- For each node v with key k
  - The key of the left child $k_L < k$
  - The key of the right child $k_R > k$



BALANCED TREES

- We try to keep binary search trees balanced.
    - Both children are about the same height
    - Both subtrees are about the same size
- AVL Trees
    - Left and right subtree heights differ by at most 1.
    - $O(\log n)$ root height maintained after each insertion and deletion.
- Splay Trees
    - Balanced in the amortized sense
    - A sequence of n `find`, `insert`, or `delete` operations take $O(n \log n)$ work.
    - So average is $O(\log n)$ work.

BASIC BST OPERATIONS

- Data type is defined by structural induction
  - Leaf
  - Node with a left child, a right child, a key, optional additional data.

```
datatype BST = Leaf |
               Node of (BST * BST * key * data)
```

BASIC BST OPERATIONS

- $\mathtt{split}(T, k) : \mathtt{BST} \times \mathtt{key} \rightarrow$
  $\mathtt{BST} \times (\mathtt{data\ option}) \times \mathtt{BST}$

- $\mathtt{split}$ divides T into two BSTs,
  - one consisting of all the keys from T less than k
  - the other all the keys greater than k
- If k appears in the tree with associated data d then $\mathtt{split}$ returns $\mathtt{SOME}(d)$
- Otherwise it returns $\mathtt{NONE}$.

BASIC BST OPERATIONS

- $\texttt{join}(L, m, R) : \mathit{BST} \times (\mathit{key} \times \mathit{data})\ \mathit{option} \times \mathit{BST} \rightarrow \mathit{BST}$
- Takes a left subtree (*L*) an optional key-data pair *m* and a right subtree (*R*)
  - Assumes all keys in *L* are less than all keys in *R*.
  - If present, the optional key is also larger than all keys in *L* and smaller than all keys in *R*.
- Creates a new BST that is the union of *L* and *R* and *m*.
- We also assume both split and join maintain balance.

BASIC BST OPERATIONS

- $\texttt{expose}(T) : \texttt{BST} \rightarrow (\texttt{BST} \times \texttt{BST} \times \texttt{key} \times \texttt{data})\ \texttt{option}$
- Returns the components if BST T is not empty.

Basic BST Operations - Search

```
fun search $T$ $k$ =
let val $(\_, v, \_) = split(T, k)$
in $v$
end
```

BASIC BST OPERATIONS - INSERT

```
fun $\mathtt{insert}\ T\ (k, v) =$
let val $(L, v', R) = \mathtt{split}(T, k)$
in $\mathtt{join}(L, \mathtt{SOME}(k, v), R)$
end
```

BASIC BST OPERATIONS - DELETE

```
fun $delete$ $T$ $k$ =
let val $(L, \_, R) = split(T, k)$
in $join(L, NONE, R)$
end
```

CONCRETE IMPLEMENTATIONS: SPLIT

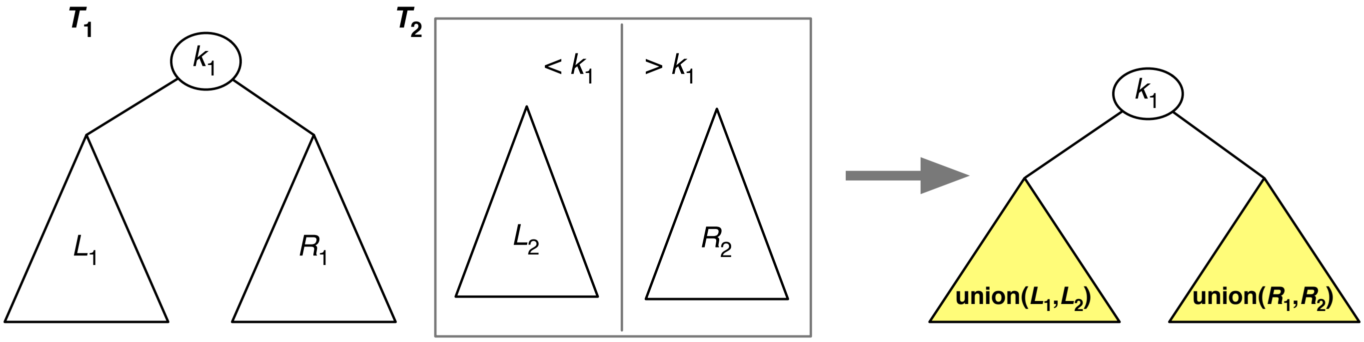
```
datatype BST = Leaf |
               Node of (BST * BST * key * data)
```

```
fun $split(T, k) =$
  case $T$ of
    $Leaf \Rightarrow (Leaf, NONE, Leaf)$
  | $Node(L, R, k', v) \Rightarrow$
      case $compare(k, k')$ of
        $LESS \Rightarrow$
          let val $(L', r, R') = split(L, k)$
          in $(L', r, Node(R', R, k', v))$ end
        $EQUAL \Rightarrow (L, SOME(v), R)$
        $GREATER \Rightarrow$
          let val $(L', r, R') = split(R, k)$
          in $(Node(L, L', k', v), r, R')$ end
```

CONCRETE IMPLEMENTATIONS: JOIN

```
fun  $join(T_1, m, T_2) =$
    case  $m$  of
        $SOME(k, v) \Rightarrow Node(T_1, T_2, k, v)$
      | $NONE \Rightarrow$
          case  $T_1$  of
            $Leaf \Rightarrow T_2$
          | $Node(L, R, k, v) \Rightarrow Node(L, join(R, NONE, T_2), k, v))$
```

CONCRETE IMPLEMENTATIONS: UNION



- For T_1 with key k_1 and children L_1 and R_1 at the root, use k_1 to split T_2 into L_2 and R_2.
- Recursively find $L_u = \texttt{union}(L_1, L_2)$ and $R_u = \texttt{union}(R_1, R_2)$.
- Now $\texttt{join}(L_u, k_1, R_u)$.

CONCRETE IMPLEMENTATIONS: UNION

```
fun union($T_1, T_2$) =
   case expose($T_1$) of
      NONE $\Rightarrow T_2$
   | SOME($L_1, R_1, k_1, v_1$) $\Rightarrow$
      let val $(L_2, v_2, R_2)$ = split($T_2, k_1$)
           val $(L, R)$ = union($L_1, L_2$) || union($R_1, R_2$)
      in join($L$, SOME($k_1, v_1$), $R$)
      end
```

- Returns the value from T_1 if a key appears in both trees.

ANALYSIS OF UNION

```
fun union($T_1, T_2$) =
    case expose($T_1$) of
        NONE $\Rightarrow T_2$
      | SOME($L_1, R_1, k_1, v_1$) $\Rightarrow$
        let val $(L_2, v_2, R_2)$ = split($T_2, k_1$)
            val $(L, R)$ = union($L_1, L_2$) || union($R_1, R_2$)
        in join($L$, SOME($k_1, v_1$), $R$)
        end
```

- `split` costs $O(\log |T_2|)$.
- Two recursive calls to `union`
- `join` costs $O(\log(|T_1| + |T_2|)$

ANALYSIS OF UNION - ASSUMPTIONS

- T_1 is perfectly balanced.
  - `expose` return subtrees of size $|T_1|/2$
  - Each a key from T_1 splits T_2, it splits exactly in half.

ANALYSIS OF UNION

$$W(|T_1|, |T_2|) = \underbrace{2W(|T_1|/2, |T_2|/2)}_{\text{recursive union calls}} + \underbrace{O(\log(|T_1| + |T_2|))}_{\text{split and join}},$$

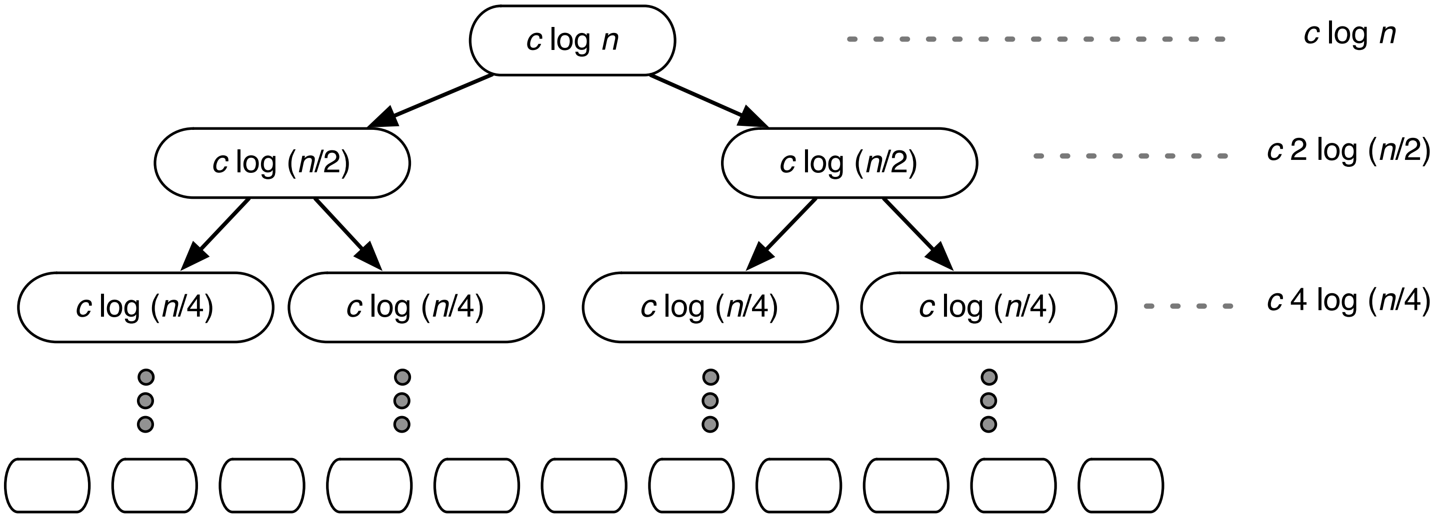
and

$$W(1, |T_2|) = O(\log(1 + |T_2|)).$$

- When $|T_1| = 1$, `expose` give us two empty subtrees L_1 and R_1
- `union`(L_1, L_2) returns L_2, `union`(R_1, R_2) returns R_2 immediately!
- Joining these costs at most $O(\log(|T_1| + |T_2|)) = O(\log(1 + |T_2|)$

ANALYSIS OF UNION

- Let $m = |T_1|$ and $n = |T_2|$



Bottom level: Each box costs log (n/m)

- Leaf dominated (Why?)

ANALYSIS OF UNION

- How many leaves are there in this recursion tree?
    - T_2 has no impact.
    - We get $m = |T_1|$ leaves.
- How deep is the tree?
    - $1 + \log_2 m$
- What is the size of T_2 at the leaves?
    - $n/2^{\log_2 m} = \frac{n}{m}$
- Total cost at the leaves $= O(m \log(1 + \frac{n}{m}))$
- Union cost $= O(m \log(1 + \frac{n}{m}))$

15-210
PARALLEL AND SEQUENTIAL ALGORITHMS AND DATA STRUCTURES

LECTURE 21

SEARCH TREES II: TREAPS

SYNOPSIS

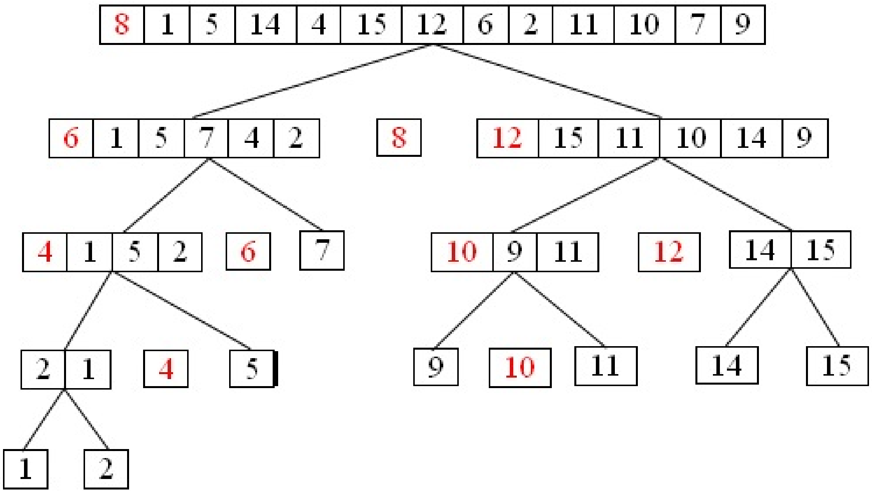
- Overview of Binary Search Trees
- Relationship between Quicksort and BSTs
- Treaps
- Expected Depth of a Treap

BST OVERVIEW

- There are many options for keeping trees balanced.
- `split` and `join` are the main structural operations to implement `find`, `insert`, `delete`, `union`, etc.
- Cost of `split` and `join` are logarithmic in the size of the input and output trees.
- Union needs $O(m \log(1 + \frac{n}{m}))$ work ($m \leq n$).

QUICKSORT AND BSTS

- Write out the recursion tree for quicksort.
  - Assume distinct keys.
- Annotate each node with the pivot picked at that stage.
- You get a BST.



SEQUENCE TO BST

```
fun $qs\_tree(S) =$
  if $|S| = 0$ then $LEAF$
  else let
      val $p =$ pick a pivot from $S$
      val $S_1 = \langle s \in S \mid s < p \rangle$
      val $S_3 = \langle s \in S \mid s > p \rangle$
      val $(T_L, T_R) = (\, qs\_tree(S_1) \parallel qs\_tree(S_3) \,)$
    in
      $NODE(T_L, p, T_R)$
    end
```

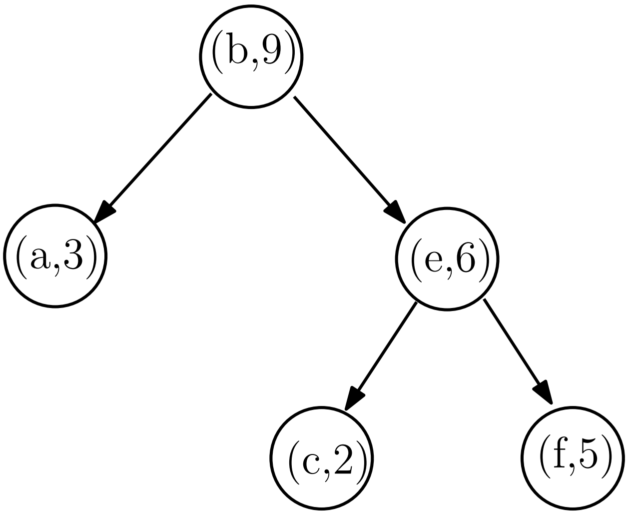
- Unlike Quicksort, we do not know what elements will be in the tree, when we start.
  - We can not select a (n) (future?) element to be the root.

TREAPS

- Treap = TRee + hEAP
- A treap is a randomized BST that maintains balance in a probabilistic way.
- Each element/key gets a unique random priority
- The nodes in the treap satisfy BST property.
    - Keys are stored in-order in the tree.
- The associated priorities satify the (max) heap property.

THE MAX-HEAP PROPERTY

- Priority at each node is greater than the priorities of the children.
- Suppose we have
  $S = (a, 3), (b, 9), (c, 2), (e, 6), (f, 5)$



LET'S DO AN EXAMPLE

- Draw the treap for the following (*key*, *priority*) sequence.

(G,50),(C,35),(E,33),(H,29),(I,25),(B,24),(A,21),(L,16),(J,13),
(K,9),(D,8)

TREAPS

THEOREM

For any set S of unique key-priority pairs, there is exactly one treap T containing the key-priority pairs in S which satisfies the treap properties.

- Key k with highest priority must be at the root.
- All keys $< k$ must be in the left subtree
- All keys $> k$ must be in the right subtree
- Subtrees of k are constructed inductively in the same manner.

Basic BST Operations - Search

```
fun $search$ $T$ $k$ =
let val $(\_, v, \_) = split(T, k)$
in $v$
end
```

BASIC BST OPERATIONS - INSERT

```
fun insert $T$ $(k, v) =$
let val $(L, v', R) = split(T, k)$
in $join(L, SOME(k, v), R)$
end
```

BASIC BST OPERATIONS - DELETE

```
fun delete T k =
let val (L, _, R) = split(T, k)
in join(L, NONE, R)
end
```

- So if `split` and `join` are implemented the other more useful operations are covered.

JOIN AND SPLIT

- $\texttt{split}(T, k) : \mathit{BST} \times \mathit{key} \rightarrow$
  $\mathit{BST} \times (\mathit{data\ option}) \times \mathit{BST}$

- $\texttt{split}$ divides T into two BSTs,
    - one consisting of all the keys from T less than k
    - the other all the keys greater than k
- If k appears in the tree with associated data d then $\texttt{split}$ returns $\mathit{SOME}(d)$
- Otherwise it returns NONE.

JOIN AND SPLIT

- $\mathtt{join}(L, m, R) : \mathtt{BST} \times (\mathtt{key} \times \mathtt{data})\ \mathtt{option} \times \mathtt{BST} \rightarrow \mathtt{BST}$

- Takes a left subtree (L) an optional key-data pair m and a right subtree (R)
  - Assumes all keys in L are less than all keys in R.
  - If present, the optional key is also larger than all keys in L and smaller than all keys in R.
- Creates a new BST that is the union of L and R and m.

SPLIT ON TREAPS

- Split code does not have to change.
- Priority orders do not change.
- Split does not put a larger priority *below* a smaller priority.

Split on Treaps

```
datatype BST = Leaf |
               Node of (BST * BST * key * data)
```

```
fun split(T, k) =
  case T of
    Leaf ⇒ (Leaf, NONE, Leaf)
  | Node(L, R, k′, v) ⇒
      case compare(k, k′) of
        LESS ⇒
          let val (L′, r, R′) = split(L, k)
          in (L′, r, Node(R′, R, k′, v)) end
        EQUAL ⇒ (L, SOME(v), R)
        GREATER ⇒
          let val (L′, r, R′) = split(R, k)
          in (Node(L, L′, k′, v), r, R′) end
```

Join on Treaps

- Join needs to change!
  - The priorities of the roots of two trees need to be compared.
  - The root with the larger priority becomes the new root.
- Basic join took the root of the first tree or the new node as the root.

```
fun join($T_1, m, T_2$) =
  case $m$ of
    SOME($k, v$) ⇒ Node($T_1, T_2, \mathbf{k}, \mathbf{v}$)
  | NONE ⇒
      case $T_1$ of
        Leaf ⇒ $T_2$
      | Node($L, R, \mathbf{k}, \mathbf{v}$) ⇒ Node($L$, join($R$, NONE, $T_2$), $\mathbf{k}, \mathbf{v}$)
```

JOIN ON TREAPS

```
fun $join(T_1, m, T_2) =$
let
    fun $singleton(k, v) = Node(Leaf, Leaf, k, v)$
    fun $join'(T_1, T_2) =$
        case $(T_1, T_2)$ of
            $(Leaf, \_) \Rightarrow T_2$
          | $(\_, Leaf) \Rightarrow T_1$
          | $(Node(L_1, R_1, k_1, v_1), Node(L_2, R_2, k_2, v_2)) \Rightarrow$
                if $(priority(k_1) > priority(k_2))$ then
                    $Node(L_1, join'(R_1, T_2), k_1, v_1)$
                else
                    $Node(join'(T_1, L_2), R_2, k_2, v_2)$
in
    case $m$ of
        $NONE \Rightarrow join'(T_1, T_2))$
      | $SOME(k, v) \Rightarrow join'(T_1, join'(singleton(k, v), T_2))$
end
```

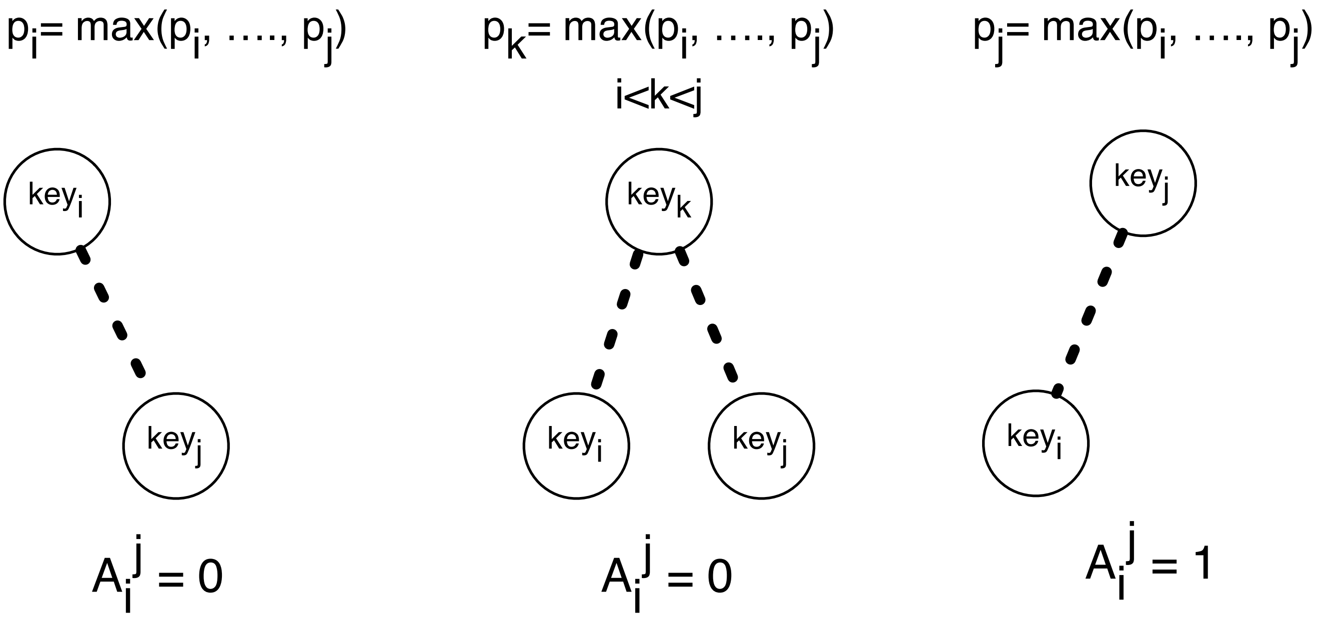
EXPECTED DEPTH OF A KEY

- Cost of split and join depend on the expected depth of a key.
- Given a set of keys K and priorities $p : \mathtt{key} \rightarrow int$
  - Priorities are unique!
- Consider the elements of the tree laid out in order
  - $key_i < key_j \Rightarrow \cdots, key_i, \cdots, key_j, \cdots$
  - $key_j < key_i \Rightarrow \cdots, key_j, \cdots, key_i, \cdots$
- A_i^j is an indicator variable:
  - $A_i^j = 1$ if key_j is an ancestor of key_i in the treap.
  - $A_i^j = 0$ otherwise.

EXPECTED DEPTH OF A KEY

$$\cdots, key_i, \cdots, key_j, \cdots$$

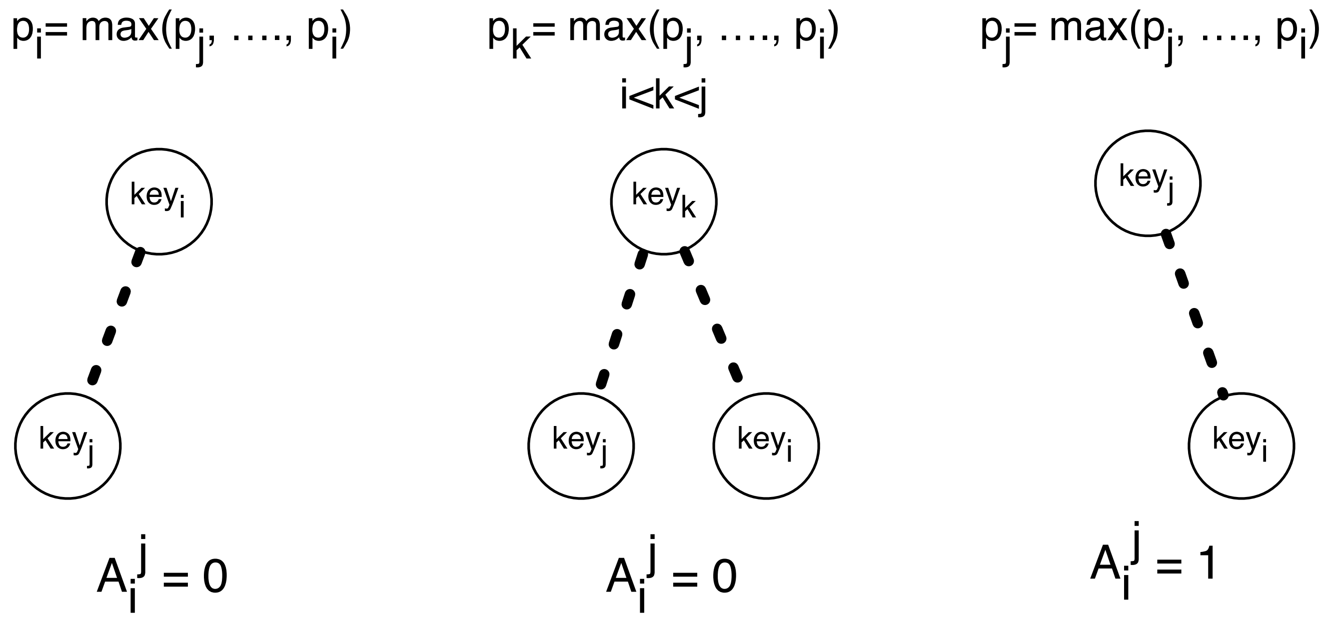
$$key_i < key_j$$



EXPECTED DEPTH OF A KEY

$$\cdots, key_j, \cdots, key_i, \cdots$$

$key_i > key_j$



$p_i = \max(p_j, \ldots, p_i)$

$p_k = \max(p_j, \ldots, p_i)$

$i<k<j$

$p_j = \max(p_j, \ldots, p_i)$

$A_i^j = 0$

$A_i^j = 0$

$A_i^j = 1$

EXPECTED DEPTH OF A KEY

$$\mathbf{E}\left[\text{depth of } i \text{ in } T\right] = \mathbf{E}\left[\sum_{j=1, j\neq i}^{n} A_i^j\right] = \sum_{j=1, j\neq i}^{n} \mathbf{E}\left[A_i^j\right].$$

$$\mathbf{E}\left[A_i^j\right] = \frac{1}{|j-i|+1} \quad \text{(Why?)}$$

EXPECTED DEPTH OF A KEY

$$\begin{aligned}
\mathbf{E}\left[\text{depth of } i \text{ in } T\right] &= \sum_{j=1, j\neq i}^{n} \frac{1}{|j-i|+1} \\
(\text{Split } |\ | \Rightarrow) &= \sum_{j=1}^{i-1} \frac{1}{i-j+1} + \sum_{j=i+1}^{n} \frac{1}{j-i+1} \\
(\text{Change variables } \Rightarrow) &= \sum_{k=2}^{i} \frac{1}{k} + \sum_{k=2}^{n-i+1} \frac{1}{k} \\
&= H_i - 1 + H_{n-i+1} - 1 \\
(\ln n < H_n < \ln n + 1 \Rightarrow) &< \ln i + \ln(n-i+1) \\
&= O(\log n)
\end{aligned}$$

- Relative (sorted) position of a key determines expected depth in treap.

COST OF SPLIT AND JOIN

THEOREM

For treaps

- `join`(T_1, m, T_2) returning T
- `split`$(T, (k, v))$

have $O(\log |T|)$ expected work and span.

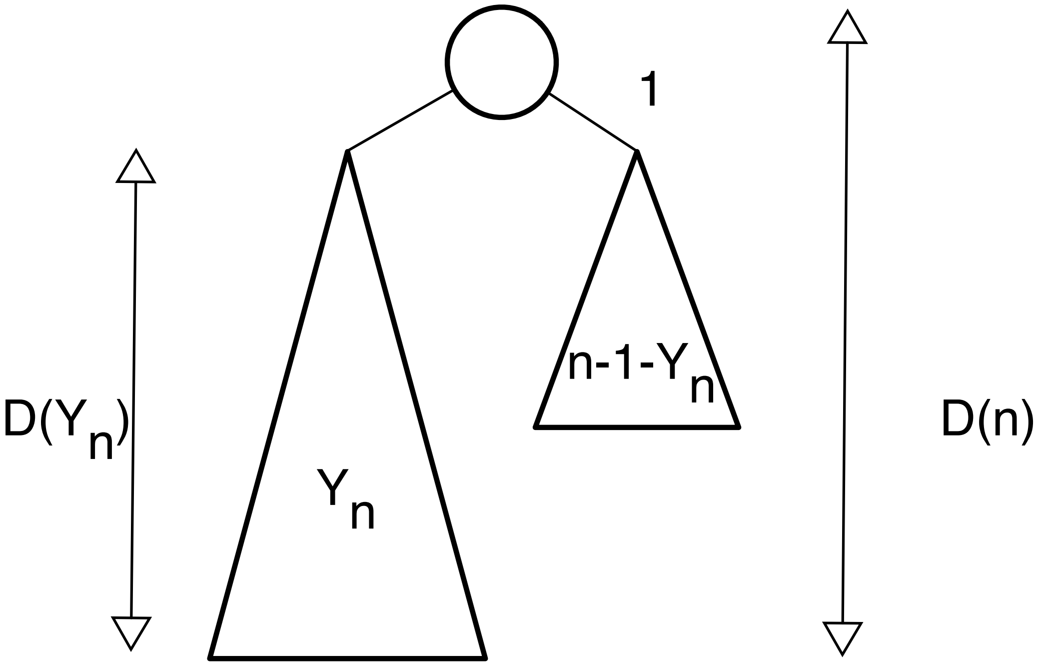
- See notes for short proofs.

EXPECTED MAX DEPTH OF A TREAP

- Expected depth of treap node is $O(\log n)$
  - Find takes on the average $O(\log n)$ work and span.
- What is the expected maximum depth of a treap?

  - Why is this important?
  - Expected worst-case cost!
- But $\mathbf{E}\left[\max_i\{A_i\}\right] \neq \max_i\{\mathbf{E}\left[A_i\right]\}$!
- It turns out this is almost the same problem as the expected span of the quicksort.

EXPECTED MAX DEPTH OF A TREAP



- Y_n is the size of the larger partition.
- $D(n) = D(Y_n) + 1 \Rightarrow D(n) \in O(\log n)$

15-210
PARALLEL AND SEQUENTIAL ALGORITHMS AND DATA STRUCTURES

LECTURE 23

MORE WITH TREES

SYNOPSIS

- Ordered Sets and Tables
- Bingle Revisited
- Augmenting Balanced Trees
- Ordered Tables with Reduced Values
- Application Examples

ORDERED SETS AND TABLES

- So far, we did not worry about the ordering of the values/keys in sets and tables.
  - Find, union, intersect, merge, etc.
- For many applications, exploiting any order is very important!
  - Find all elements between 3 and 17.
  - Find all customers who bought more that 5 of one item.
  - Find all emails in the week of March 31st.
- Ordered sets and tables.

Ordered Set ADT

- We have a totally ordered universe $\mathbb{U}$, and $\mathbb{S}$ represents the set of all subsets of $\mathbb{U}$.
- With the following operations

all operations supported by the Set ADT, and

$$
\begin{array}{llll}
\texttt{last}(S) & : \ \mathbb{S} \rightarrow \mathbb{U} & = & \max\ S \\
\texttt{first}(S) & : \ \mathbb{S} \rightarrow \mathbb{U} & = & \min\ S \\
\texttt{split}(S, k) & : \ \mathbb{S} \times \mathbb{U} \rightarrow \mathbb{S} & = & (\{k' \in S \mid k' < k\}, k \overset{?}{\in} S, \\
 & \quad \times \mathit{bool} \times \mathbb{S} & & \quad \{k' \in S \mid k' > k\}) \\
\texttt{join}(S_1, S_2) & : \ \mathbb{S} \times \mathbb{S} \rightarrow \mathbb{S} & = & S_1 \cup S_2, \text{assuming} \\
 & & & \quad \max S_1 < \min S_2 \\
\texttt{getRange}(S, k_1, k_2) & : \ \mathbb{S} \times \mathbb{U} \times \mathbb{U} \rightarrow \mathbb{S} & = & \{k \in S \mid k_1 \le k \le k_2\}
\end{array}
$$

ORDERED SET ADT

- Underlying implementation uses trees.
- `first` and `last` are easy
    - `first` traverses down the left spine to the minimum value.
    - `last` traverses down the right spine to the maximum value.
- `getRange` involves two splits.

IMPROVISING BINGLE

```
signature INDEX = sig
  type word = string
  type docId = string
  type 'a seq
  type index
  type docList

  val makeIndex : (docId * string) seq -> index
  val find : index -> word -> docList
  val And : docList * docList -> docList
  val AndNot : docList * docList -> docList
  val Or : docList * docList -> docList
  val size : docList -> int
  val toSeq : docList -> docId seq
end
```

- `docList` is a set.
- `index` is a table.

IMPROVISING BINGLE

- We want to limit the search to certain domains (e.g., `cmu.edu`)
  - or docs with a certain name.
- We want to add

```
val inDomain : domain * docList -> docList
```

- For example

```
inDomain("cs.cmu.edu",
         and(find idx "cool", find idx "TAs"))
```

IMPROVISING BINGLE

- Assume doc ids are URLs.
- Assume they are “reverse” lexicographically ordered.
    - The last character is the most important!

```
fun inDomain(domain, L) =
    getRange(L, domain, string.prepend(domain, "$"))
```

- $ is a character that is greater than any character.

AUGMENTING BALANCED TREES

- Sets (and underlying trees) hold the key and any associated values.
- We can add other additional values to help with other search operations.
  - Track key positions and certain subset sizes.
- `rank(S, k)`: How many elements in S are less than k?
- `select(S, i)`: Which element in S has rank i?
- `splitIdx(S,i)`: Split S into two sets: first i keys and the remaining $n - i$ keys.

AUGMENTING BALANCED TREES

$$\begin{array}{llll}
\texttt{rank}(S,k) & : \ \mathbb{S}\times\mathbb{U}\rightarrow \mathit{int} & = & |\{k' \in S \mid k' < k\}| \\
\texttt{select}(S,i) & : \ \mathbb{S}\times \mathit{int} \rightarrow \mathbb{U} & = & k \text{ such that } |\{k' \in S \mid k' < k\}| = i \\
\texttt{splitIdx}(S,i) & : \ \mathbb{S}\times \mathit{int} \rightarrow & = & (\{k \in S \mid k < \texttt{select}(S,i)\}, \\
 & \quad \mathbb{S}\times\mathbb{S} & & \ \{k \in S \mid k \geq \texttt{select}(S,i)\})
\end{array}$$

- Without additional information stored with the keys, these operations would take $\theta(|S|)$ work.

AUGMENTING BALANCED TREES

- Let $S = \{1, 2, 3, 4, 5, 6\}$
- `rank(S, 4)` $= |\{1, 2, 3\}| = 3$
- `select(S, 3)` $= 4$ since `rank(S, 4)` $= 3$
- `splitIdx(S, 3)` $= (\{1, 2, 3\}, \{4, 5, 6\})$

AUGMENTING BALANCED TREES

- At each node keep the size of the subtree.
- This allows `size` and the three other operations in $O(d)$ work with d as the depth of the tree.
- Size can be computed on the fly by adding 1 to the sum of the subtree sizes!

SELECT WITH AUGMENTED TREES

```
fun select(T, i) =
   case expose(T) of
         NONE ⇒ raise Range
      | SOME(L, R, k) ⇒
         case compare(i, |L|) of
               LESS ⇒ select(L, i)
            | EQUAL ⇒ k
            | GREATER ⇒ select(R, i − |L| − 1)
```

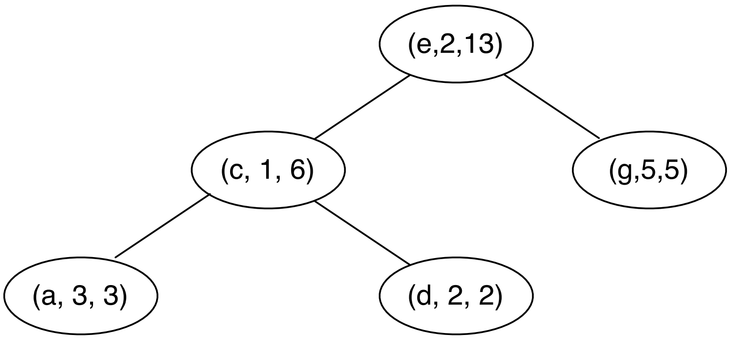
RANK AND SPLITIDX

- `rank` is easy: just split and return the size of the left tree!
- `splitIdx` is just like split (or you navigate using sizes (as opposed to key values))

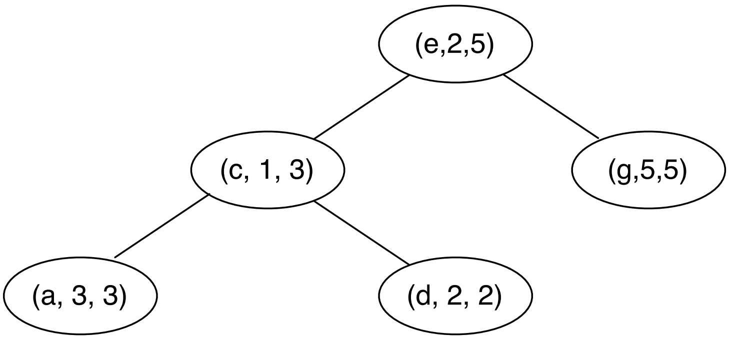
ORDERED TABLES WITH REDUCED VALUES

- Maintain at each node a “sum” based on an associative operator f.
  - Updated during insert/delete, merge, extract, etc.
- Given $f : v \times v \rightarrow v$, and I_f
  - All operations on ordered tables are supported, and
  - $$\texttt{reduceVal}(A) : \mathbb{T} \rightarrow v = \texttt{reduce}\ f\ I_f\ \ A$$
  - We want to be able to do `reduceVal` in $O(1)$ work (assuming f needs $O(1)$ work).
  - f is known beforehand!

ORDERED TABLES WITH REDUCED VALUES



f is $+$



f is max

IMPLEMENTATION

```
datatype Treap = Leaf | Node of (Treap × Treap
                                 × key × data × data)
fun reduceVal(T) =
  case T of
    Leaf ⇒ Reduce.I
  | Node(_, _, _, _, r) ⇒ r
fun makeNode(L, R, k, v) =
  Node(L, R, k, v, Reduce.f(reduceVal(L),
                            Reduce.f(v, reduceVal(R))))
```

IMPLEMENTATION

```
fun $join'(T_1, T_2) =$
    case $(T_1, T_2)$ of
        $(\texttt{Leaf}, \_) \Rightarrow T_2$
      | $(\_, \texttt{Leaf}) \Rightarrow T_1$
      | $(\texttt{Node}(L_1, R_1, k_1, v_1, s_1), \texttt{Node}(L_2, R_2, k_2, v_2, s_2)) \Rightarrow$
            if $(\texttt{priority}(k_1) > \texttt{priority}(k_2))$ then
                $\texttt{makeNode}(L_1, \texttt{join}(R_1, T_2), k_1, v_1)$
            else
                $\texttt{makeNode}(\texttt{join}(T_1, L_2), R_2, k_2, v_2)$
```

EXAMPLE APPLICATION - SALES DATA

- Sales information are kept by the time stamp in an ordered table.
  - $(2/3/2013 - 12:30, \$120)$
- Find the total sales between t_1 and t_2
- f is $+$
- $\texttt{reduceVal}(\texttt{getRange}(T, t_1, t_2))$ takes $O(logn)$ work

EXAMPLE APPLICATION - STOCK DATA

- Stock prices information are kept by the time stamp in an ordered table.
  - $(2/3/2013 - 12:30, \$120/share)$
- Find the maximum price between t_1 and t_2
- f is max
- $\texttt{reduceVal}(\texttt{getRange}(T, t_1, t_2))$ takes $O(logn)$ work

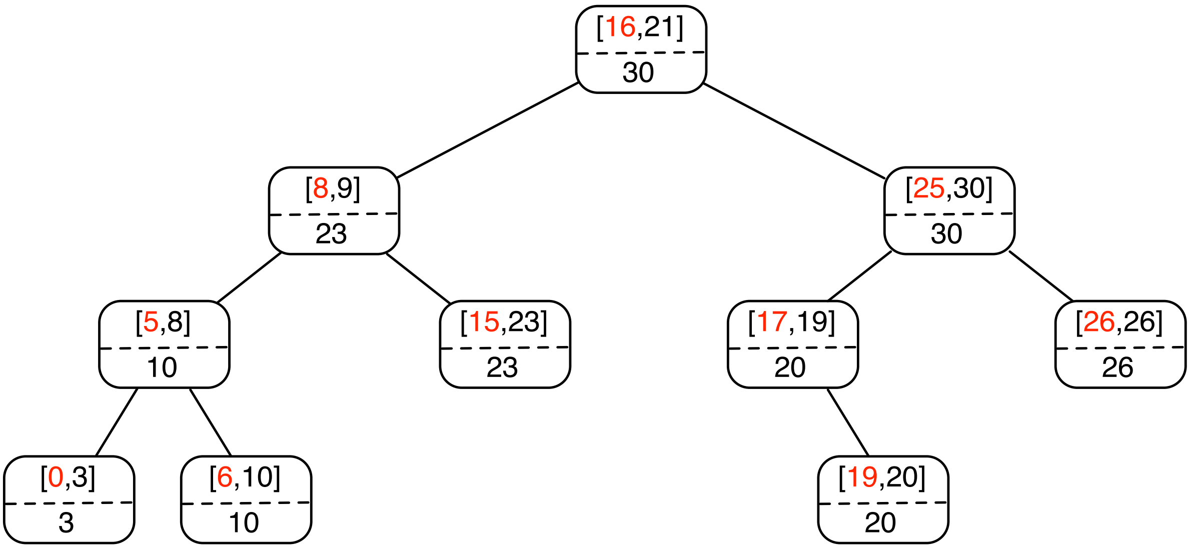
EXAMPLE APPLICATION- INTERVAL TREES

- An interval is a region on the real number line starting at x_l and ending at x_r
- an interval table supports the following operations on intervals:

$\texttt{insert}(A, I)$	:	$\mathbb{T} \times (\texttt{real} \times \texttt{real}) \rightarrow \mathbb{T}$	*insert interval I into table A*
$\texttt{delete}(A, I)$	:	$\mathbb{T} \times (\texttt{real} \times \texttt{real}) \rightarrow \mathbb{T}$	*delete interval I from table A*
$\texttt{count}(A, x)$	:	$\mathbb{T} \times \texttt{real} \rightarrow \texttt{int}$	*return the number of intervals crossing x in A*

INTERVAL TREES

- Organize intervals as a BST based on lower-boundary as key
- Use the max upper boundary in the subtree as additional information.



COUNTING INTERVALS

```
datatype intTree = Leaf | Node of (intTree × intTree
                                   × real × real × real)

fun overlap(x, low, high) =
      if ($x \geq low$ & $x \leq high$) then 1 else 0

fun countInt(T, x) =
   case T of
      Leaf ⇒ 0
    | Node(L, R, low, high, max) ⇒
         if ($x > max$) then 0
         else countInt(L, x)+
              overlap(x, low, high)+
              if ($x > low$) then countInt(R, x) else 0
```

15-210
PARALLEL AND SEQUENTIAL ALGORITHMS AND DATA STRUCTURES

LECTURE 24

DYNAMIC PROGRAMMING

SYNOPSIS

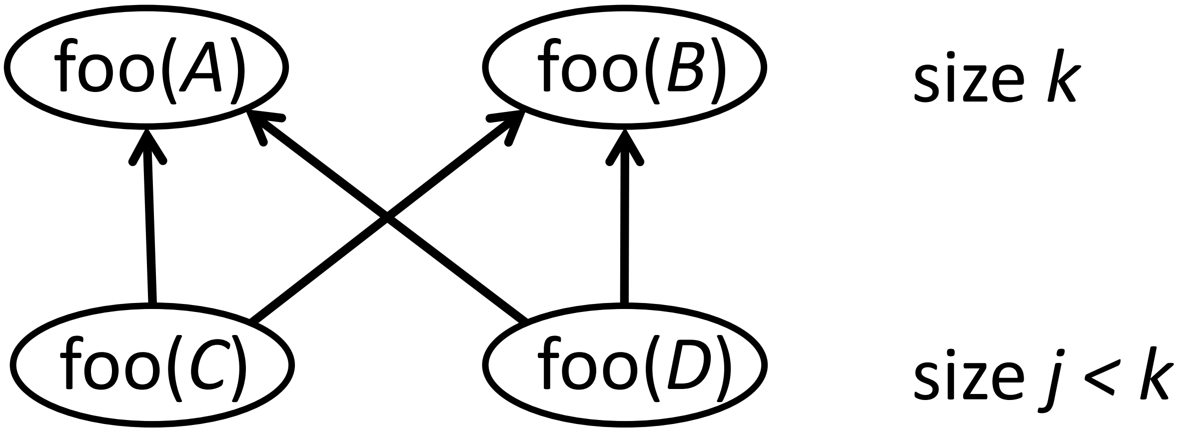
- Dynamic Programming
- Subset Sum Problem
- Minimum Edit Distance Problem
- Additional example applications

ALGORITHMIC PARADIGMS CONTRASTED

- Inductive Paradigms combine solutions to smaller subproblem(s).

Paradigm	Subproblems	Reuse of Solutions
Divide and Conquer	> 1	NO
Contraction	$= 1$	NO
Greedy	$= 1$	NO
Dynamic Programming	> 1	YES

Reusing Solutions



- You can save some work if you semember the solutions to the smaller subproblems.

RESUSING SOLUTIONS

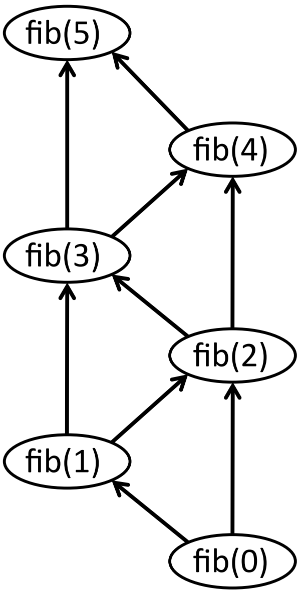
- How much work does this code need?

```
fun fib(n) =
    if (n ≤ 1) then 1
    else fib(n − 1) + fib(n − 2)
```

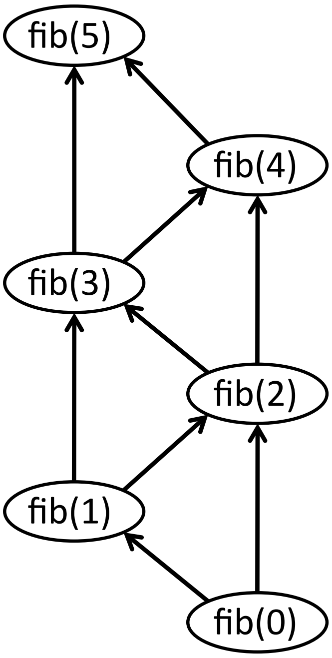
- It turns out $W_{fib}(n) = O(c^n)$ (Why?)

REUSING SOLUTIONS

- It also turns out that `fib`(n) can be computed with $O(n)$ work.
  - Note that n is not the right measure for modeling work here (Why?) but it is convenient!



SOLUTION COMPOSITION GRAPH



- DAG
- Each node is a subproblem instance
- Edges model dependences
- Edges go from smaller to larger subproblems
- Vertices with no in-edges are base cases
- Vertices with no out edges are the instance we are trying to solve.

DYNAMIC PROGRAMMING

- Dynamic programming can be seen as evaluating a DAG by navigating from the leaves to the root.
    - Computing the subsolutions at each node as needed and when possible.
- Work and span fall out of the DGA structure!
    - Work: sum over nodes
    - Span: Find the longest path!
- Many DP solutions have significant parallelism, but some do not.

DYNAMIC PROGRAMMING

- The challenge is to find the appropriate DAG structure for a given problem.
- DP is most suitable for optimization problems.
    - Solution optimizes (minimizes/maximizes) some criteria.
- DP is also suitable for decision problems.
    - Is there a solution to this instance?

DYNAMIC PROGRAMMING

- Top-down approach
  - Starts at the root
  - Uses recursion to solve the subproblems
  - But remembers the solutions – memoization.
  - Usually elegant and evaluates only the needed subproblems.
- Bottom-up approach
  - Starts at the leaves
  - Traverses the DAG in some fashion.
  - All subproblems may need to be computed.
  - More parallelizable.
- Coming up with the abstract inductive structure is important.
  - Sharing and coding comes later.

THE SUBSET SUM PROBLEM

THE SUBSET SUM (SS) PROBLEM

Given a multiset of positive integers S and a positive integer value k, determine if there is any $X \subseteq S$ such that $\sum_{x \in X} x = k$.

- Given $S = \{1, 4, 2, 9, 9\}$
    - No solution for $k = 8$
    - For $k = 7$ $\{1, 4, 2\}$ is a solution.
- $NP-$hard if k is unconstrained.
- We will include k in the work bounds.
- k is polynomial in $|S|$, work is polynomial in $|S|$.
- *Pseudo-polynomial work* solution.

THE SUBSET SUM PROBLEM

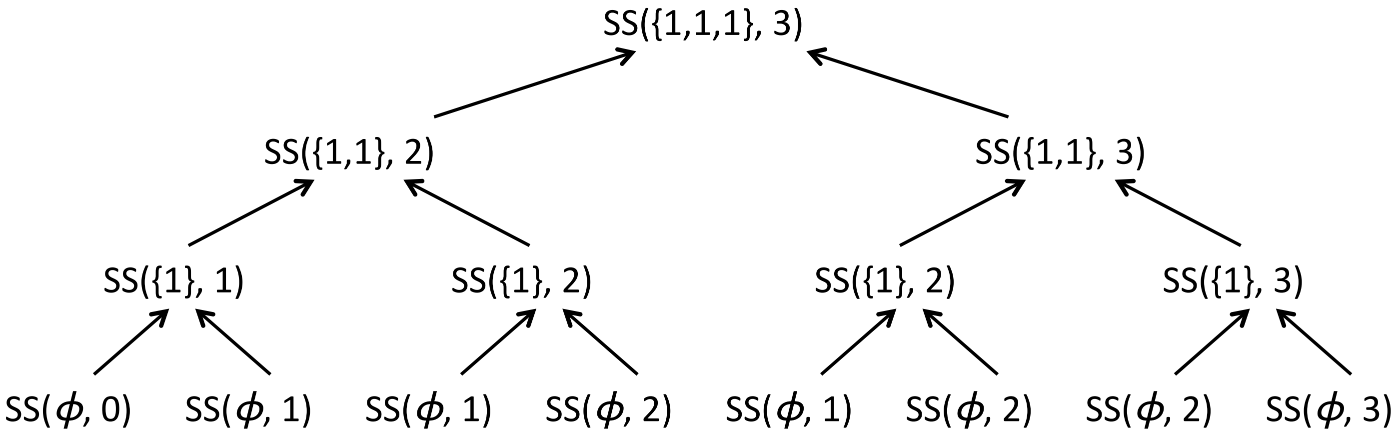
- Brute force: Consider all 2^n subset for a total work of $O(n2^n)$.
- Divide and Conquer: also ends up being exponential work.
- Sharing solutions however works.

THE SUBSET SUM PROBLEM

- To solve $SS(S, k)$, pick some element $a \in S$
- Solve (recursively) $SS(S \setminus \{a\}, k - a)$
  - If there is a solution, we are done.
- If not, solve $SS(S \setminus \{a\}, k)$.

```
fun  $SS(S, k) =$
   case  ($showl(S)$,  $k$)  of
      ($\_$, 0) $\Rightarrow$ $true$
    | ($NIL$, $\_$) $\Rightarrow$ $false$
    | ($CONS(a, R)$, $\_$) $\Rightarrow$
         if  ($a > k$)  then  $SS(R, k)$
         else  ($SS(R, k - a)$  orelse  $SS(R, k)$)
```

THE SUBSET SUM PROBLEM DAG



SS({1,1,1}, 3)

SS({1,1}, 2)

SS({1,1}, 3)

SS({1}, 1)

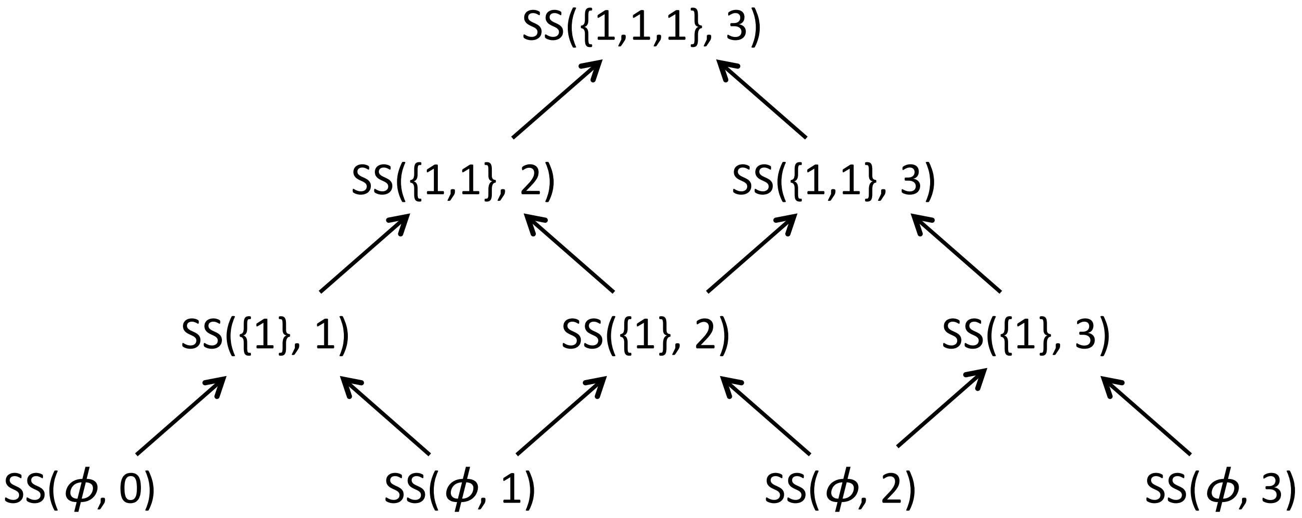
SS({1}, 2)

SS({1}, 2)

SS({1}, 3)

SS(ϕ, 0) SS(ϕ, 1) SS(ϕ, 1) SS(ϕ, 2) SS(ϕ, 1) SS(ϕ, 2) SS(ϕ, 2) SS(ϕ, 3)

THE SUBSET SUM PROBLEM DAG



SS({1,1,1}, 3)

SS({1,1}, 2) SS({1,1}, 3)

SS({1}, 1) SS({1}, 2) SS({1}, 3)

SS(ϕ, 0) SS(ϕ, 1) SS(ϕ, 2) SS(ϕ, 3)

- How many distinct subproblems do we need to solve?

THE SUBSET SUM PROBLEM

- For $SS(S, k)$, there are only $|S|$ distinct lists ever used.
- The second argument decreases down to 0, so has at most $k + 1$ values.
- So we have at most $|S|(k + 1) = O(k|S|)$ instances.
- Each instance has constant work $\Rightarrow$ total $O(k|S|)$ work.
- Longest path in DAG is $|S|$ $\Rightarrow$ span is $O(|S|)$
    - $O(k)$ parallelism.

THE SUBSET SUM PROBLEM

- Why *pseudo-polynomial*?
- For k, the input size is $\log k$, but the work is $O(2^{\log k}|S|)$
  - *Exponential* in input size!
- If $k \leq |S|^c$ for some constant c, then work is $O(k|S|) = O(|S|^{c+1})$ on input of size $c \log |S| + |S|$

MINIMUM EDIT DISTANCE

MINIMUM EDIT DISTANCE (MED)

Given a character set Σ and two sequences of characters $S = \Sigma^*$ and $T = \Sigma^*$, determine the minimum number of insertions and deletions of single characters required to transform S to T.

- Start with $S = \langle A, B, C, A, D, A \rangle$
  - Delete C
  - Delete last A
  - Insert a C
- You get $T = \langle A, B, A, D, C \rangle$
- So $MED(S, T) = 3$

APPLICATIONS OF MED

- Spelling correction
    - What is an English word close to *Ynglisd*?
- Storing multiple versions of files efficiently.
- Approximate matching of genome sequences

MINIMUM EDIT DISTANCE

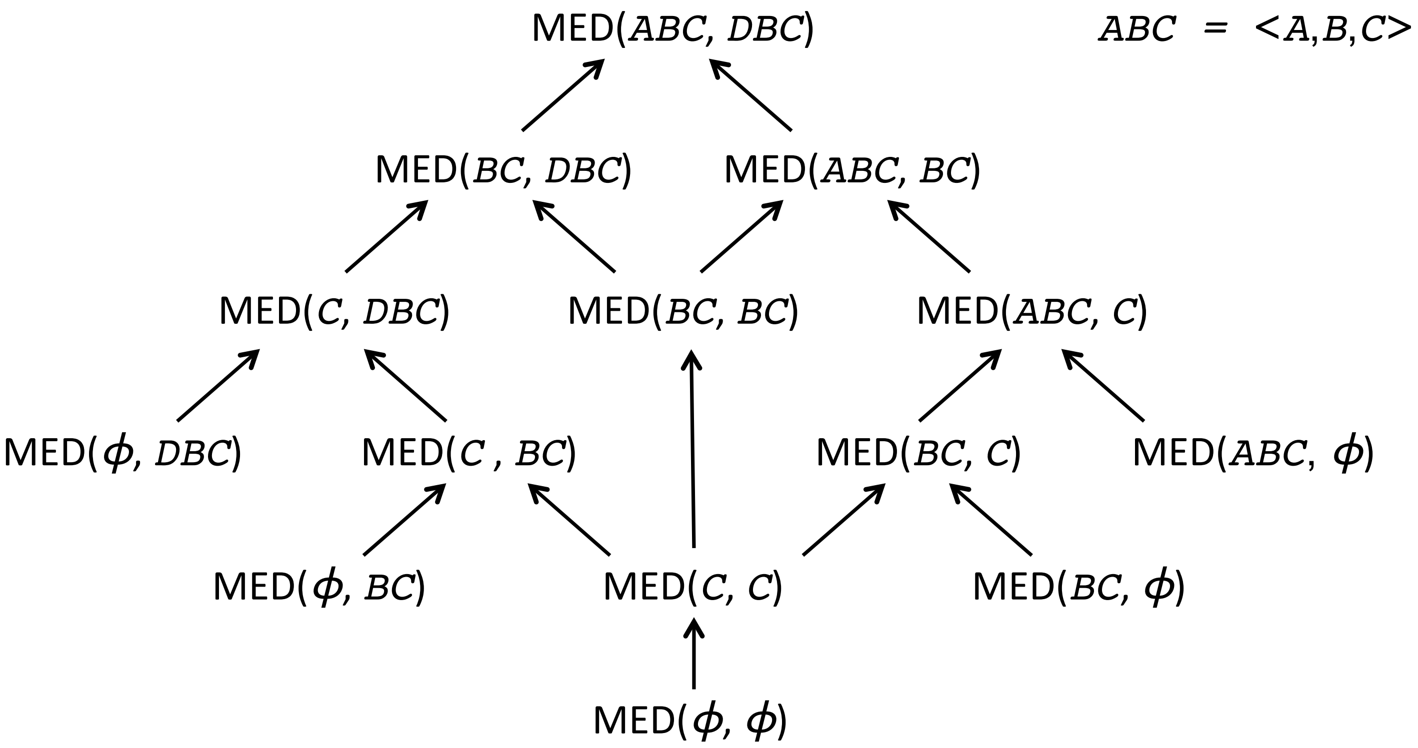
- Given $S = s :: S'$ and $T = t :: T'$
- If $s = t$, $MED(S, T)$ is determined by S' and T'
- Otherwise we have two subproblems:
    - Find $MED(S, T')$ – consider a deletion from T to get T'
    - Find $MED(S', T)$ – consider an deletion to S to get S'
- Find the minimum and add 1.

MINIMUM EDIT DISTANCE

```
**fun** $\mathit{MED}(S, T) =$
    **case** $(\mathtt{showl}(S), \mathtt{showl}(T))$ **of**
        $(\_, \mathit{NIL}) \Rightarrow |S|$
      $|$ $(\mathit{NIL}, \_) \Rightarrow |T|$
      $|$ $(\mathit{CONS}(s, S'), \mathit{CONS}(t, T')) \Rightarrow$
            **if** $(s = t)$ **then** $\mathit{MED}(S', T')$
            **else** $1 + \min(\mathit{MED}(S, T'), \mathit{MED}(S', T))$
```

- If run recursively, this would take exponential work.
    - Binary tree with linear depth!
- But there is significant sharing!

MINIMUM EDIT DISTANCE



ABC = <A,B,C>

MED(ABC, DBC)

MED(BC, DBC) MED(ABC, BC)

MED(C, DBC) MED(BC, BC) MED(ABC, C)

MED(ϕ, DBC) MED(C , BC) MED(BC, C) MED(ABC, ϕ)

MED(ϕ, BC) MED(C, C) MED(BC, ϕ)

MED(ϕ, ϕ)

MINIMUM EDIT DISTANCE

- There are at most $|S| + 1$ possible values for the first argument.
- There are at most $|T| + 1$ possible values for the second argument.
- So we have $(|S| + 1) \times (|T| + 1) = O(|S||T|)$ possible subproblems, each of constant work.
  - Total work is $O(|S||T|)$.
- Total span is $O(|S| + |T|)$ (Why?)

THE LONGEST COMMON SUBSEQUENCE (LENGTH)

- A longest common subsequence of strings S_1 and S_2 is a longest subsequence shared by both.
- $LCS(ABCDEF, EBCEG) = BCE$
- May be empty or not necessarily unique.
- $LLCS(S_1, S_2)$ computes the length of the LCS.
- Subproblem structure is very similar to MED. (Work it out!)

OPTIMAL CHANGE

- For a currency with coins $C_1, C_2, \dots C_n = 1$ (cents), what is the minimum number of coins needed to make K cents of change.
- US Currency has 25, 10, 5, 1 cent coins.
- To give back 63 cents, you need to give 25+25+10+1+1+1, a total of 6 coins.
    - Greedy works in this case, but not always
    - If you had a 21 cent coin (for some strange reason), greedy would not work.
- DP solutions solves two subproblems $K_1 = i$ and $K_2 = K - i$ for all $i = 1, \dots \lfloor K/2 \rfloor$
- Then chooses i that minimizes the sum of the solutions

0-1 KNAPSACK

- Items with "benefit" p_i and cost w_i
  - $x_i = 1$ or 0 – take item i or not.
- Maximize $\sum_{j=1}^{n} p_j \cdot x_j$
- Subject to $\sum_{j=1}^{n} w_j \cdot x_j \leq c$
- Optimal Exam Strategy Problem (:-)
  - Questions 1 through n, worth $p_1, \ldots p_n$ points.
  - Time estimate for solving question j is w_j
  - You have T units of time.
  - Which questions do you solve to maximize your grade?
  - Subproblem structure is resembles the thinking for subset sum problem

OPTIMAL MATRIX MULTIPLICATION

- We need to multiply n matrices $A_1 \times A_2 \times \cdots A_n$
    - A_i has sizes $p_{i-1} \times p_i$ and A_{i+1} has sizes $p_i \times p_{i+1}$
    - Multiplying A_i and A_{i+1} needs $O(p_{i-1} \cdot p_i \cdot p_{i+1})$ work
- What is the best way to “parenthesize” the sequence to minimize the number of scalar mutiplications?
- $m[i,j]$ is the minimum number of scalar multiplications for multiplying $A_i \times \cdots \times A_j$
    - A subproblem

OPTIMAL MATRIX MULTIPLICATION

$$m[i,j] = \begin{cases} 0 & i = j \\ \min_{i \le k < j}\{m[i,k] + m[k+1,j] + p_{i-1} \cdot p_k \cdot p_j\} & i < j \end{cases}$$

- Find that k that minimizes the cost of multiplying $A_i \times \cdots \times A_j$
- We need to compute $m[1,n]$ and how we got that (the choice of k's when we are minimizing subproblems)

15-210
PARALLEL AND SEQUENTIAL ALGORITHMS AND DATA STRUCTURES

LECTURE 25

DYNAMIC PROGRAMMING – II

SYNOPSIS

- Top-down Dynamic Programming
- Bottom-up Dynamic Programming
- Optimal Binary Search Trees

TOP-DOWN DP

- Run the recursive code as is:
    - Start with the root
    - Work down to the leaves
- Memoization: We need to avoid redundant computation.
    - If we encounter the same arguments, we just look up the solution
    - If not, we compute once and store in a memo table.
- Checking for equal arguments could be costly.
    - We use simple surrogates for actual arguments (e.g., integers)

TOP-DOWN DP FOR MED

- MED takes two sequences and on each recursive call, uses suffixes of the original sequences.
  - There is a one-to-one mapping from non-negative integers to suffixes (rather to suffix lengths!)
  - Could also use prefixes!
  - This makes indexing a bit easier.

ORIGINAL MED CODE

```
fun MED($S$, $T$) =
    case (showl($S$), showl($T$)) of
        (_, NIL) ⇒ $|S|$
      | (NIL, _) ⇒ $|T|$
      | (CONS($s$, $S'$), CONS($t$, $T'$)) ⇒
            if ($s = t$) then MED($S'$, $T'$)
            else $1 + \min$(MED($S$, $T'$), MED($S'$, $T$))
```

MED with Surrogates

```
fun $MED(S, T)$ = let
    fun $MED'(i, 0) = i$
      | $MED'(0, j) = j$
      | $MED'(i, j)$ = case $(S_i = T_j)$ of
                              $true \Rightarrow MED'(i-1, j-1)$
                            | $false \Rightarrow 1 + \min(MED'(i, j-1),$
                                                  $MED'(i-1, j))$
in
    $MED'(|S|, |T|)$
end
```

- `MED′` has i and j, instead of S and T
  - i represents $S\langle 0, \ldots, i-1 \rangle$
  - j represents $T\langle 0, \ldots, j-1 \rangle$
- No memo table yet!

MEMO TABLE

- We can now add a memo table, accessed with (i, j)
    - We can also use a two dimensional array!

```
fun memo f (M, a) =
  case find(M, a) of
    SOME(v) ⇒ (M, v)
  | NONE ⇒ let
      val (M′, v) = f(M, a)
    in
      (update(M′, a, v), v)
    end
```

MEMOIZED MED

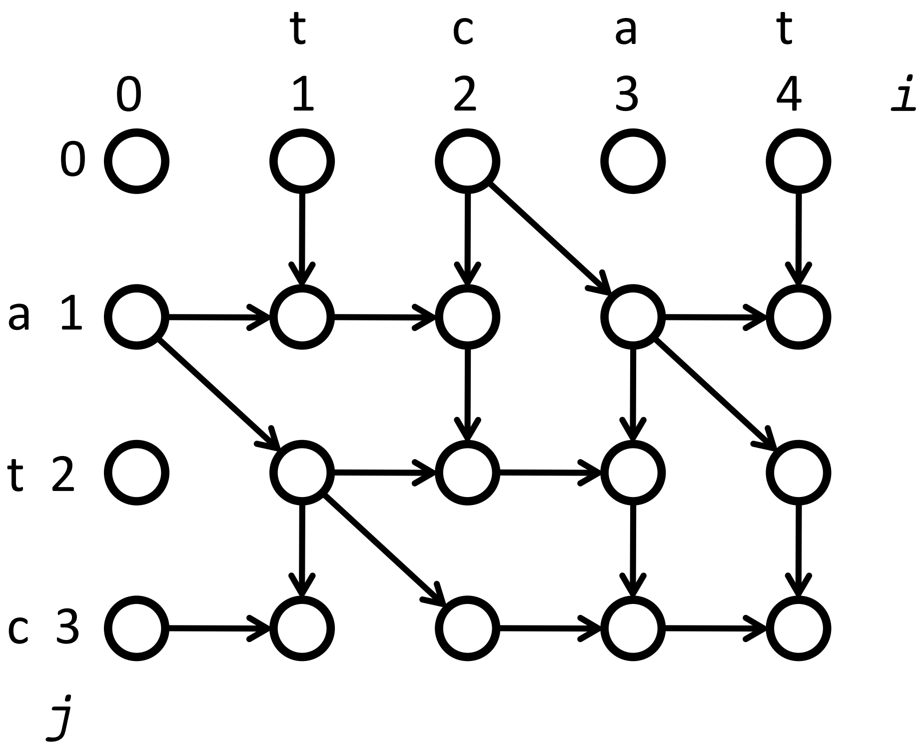
```
fun $MED(S, T)$ = let
    fun $MED'(M, (i, 0)) = (M, i)$
      | $MED'(M, (0, j)) = (M, j)$
      | $MED'(M, (i, j))$ = case $(S_i = T_j)$ of
                                  true ⇒ $MED''(M, (i-1, j-1))$
                                | false ⇒ let
                                      val $(M', v_1) = MED''(M, (i, j-1))$
                                      val $(M'', v_2) = MED''(M', (i-1, j))$
                                  in $(M'', 1 + \min(v_1, v_2))$ end
    and $MED''(M, (i, j))$ = memo $MED'$ $(M, (i, j))$
in
    $MED'(\{\}, (|S|, |T|))$
end
```

- Purely functional
- but highly sequential

BOTTOM-UP DP

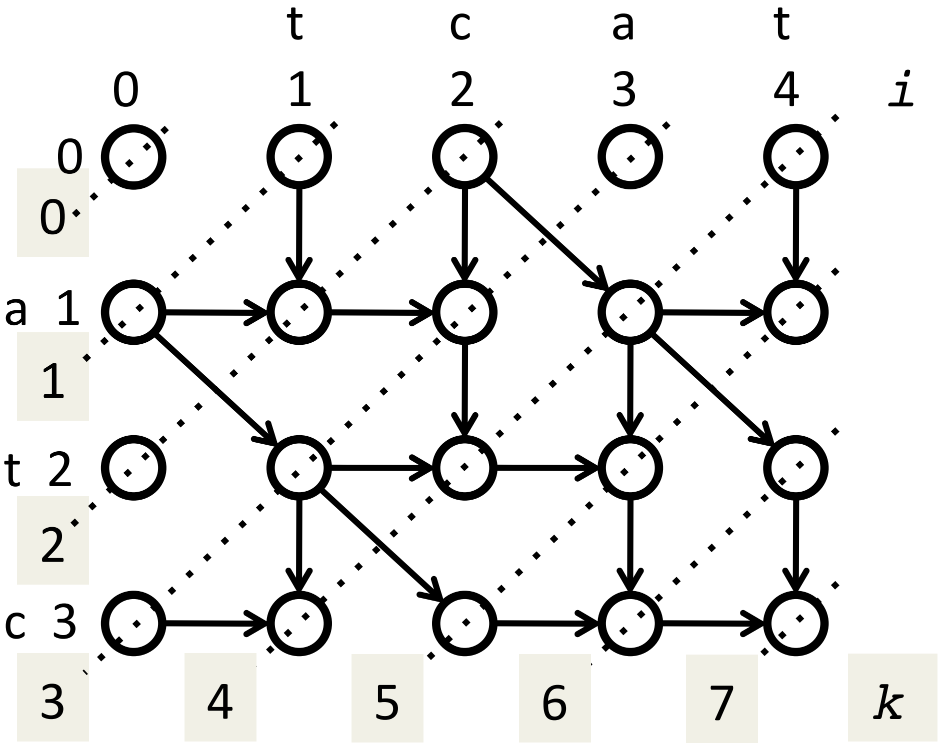
- Start with the leaves
- Works through the subproblems consistent with the DAG
  - if (u, v) is a dependency edge in the DAG, compute u before v, for all such u.
  - All values will be available for v when they are needed!
- Uses a memo table.
- Understanding the DAG structure is important

BOTTOM-UP DP FOR MED



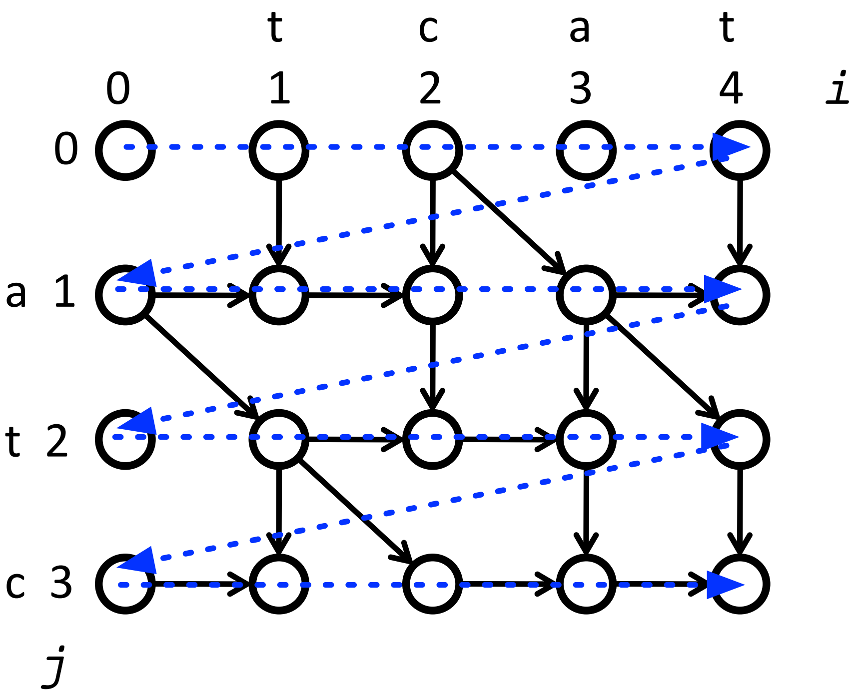
Dag for $MED(''tcat'', ''atc'')$

BOTTOM-UP DP FOR MED



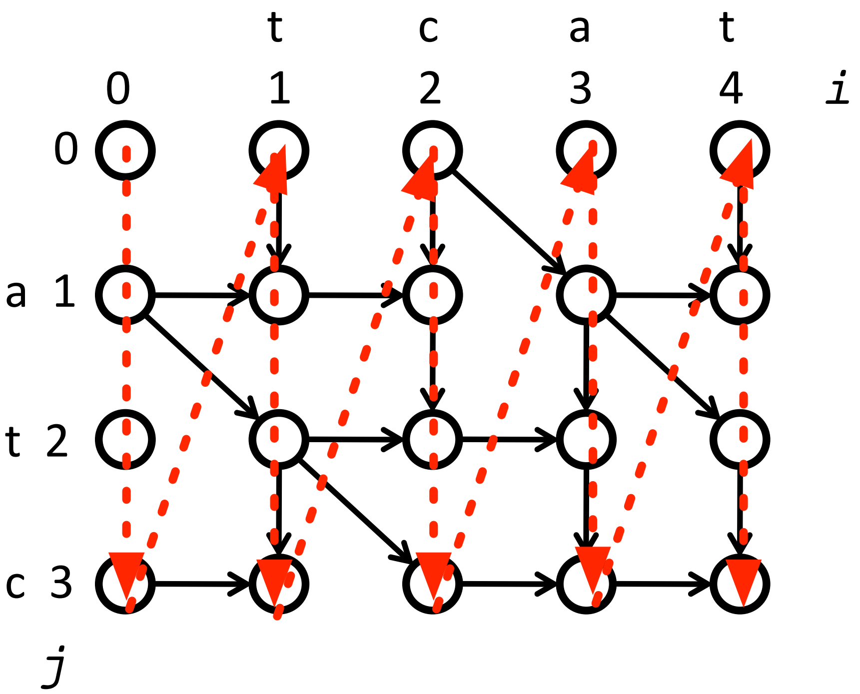
We can go by diagonals.

BOTTOM-UP DP FOR MED



We can go by rows.

BOTTOM-UP DP FOR MED



We can go by columns.

BOTTOM-UP DP FOR MED

```
fun $MED(S, T)$ = let
  fun $MED'(M, (i, 0)) = i$
    | $MED'(M, (0, j)) = j$
    | $MED'(M, (i, j)) =$ case $(S_i = T_j)$ of
                           true $\Rightarrow M_{i-1,j-1}$
                         | false $\Rightarrow 1 + \min(M_{i,j-1}, M_{i-1,j})$

  fun diagonals$(M, k) =$
    if $(k > |S| + |T|)$ then $M$
    else let
      val $s = \max(0, k - |T|)$
      val $e = \min(k, |S|)$
      val $M' = M \cup \{(i, k - i) \mapsto MED'(M, (i, k - i)) : i \in \{s, \ldots, e\}\}$
    in
      diagonals$(M', k + 1)$
    end

in
  diagonals$(\{\}, 0)$
end
```

BOTTOM-UP DP FOR MED

- In Round 0, we compute $M_{0,0}$
- In Round 1, we compute $M_{0,1}$ and $M_{1,0}$
- In Round 2, we compute $M_{0,2}$, $M_{1,1}$, $M_{2,0}$
- In Round 3, we compute $M_{0,3}$, $M_{1,2}$, $M_{2,1}$, $M_{3,0}$
- …
- How about parallelism?

OPTIMAL BINARY SEARCH TREES

- Let's revisit BSTs
  - The cost of finding a key is proportional to the depth of the key in the tree.
  - Fully balanced BST with n nodes $\Rightarrow$ average depth is $\log n$
- Suppose you have a (fixed/static) dictionary and you know the **probability** that a given key will be accessed
- What is the BST structure with the lowest overall cost?

OPTIMAL BINARY SEARCH TREES

OPTIMAL BST

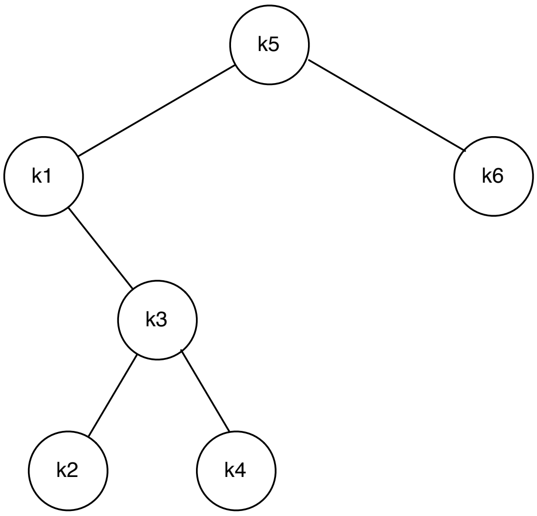
The *optimal binary search tree* (OBST) problem is given an ordered set of keys S and a probability function $p : S \to [0 : 1]$, to find $\hat{T}$

$$\hat{T} = arg\,min_{T \in \mathtt{Trees}(S)} \left(\sum_{s \in S} d(s, T) \cdot p(s) \right)$$

where $\mathtt{Trees}(S)$ is the set of all BSTs on S, and $d(s, T)$ is the depth of the key s in the tree T (Assume the root has depth 1).

OPTIMAL BINARY SEARCH TREES

key	k_1	k_2	k_3	k_4	k_5	k_6
$p(\text{key})$	$1/8$	$1/32$	$1/16$	$1/32$	$1/4$	$1/2$



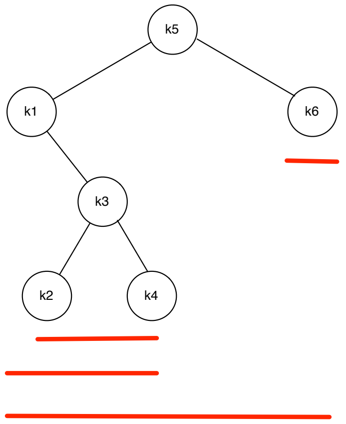
$$Cost = \frac{1}{8}\times 2 + \frac{1}{32}\times 4 + \frac{1}{16}\times 3 + \frac{1}{32}\times 4 + \frac{1}{4}\times 1 + \frac{1}{2}\times 2 = \frac{31}{16}$$

OPTIMAL BINARY SEARCH TREES

- How many binary search trees of n distinct keys are there?
  - Hint: Think of matrix chain multiplication!
- in DP, an optimal solution should be based on optimal subproblem solutions.
- One of the keys (S_r) must be at the root of the optimal tree.
  - Both subtrees **must be optimal**.
- How do we select S_r?
  - Pick the key with highest probability and put it at the root, and recurse?
  - Does not really work!

OPTIMAL BINARY SEARCH TREES

- Try all elements as a potential root
- For each, recursively find their optimal solutions
- Pick the best among the $|S|$ possibilities.
- All elements under a root are contiguous in the sorted sequence.



OPTIMAL BINARY SEARCH TREES

- Use (i, j) as a surrogate for the tree spanning $S_i, \ldots, S_j$.
- Let T be the tree covering $S_i, \ldots, S_j$ with root $S_r, i \leq r \leq j$, with T_L T_R as the subtrees.

$$\begin{aligned}
\texttt{Cost}(T) &= \sum_{s \in T} d(s, T) \cdot p(s) \\
&= p(S_r) + \sum_{s \in T_L} (d(s, T_L) + 1) \cdot p(s) + \sum_{s \in T_R} (d(s, T_R) + 1) \cdot p(s) \\
&= \sum_{s \in T} p(s) + \sum_{s \in T_L} d(s, T_L) \cdot p(s) + \sum_{s \in T_R} d(s, T_R) \cdot p(s) \\
&= \sum_{s \in T} p(s) + \texttt{Cost}(T_L) + \texttt{Cost}(T_R)
\end{aligned}$$

- Find the $r, i \leq r \leq j$ that minimizes this cost.

OPTIMAL BINARY SEARCH TREES

```
fun $OBST(S) =$
    if $|S| = 0$ then $0$
    else $\left(\sum_{s \in S} p(s)\right) + \min_{i \in \langle 1 \ldots |S| \rangle} (OBST(S_{1,i-1}) +$
                                                    $OBST(S_{i+1,|S|}))$
```

- How many possible subproblems are there?
  - A subsequence can end at n different positions
  - For the i^{th} end position there are i possible start positions.
- $\sum_{i=1}^{n} i = n(n+1)/2 \in O(n^2)$ possible subproblems.
- Longest path of dependences in the DAG is $O(n)$ since recursion can go down for n levels (Why?)

WORK AND SPAN

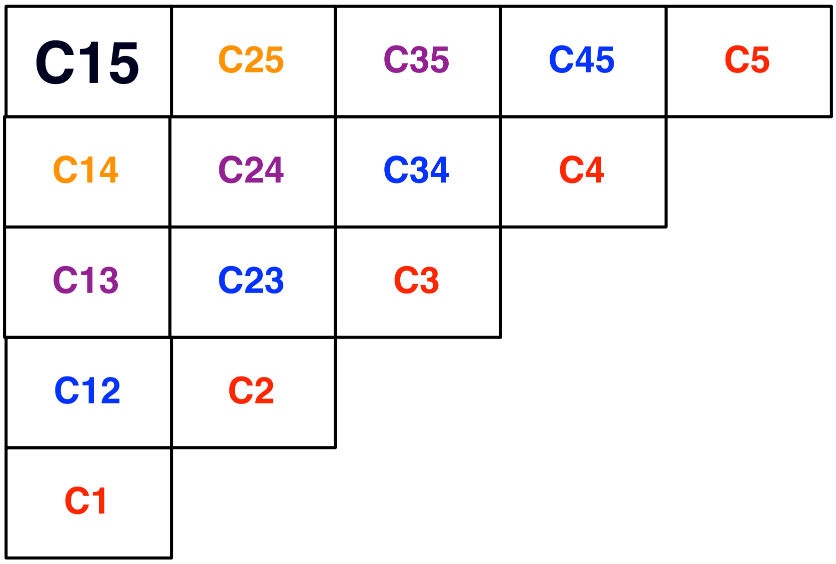
- Cost of each subproblem is not uniform! (Why?)
- Each subproblem has $O(n)$ work and $O(\log n)$ span (Why?)
- We get total $O(n^3)$ work and $O(n \log n)$ span. (Why?)

CODE FOR OPTIMAL BST

```
fun $\mathit{OBST}(S)$ = let
    fun $\mathit{OBST}'(i, l)$ =
        if $l = 0$ then 0
        else $\sum_{k=0}^{l-1} p(S_{i+k}) + \min_{k=0}^{l-1}(\mathit{OBST}'(i, k) +$
                                      $\mathit{OBST}'(i + k + 1, l - k - 1))$
in
    $\mathit{OBST}'(1, |S|)$
end
```

BOTTOM-UP OPTIMAL BST

- For a bottom up version, a triangular table is sufficient



C15	C25	C35	C45	C5
C14	C24	C34	C4	
C13	C23	C3		
C12	C2			
C1				

cij = optimal cost of the tree covering Sij

15-210
PARALLEL AND SEQUENTIAL ALGORITHMS AND DATA STRUCTURES

LECTURE 26

HASH TABLES

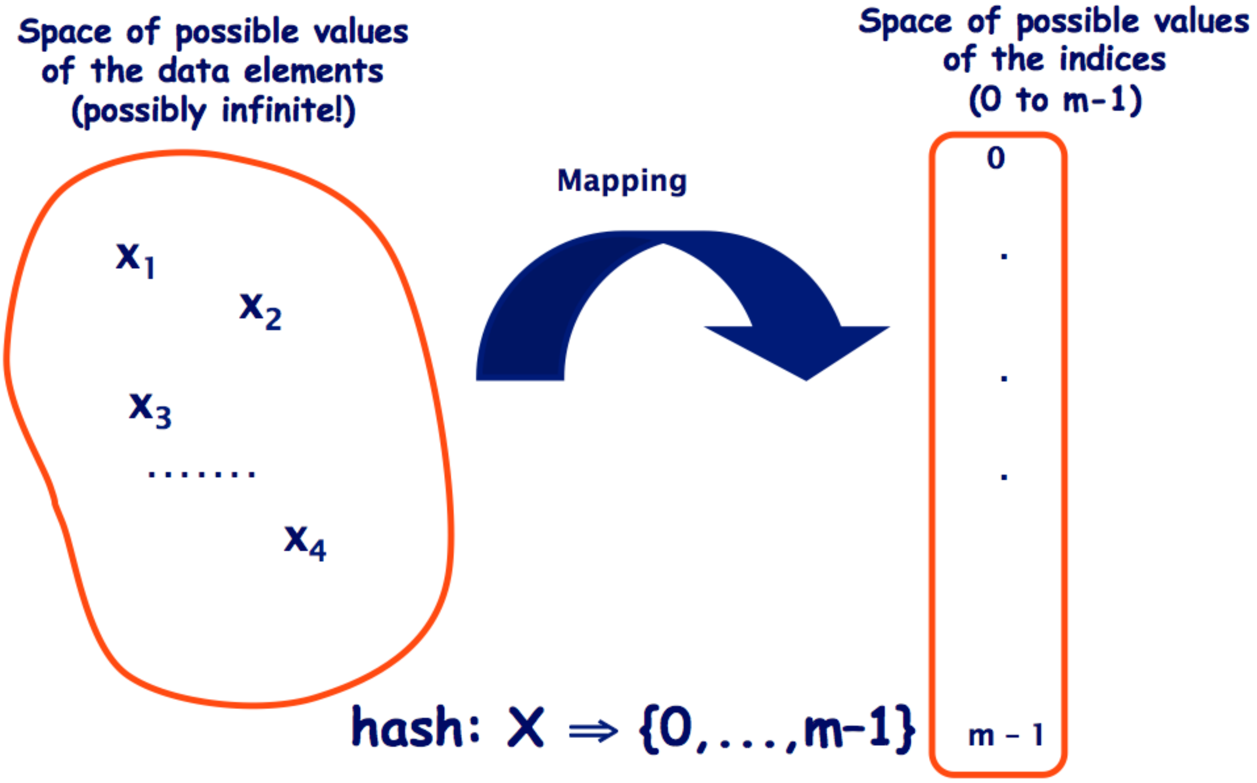
SYNOPSIS

- Hashing and Hash Tables
- Handling Collisions
    - Linear Probing
    - Quadratic Probing

HASH TABLES – BASIC IDEAS

- Data structure that allows you to quickly insert, delete, and retrieve items with expected $O(1)$ work.
- Relies on
  - a fixed size array data structure (of some size m), and
  - a hash function that can map from a potentially infinite space of keys to integer indexes $[0, \ldots, m-1]$
- Disadvantages
  - Collisions
  - Increased memory use to avoid collisions
  - Not work efficient for *findmin*, *findmax*, or *extracting keys in sorted order*

HASH TABLE - BASIC IDEAS



hash: X $\Rightarrow$ {0,...,m-1}

HASH FUNCTIONS

- There is a deep theory behind hash functions.
- We will be interested in some simple functions.
- We will assume hash functions have the idealized property of *simple uniform hashing*:
  - The hash function uniformly distributes keys in range $[0, \ldots, m-1]$
  - Hash value for one key is independent of the hash value for another key.

HASH FUNCTIONS

- For integers key we can use a linear congruential hash function

$$h(x) = (ax + b) \bmod m$$

where $a \in [1, \ldots, m-1]$, $b \in [0, \ldots, m-1]$, and m is prime.

HASH FUNCTIONS

- For strings, we can use a polynomial like

$$h(S) = \left(\sum_{i=1}^{|S|} s_i a^i \right) \bmod m$$

HASH TABLES

- Support `insert`, `find` and `delete`.
- Can implement abstract data types `Set` and `Table`.
- Do not require total ordering on the universe of keys.
- *Collision* is the main issue
  - Two keys hash to the same location.
  - Impossible to avoid if we do not know the keys in advance
    - Size of key universe $>>$ size of table.

COLLISIONS

- For a table size of 365, one needs 23 keys for a 50% chance of collision and 66 for a 99% chance of collision (Why?)
    - Birthday paradox

HANDLING COLLISIONS

- **Separate chaining**
  - Store elements not in a table, but in linked lists (containers,bins) hanging off the table.
- **Open addressing**:
  - Put everything into the table, but not necessarily into cell $h(k)$.
- **The perfect hash**:
  - When you know the keys in advance, construct hash functions that avoids collisions entirely.
- **Multiple-choice hashing/Cuckoo hashing**:
  - Consider exactly two locations $h_1(k)$ and $h_2(k)$ only.

HANDLING COLLISIONS

- We will only consider the first two.
- We will assume we have a set n keys K and a hash function $h : \mathit{key} \rightarrow [0, \ldots, m - 1]$ for some m.

SEPARATE CHAINING

- Maintain an array of linked lists (buckets).
- Keys that hash to the same value live in the same list at location $h(k)$
- **Insertion**: Insert at the beginning
    - Multiple inserts for the same key ⇒ traverse the list
    - May as well insert at the end.
- **Find**: hash to $h(k)$ and search in the list.
- **Delete**: remove from the list.

SEPARATE CHAINING

- Costs depend on the *load factor* $\lambda = n/m$ which is also the average length of a list.

SEPARATE CHAINING

- Assume $h(k)$ takes $O(1)$ work and we have simple uniform hashing
- *Unsuccessful search* takes expected $\Theta(1 + \lambda)$ work.
  - $O(1)$ for $h(k)$ and λ for traversing the list.

SEPARATE CHAINING

- *Successful search* takes expected $\Theta(1+\lambda)$ work.
- Cost of *Successful search* = Cost of *unsuccessful search* at the time of insertion (Why?)
- With i keys, the unsuccesssful search would take $(1+i/m)$ work.
- Averaging over i we get

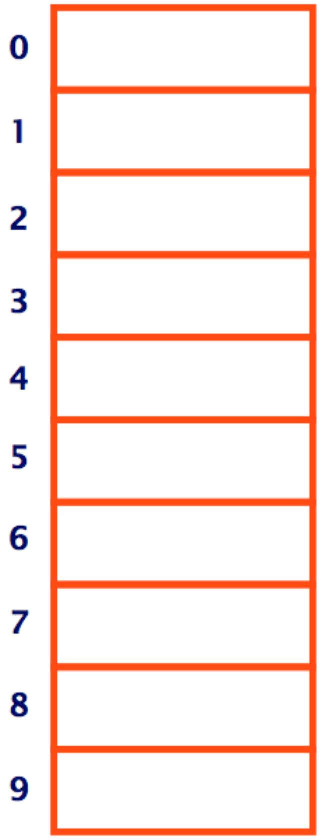
$$\frac{1}{n}\sum_{i=0}^{n-1}(1+i/m) = 1+(n-1)/2m = 1+\lambda/2-\lambda/2m = \Theta(1+\lambda)$$

- Considering constant factors, successful search looks at 1/2 the list on the average.

OPEN ADDRESSING

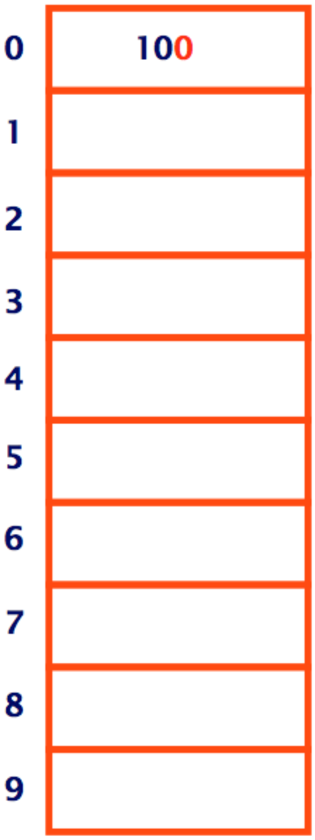
- No lists – everything is stored in the array directly
- The arrays is some constant factor larger than the maximum number of keys we want to store.

AN EXAMPLE



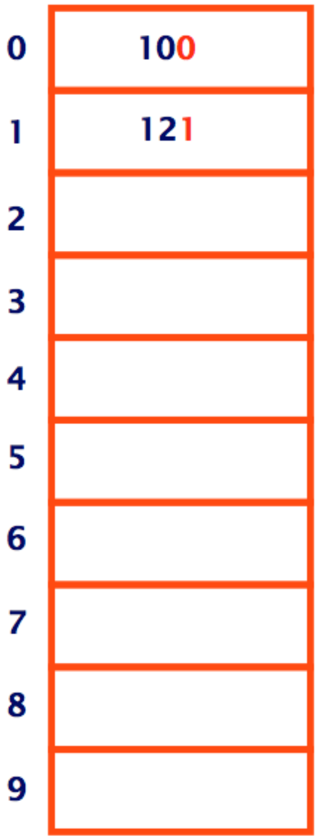
Initially the hash table is empty

AN EXAMPLE



Insert 100

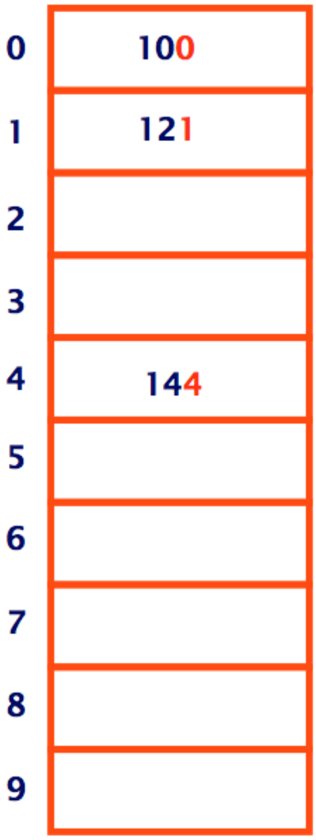
AN EXAMPLE



0	100
1	121
2	
3	
4	
5	
6	
7	
8	
9	

Insert 121

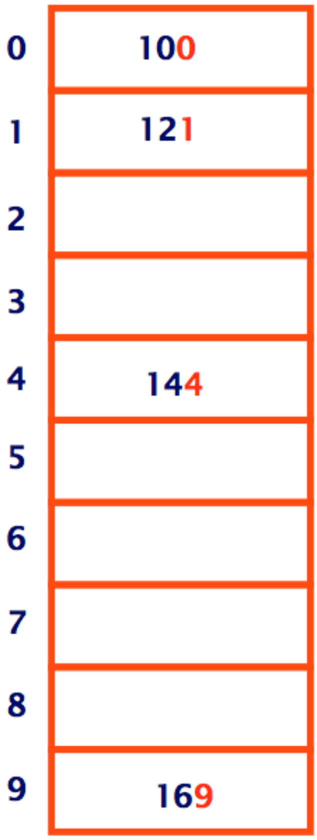
AN EXAMPLE



0	100
1	121
2	
3	
4	144
5	
6	
7	
8	
9	

Insert 144

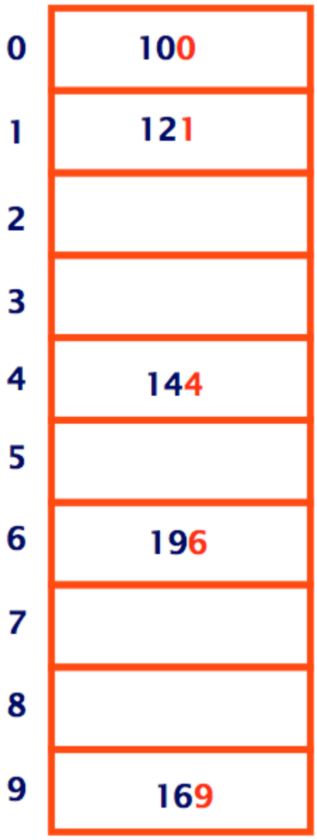
AN EXAMPLE



0	100
1	121
2	
3	
4	144
5	
6	
7	
8	
9	169

Insert 169

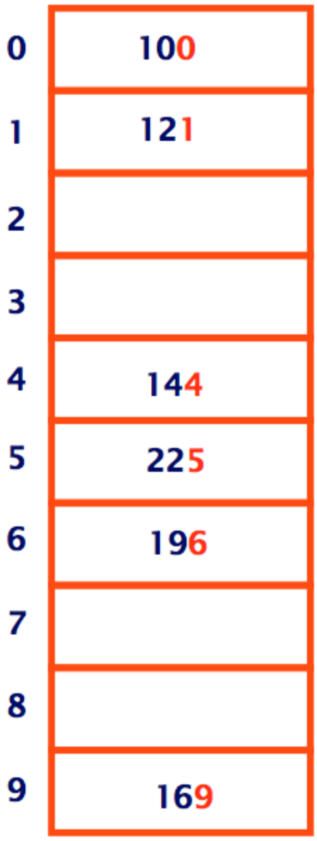
AN EXAMPLE



0	100
1	121
2	
3	
4	144
5	
6	196
7	
8	
9	169

Insert 196

AN EXAMPLE



0	100
1	121
2	
3	
4	144
5	225
6	196
7	
8	
9	169

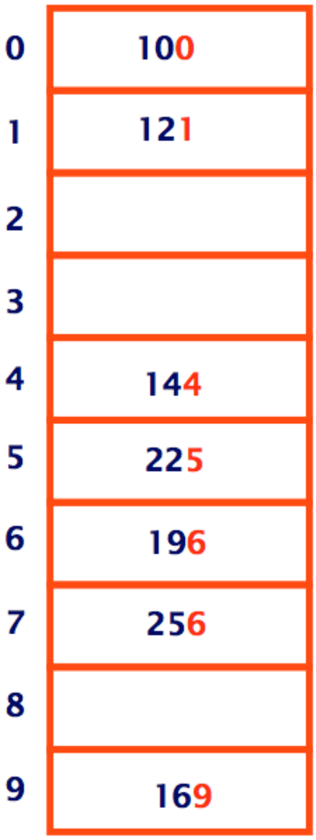
Insert 225

AN EXAMPLE

0	100
1	121
2	
3	
4	144
5	225
6	196
7	256
8	
9	169

Insert 256 COLLISION because location 6 is full. Try location 6+1=7

AN EXAMPLE



0	100
1	121
2	
3	
4	144
5	225
6	196
7	256
8	
9	169

Insert 289 COLLISION because location 9 is full. Try location (9+1)mod 10=0

AN EXAMPLE

0	100
1	121
2	
3	
4	144
5	225
6	196
7	256
8	
9	169

Insert 289 COLLISION because location 9 is full.

Try location (9+1)mod 10= 0 FULL

AN EXAMPLE

0	100
1	121
2	
3	
4	144
5	225
6	196
7	256
8	
9	169

Insert 289 COLLISION because location 9 is full.

Try location (9+1)mod 10= 0 FULL

Try location (9+2)mod 10= 1 FULL

AN EXAMPLE

0	100
1	121
2	289
3	
4	144
5	225
6	196
7	256
8	
9	169

Insert 289 COLLISION because location 9 is full.

Try location (9+1)mod 10= 0 FULL

Try location (9+2)mod 10= 1 FULL

Try location (9+3)mod 10= 2 AVAILABLE

AN EXAMPLE

0	100
1	121
2	289
3	
4	144
5	225
6	196
7	256
8	
9	169

Insert 324 COLLISION because location 4 is full.

Try location (4+1)mod 10= 5 FULL

AN EXAMPLE

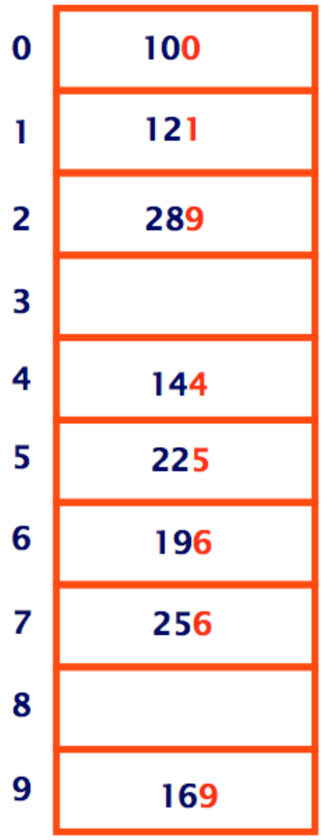
0	100
1	121
2	289
3	
4	144
5	225
6	196
7	256
8	
9	169

Insert 324 COLLISION because location 4 is full.

Try location (4+1)mod 10= 5 FULL

Try location (4+2)mod 10= 6 FULL

AN EXAMPLE



0	100
1	121
2	289
3	
4	144
5	225
6	196
7	256
8	
9	169

Insert 324 COLLISION because location 4 is full.

Try location (4+1)mod 10= 5 FULL

Try location (4+2)mod 10= 6 FULL

Try location (4+3)mod 10= 7 FULL

AN EXAMPLE

0	100
1	121
2	289
3	
4	144
5	225
6	196
7	256
8	324
9	169

Insert 324 COLLISION because location 4 is full.

Try location (4+1)mod 10= 5 FULL

Try location (4+2)mod 10= 6 FULL

Try location (4+3)mod 10= 7 FULL

Try location (4+4)mod 10= 8 AVAILABLE

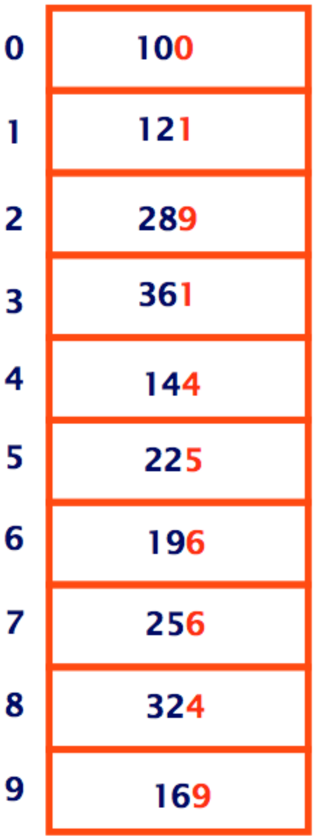
AN EXAMPLE

0	100
1	121
2	289
3	
4	144
5	225
6	196
7	256
8	324
9	169

Insert 361 COLLISION because location 1 is full.

Try location (1+1)mod 10= 2 FULL

AN EXAMPLE



0	100
1	121
2	289
3	361
4	144
5	225
6	196
7	256
8	324
9	169

Insert 361 COLLISION because location 1 is full.

Try location (1+1)mod 10= 2 FULL

Try location (1+2)mod 10= 3 AVAILABLE

OPEN ADDRESSING

- Open addressing uses an ordered sequence of locations.
- $h(k, i)$ gives us the i^{th} location for key k.
- $\langle\, h(k, 0), h(k, 1), h(k, 2), \ldots\,\rangle$ is the *probe sequence*.
- Try these locations in order until an empty cell is found and insert there.

OPEN ADDRESSING - INSERT

```
fun $insert(T, k) =$
let
    fun $insert'(T, k, i) =$
        case $nth\ T\ h(k, i)$ of
            $NONE \Rightarrow update(h(k, i), k)\ T$
          | $\_ \Rightarrow insert'(T, k, i + 1)$
in
    $insert'(T, k, 1)$
end
```

- T must be an ST array - otherwise work and span are not constant.
- Need to check if table is full and the key is already in the table or not.

OPEN ADDRESSING-SEARCH

```
fun $\mathtt{find}(T, k) =$
let
    fun $\mathtt{find'}(T, k, i) =$
        case $\mathtt{nth}\ T\ h(k, i)$ of
            $\mathit{NONE} \Rightarrow \mathtt{false}$
          | $\mathit{SOME}(k') \Rightarrow$ if $(eq(k, k'))$ then $\mathtt{true}$
                                 else $\mathtt{find'}(T, k, i + 1)$
in
    $\mathtt{find'}(T, k, 1)$
end
```

OPEN ADDRESSING-DELETE

- We can not just delete an items and set its cell to *NONE*! (Why ?)
- `find` will stop searching if it encounters an empty cell.
- Use *lazy delete*
    - Instead of deleting, use a special value `HOLD`.

```
datatype α entry = EMPTY | HOLD | FULL of α
```

- Find and Insert will need to be changed accordingly.
- Lazy delete effectively increases load factor.
- Rehashing to the rescue!

OPEN ADDRESSING

- Linear Probing
- Quadratic Probing
- Double Hashing

LINEAR PROBING

- We check cell at $h(k, i) = (h(k) + i) \mod m$ in i^{th} probe.
- m possible probe sequences.
- Keys tend to cluster – *primary clustering*.
    - Inserts add to a cluster
    - Probe sequences get longer and longer

IMPACT OF CLUSTERING

- Assume table is half full ($\lambda = 1/2$)
- Minimum clustering when every other cell is empty!
- Average probes for insert is 3/2
  - One probe to check cell $h(k)$
  - $+$ with 1/2 chance try the next cell (which by design should be empty)

IMPACT OF CLUSTERING

- Worst case: all keys are clustered to the second half of the array. (Remember $\lambda = 1/2 \Rightarrow m = 2n$)
- How many probes for positions 0 through $n - 1$?
  - 1 (Why?)
- How many probes when initial hash is to cell n?
  - n (Why?)
- How many probes when initial hash is to cell $n + 1$?
  - $n - 1$ (Why?)
- Average is

  $$(n+[n+(n-1)+(n-2)+....+1])/m = n/m+n(n+1)/2m \approx n/4$$
- Even though though the average cluster length is 2, the cost is about $n/4$ probes.

COSTS FOR LINEAR PROBING

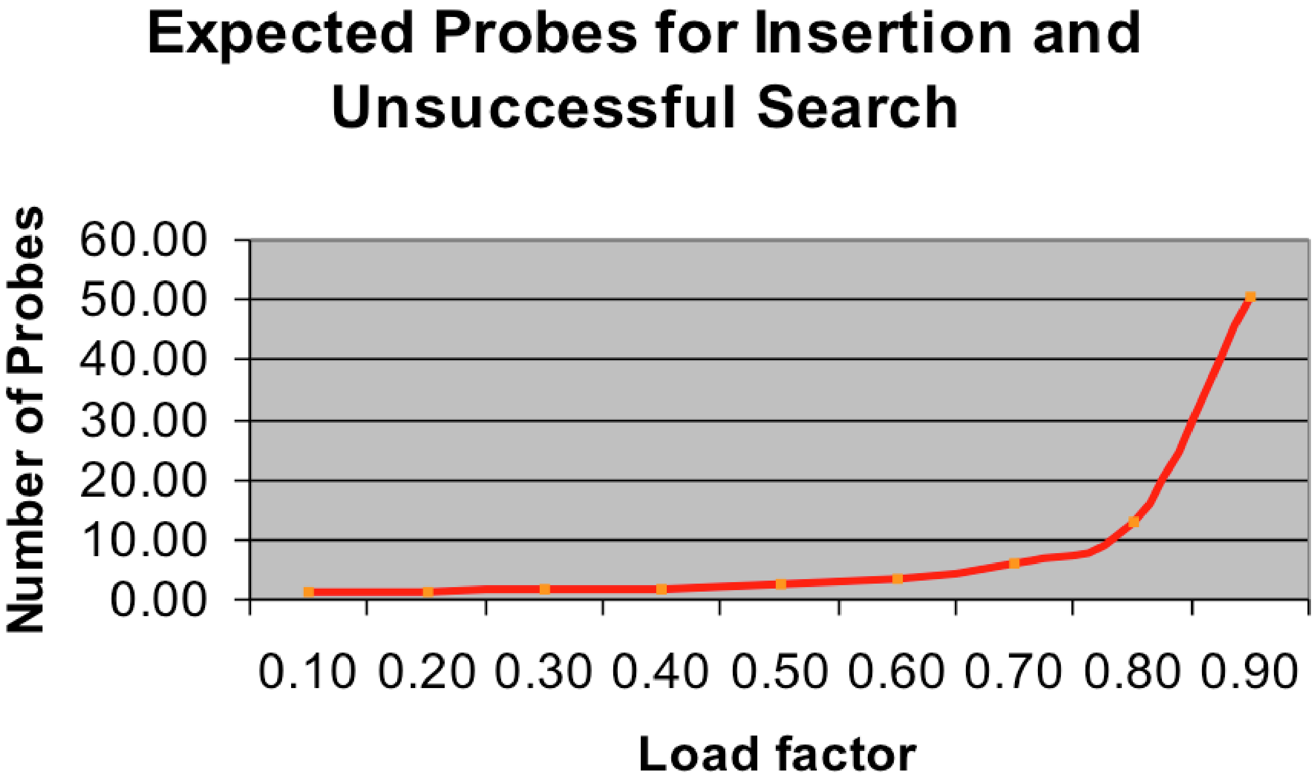
- Given a hash table of size m and with $n = \lambda m$ keys.
- The cost of an unsuccessful search/insert is

$$\frac{1}{2}\left(1 + \frac{1}{1-\lambda^2}\right)$$

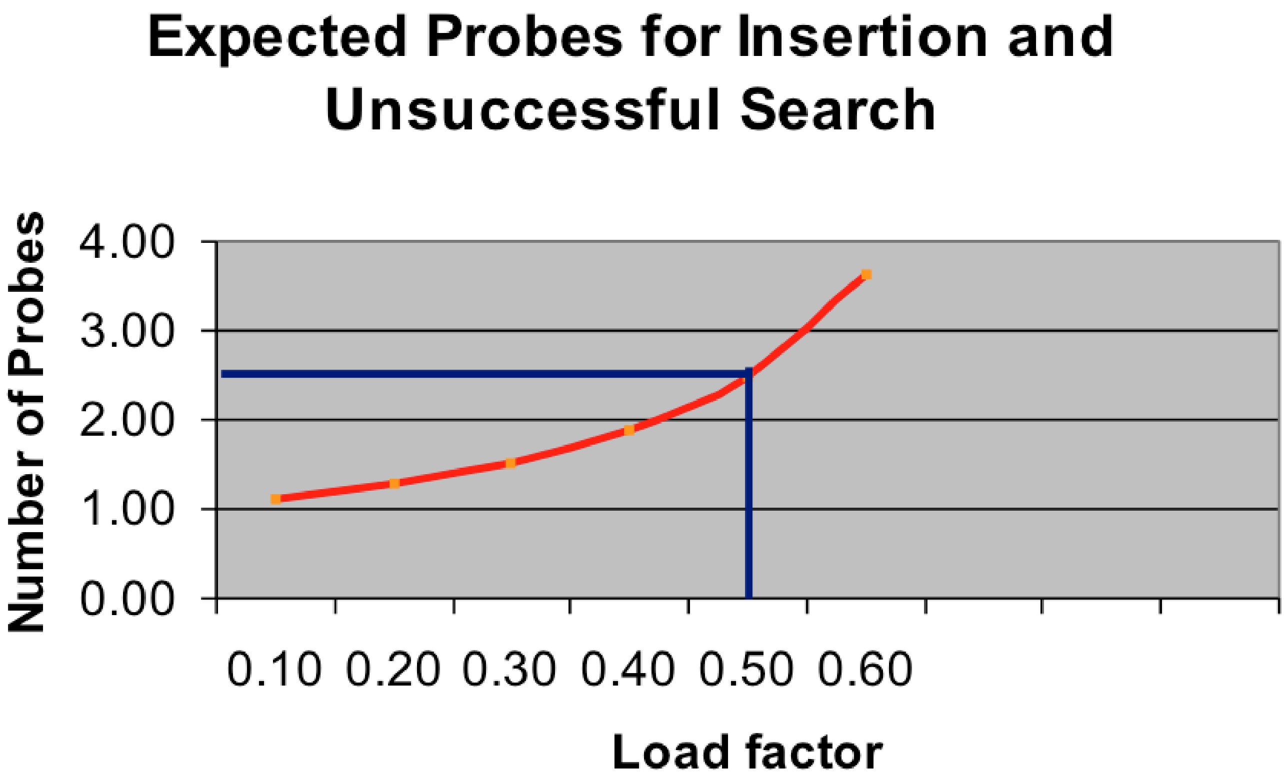
- The cost of an successful search is

$$\frac{1}{2}\left(1 + \frac{1}{1-\lambda}\right).$$

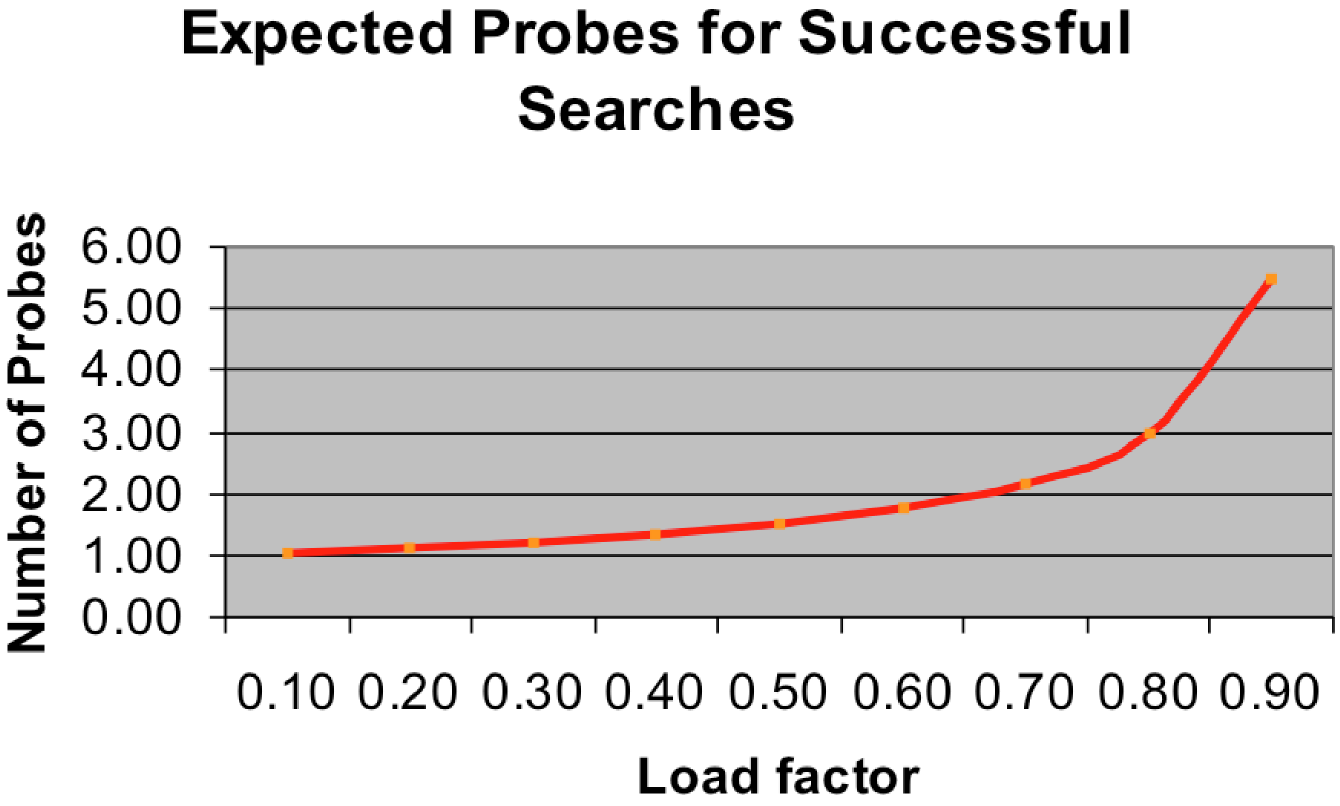
COSTS FOR LINEAR PROBING



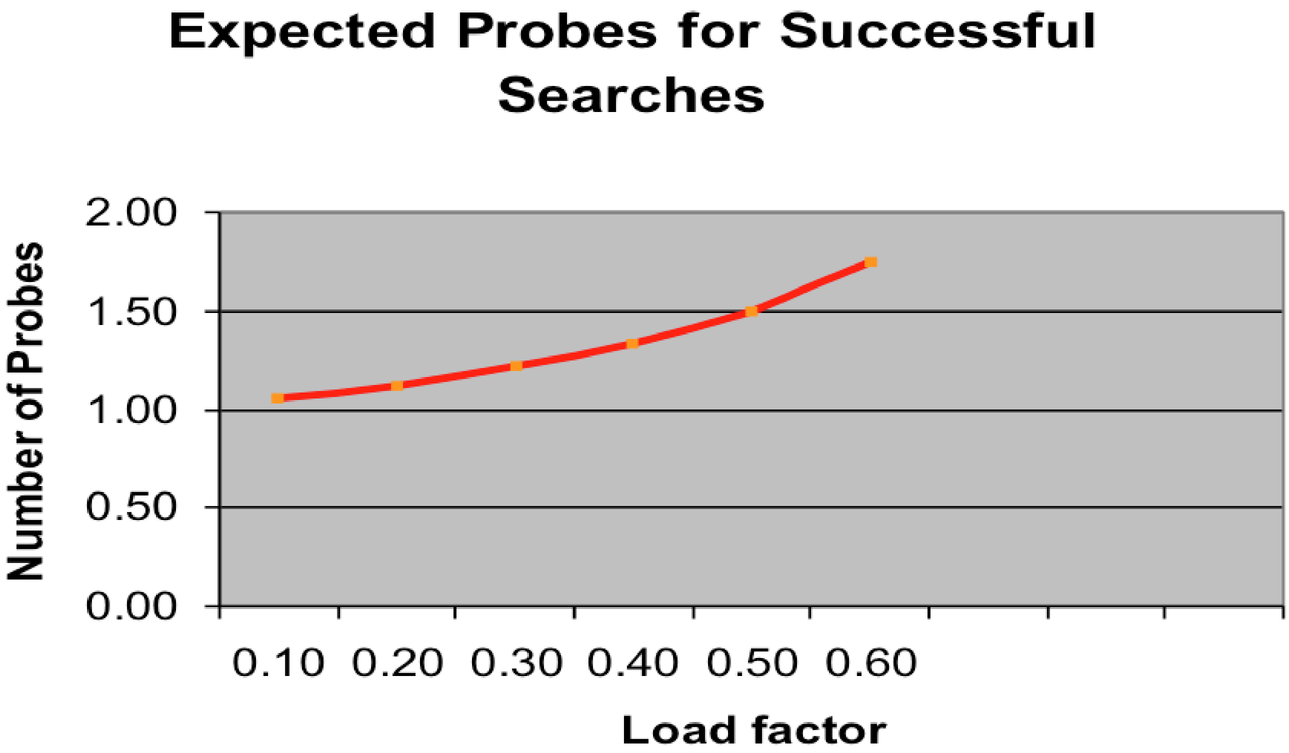
COSTS FOR LINEAR PROBING



COSTS FOR LINEAR PROBING



COSTS FOR LINEAR PROBING



QUADRATIC PROBING

- We check cell at $h(k, i) = (h(k) + i^2) \mod m$ in i^{th} probe.
- Makes longer jumps
- Avoids primary clustering
- But has *secondary clustering*.
- Since there are m possible positions there are m probe sequences.
- Not all available cells get probed (Why?)

QUADRATIC PROBING

- If m is prime and the table is at least half empty, then quadratic probing will always find an empty location.
- Furthermore, no locations are checked twice.

QUADRATIC PROBING

- Consider two probe locations $h(k) + i^2$ and $h(k) + j^2, 0 \leq i, j < \lceil m/2 \rceil$.
- Suppose the locations are the same but $i \neq j$.

$$\begin{aligned} h(k) + i^2 &\equiv (h(k) + j^2) \mod m \\ i^2 &\equiv j^2 \mod m \\ i^2 - j^2 &\equiv 0 \mod m \\ (i - j)(i + j) &\equiv 0 \mod m \end{aligned}$$

- Therefore, either $i - j$ or $i + j$ are divisible by m.
- But since both $i - j$ and $i + j$ are less than m and m is prime, they cannot be divisible by m.
- Thus the first $\lceil m/2 \rceil$ probes are distinct and guaranteed to find an empty location.

QUADRATIC PROBING

- Computing the next hash value is only slightly more expensive

$$\begin{aligned} h_i - h_{i-1} &\equiv (i^2 - (i-1)^2) \mod m \\ h_i &\equiv (h_{i-1} + 2i - 1) \mod m \end{aligned}$$

- If the table gets too full, one can resize and rehash
  - Constant additional overhead

DOUBLE HASHING

- Uses two hash-functions:
  - initial location
  - size of the jump
- i^{th} probe is

$$h(k,i) = (h_1(k) + i \cdot h_2(k)) \mod m.$$

- Different keys are likely to have different values jump function if they collide.
- Avoids secondary clustering
- $h_2(k)$ should be relatively prime to m to probe each locations.
  - m prime and $0 < h_2(k) < m$ is one option.

DOUBLE HASHING

- The average number of probes for an unsuccessful search or an insert is at most

$$1 + \lambda + \lambda^2 + \ldots = \left(\frac{1}{1-\lambda}\right)$$

  - Why?
- The average number of probes for a successful search is

$$\frac{1}{\lambda}\left(1 + \ln\left(\frac{1}{1-\lambda}\right)\right).$$

  - Same argument of averaging over probes at insertion time.

DOUBLE HASHING

λ	1/4	1/2	2/3	3/4	9/10
successful	1.2	1.4	1.6	1.8	2.6
unsuccessful	1.3	1.5	2.0	3.0	5.5

- Allows for smaller tables than linear or quadratic probing
- Higher cost for hash function

Parallel Hashing

- $\mathtt{injectCond}(IV, S) : (int \times \alpha)\mathtt{seq} \times (\alpha\,\mathtt{option})seq \rightarrow (\alpha\,\mathtt{option})seq$.
- Conditionally writes each value v_j into location i_j of S
  - if the location is set to NONE

```
fun insert(T, K) =
let
   fun insert'(T, K, i) =
      if |K| = 0 then T
      else let
         val T' = injectCond({(h(k, i), k) : k ∈ K}, T)
         val K' = {k : k ∈ K | T[h(k, i)] ≠ k}
      in
         insert'(T', K', i + 1)      end
in
   insert'(T, k, 1)
end
```

15-210
PARALLEL AND SEQUENTIAL ALGORITHMS AND DATA STRUCTURES

LECTURE 27

PRIORITY QUEUES

SYNOPSIS

- Priority Queues
- Heaps
- Meldable Priority Queues
- Leftist Heaps

PRIORITY QUEUES

- Abstract Data Type supporting
  - `deleteMin/deleteMax`
  - `insert`
- Used in many useful algorithms
  - Dijkstra' Algorithm
  - Prim's Algorithm for MST
  - Constructing Huffman Codes
  - *Heapsort*

HEAPSORT

```
fun sort S =
let
    val pq = iter Q.insert Q.empty S
    fun sort' pq =
    let
        case (PQ.deleteMin pq) of
            NONE ⇒ []
          | SOME(v, pq') ⇒ v :: sort'(pq')
in
    Seq.fromList(sort' pq)
end
```

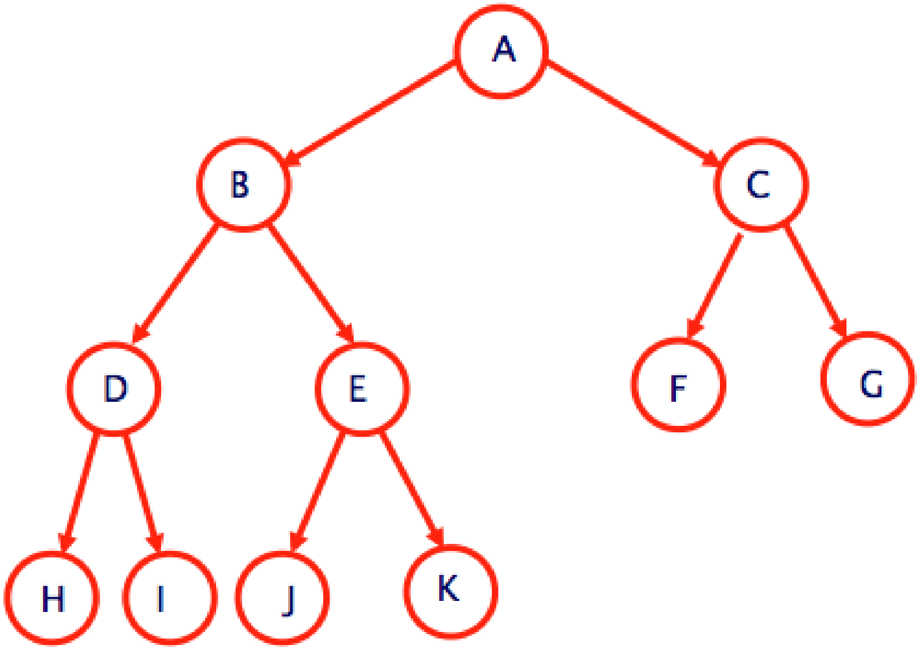
UNDERLYING IMPLEMENTATIONS

- **Sorted and Unsorted Lists/Arrays**
  - One of `deleteMin` and `insert` is fast ($O(1)$)
  - The other is slow. $O(n)$
- **Balanced binary search trees**
  - Both operations have $O(\log n)$ work and span.
- **Binary heaps**
  - Both operations have $O(\log n)$ work and span.
  - But binary heaps provide a $O(1)$ work `findMin` operation.

HEAPS

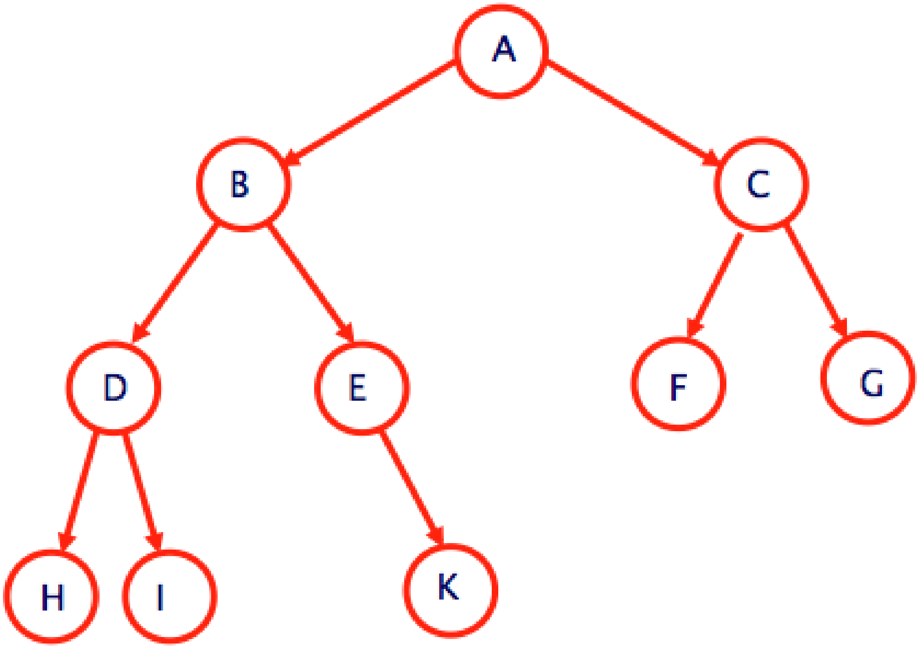
- A *min-heap* (*max-heap*) is a rooted tree
- Key at every node is $\leq$ ($\geq$) all descendants.
- A *binary heap* is heap which has
    - *Shape property*: The tree is a complete binary tree
        - All levels of the tree are completely filled except the bottom level, which is filled from the left
    - *Heap Property*

BINARY HEAPS



A complete tree

BINARY HEAPS

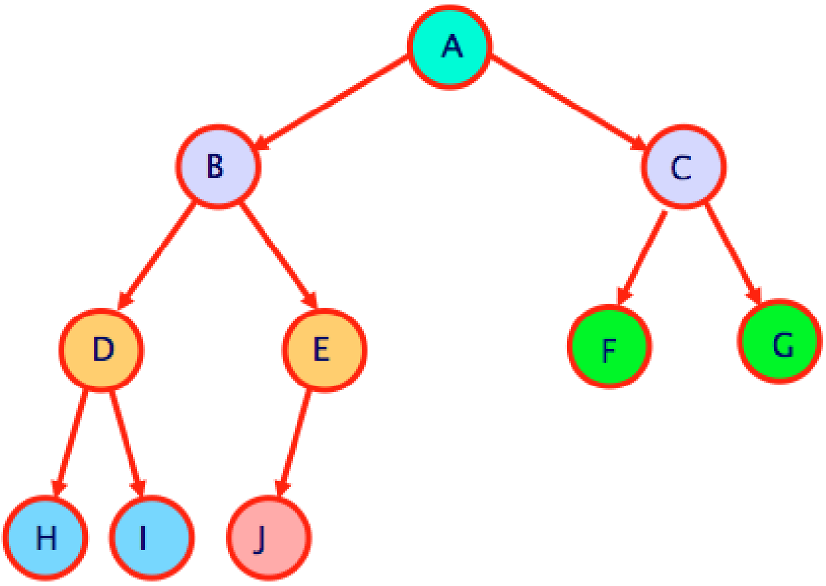


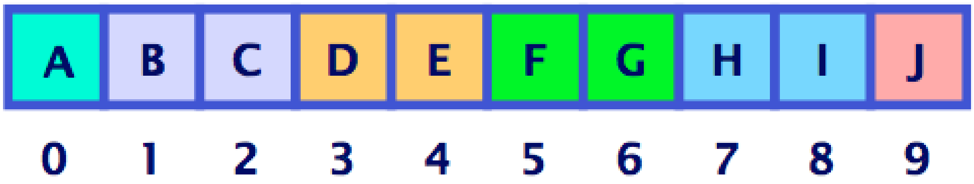
An incomplete tree

BINARY HEAPS

- Shape Property ⇒ binary heap can be maintained in an array.
- Index of a parent or a child is very easy to compute
- Operations first restore shape property, then heap property.

BINARY HEAPS AND ARRAYS





A	B	C	D	E	F	G	H	I	J
0	1	2	3	4	5	6	7	8	9

BUILDING PRIORITY QUEUES

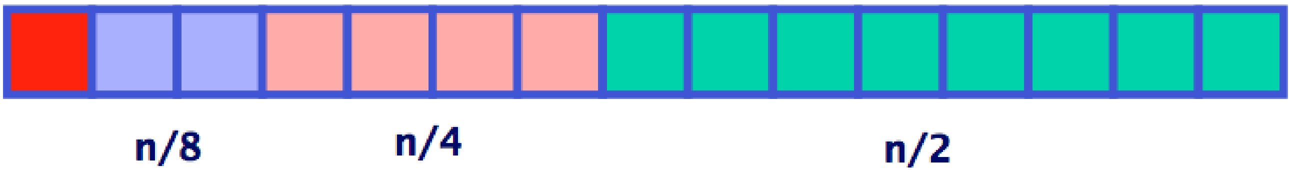
- We can insert elements one-by-one
    - With balanced binary trees and binary heaps, work is $O(n \log n)$
    - Can we do better?
- Build the heap recursively
    - If left and right sides are already heaps, just *shift down* the root element.

Building Heaps Directly

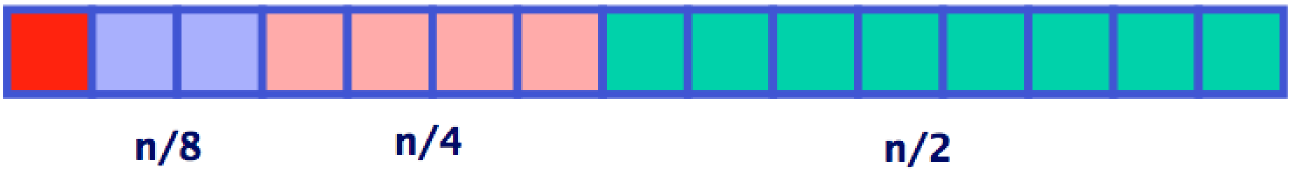
```
fun $sequentialFromSeq S =$
let
    fun $heapify(S, i) =$
        if $(i >= |S|/2)$ then $S$
        else let
                val $S' = heapify(S, 2 * i + 1)$
                val $S'' = heapify(S', 2 * i + 2)$
                in $shiftDown(S'', i)$ end
in $heapify(S, 0)$ end
```

COST ANALYSIS

- `shiftDown` does $O(\log n)$ work on subtree of size n
- $W(n) = 2W(n/2) + O(\log n) \in O(n)$
- Opportunities for parallelism?

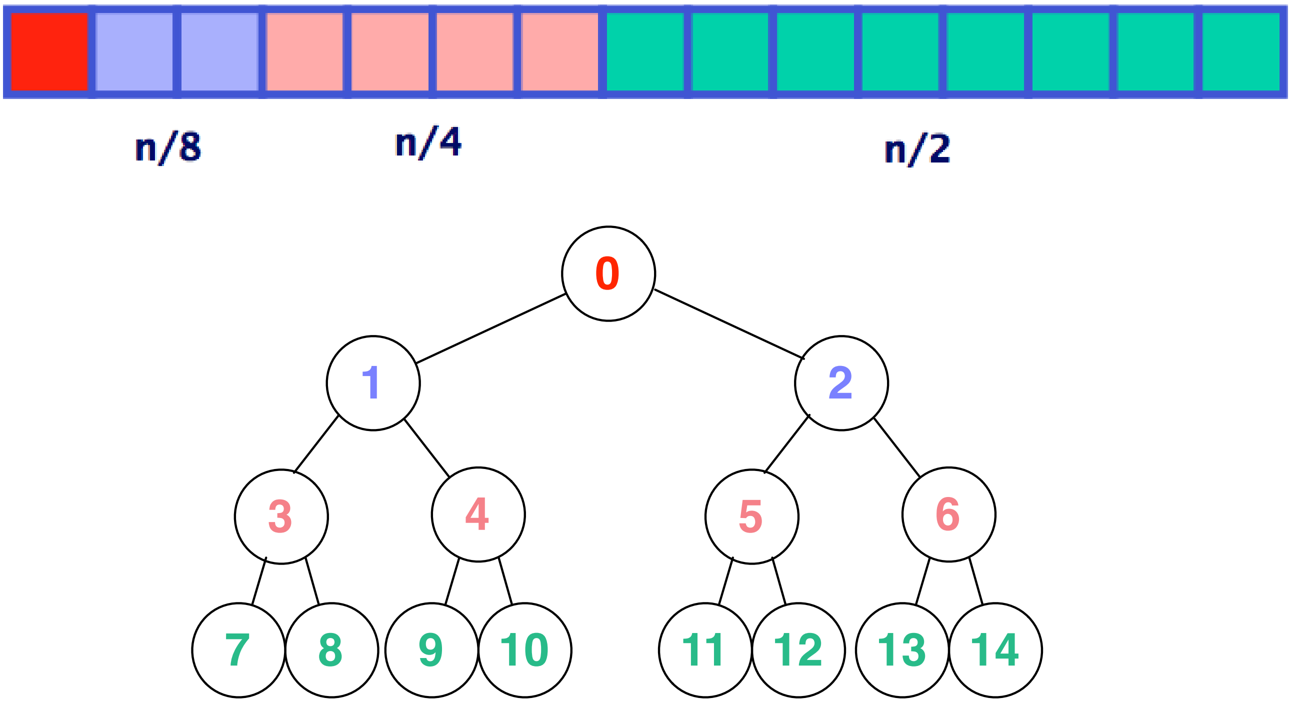


PARALLEL HEAPIFY



- Green cells are OK
- All the pinks cells can be shifted down in parallel
- Then all purple cells can be shifted down in parallel
- (All) Red cell(s) can be shifted down in parallel

PARALLEL HEAPIFY



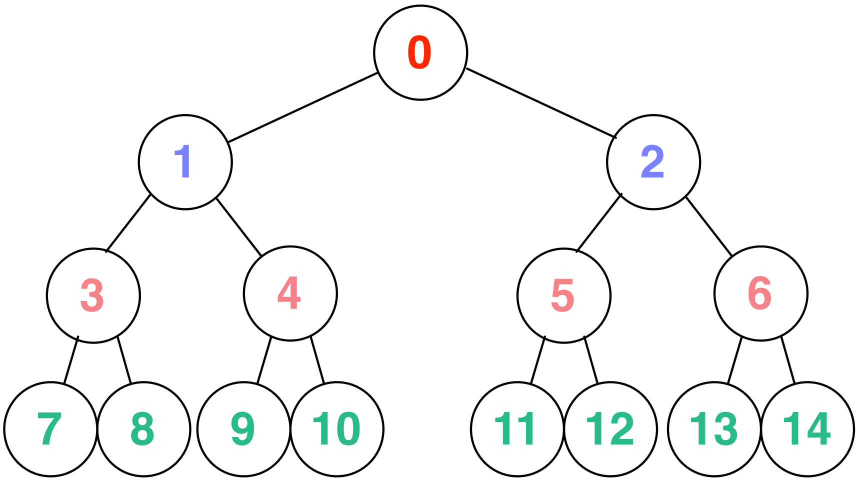
PARALLEL HEAPIFY

- We use Single-threaded sequences

```
fun fromSeq $S$ : 'a seq =
let
  fun heapify ($S$, $d$) =
  let
    val $S' = $ shiftDown ($S$, $\langle 2^d - 1, \ldots, 2^{d+1} - 2 \rangle$, $d$)
  in
    if ($d = 0$) then $S'$
    else heapify ($S'$, $d - 1$)
in heapify ($S$, $\lfloor \log_2 n \rfloor - 1$) end
```

- $S(n) = S(n/2) + O(\log n) \in O(\log^2 n)$

PARALLEL HEAPIFY



- $d = 2 \Rightarrow$ `shiftDown` (S, <3, 4, 5, 6>, 2)
- $d = 1 \Rightarrow$ `shiftDown` (S, <1, 2>, 1)
- $d = 0 \Rightarrow$ `shiftDown` (S, <0>, 0)

PRIORITY QUEUES – SUMMARY

Data. Str.	`findMin`	`deleteMin`	`insert`	`fromSeq`
sorted linked list	$O(1)$	$O(1)$	$O(n)$	$O(n \log n)$
unsorted linked list	$O(n)$	$O(n)$	$O(1)$	$O(n)$
balanced search tree	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(n \log n)$
binary heap	$O(1)$	$O(\log n)$	$O(\log n)$	$O(n)$

MELDABLE PRIORITY QUEUES

- Priority Queues with an additional *meld* operation
  - Just like the union in BSTs
  - Takes two meldable PQs and returns the union as a meldable PQ
- Implementations uses *leftist heaps*
  - Same work and span as binary heaps for insert, deletemin
  - Meld has $O(\log n + \log m)$ work and span where m and n are the heap sizes

MIN HEAPS

- Binary tree
- Maintains the heap property
- But does *not* maintain the complete binary tree property
- Here is an example

```
       o 3
      / \
   7 o   o 8
    / \
11 o   o 15
      / \
  22 o   o 16
```

MIN HEAPS

- To implement `deleteMin`
  - Remove the root

```
     7 o                 o 8
      / \
  11 o   o 15
        / \
    22 o   o 16
```

- We can then use `meld` to union the heaps.

MIN HEAPS

- To implement `insert`
    - We create a single node heap
    - `meld` it with the original heap
- `fromSeq` is also easy using reduce

```
val pq = Seq.reduce Q.meld Q.empty
                   (Seq.map Q.singleton S)
```

THE MELD OPERATION

- So we only need the `meld` operation
- Consider

```
     4 o             o 3
      / \           / \
  11 o   o 7     8 o   o 5
    / \           /
19 o   o 23   14 o
```

- Which element goes to the root?

THE MELD OPERATION

- Select the tree with the smaller root and recursively `meld` with one of its children

```
       o 3
      / \
   8 o   = meld ( 4 o  ,  o 5 )
    /              / \
14 o           11 o   o 7
                 / \
             19 o   o 23
```

THE MELD OPERATION

- Applying recursively

```
       o 3
      / \
   8 o    o 4
    /    / \
14 o  11 o  = meld ( o 7  o 5)
        / \
    19 o   o 23
```

THE MELD OPERATION

- Applying recursively

```
       o 3
      / \
   8 o     o 4
    /    /  \
14 o  11 o    o 5
        / \    \
    19 o   o 23  = meld( o 7 ,empty)
```

- Melding *A* with an empty heap gives *A*

THE MELD OPERATION

```
**datatype** $PQ = \texttt{Leaf} \mid \texttt{Node}$ **of** $(\texttt{key} \times PQ \times PQ)$
**fun** $\texttt{meld}(A, B) =$
  **case** $(A, B)$ **of**
      $(\_, \texttt{Leaf}) \Rightarrow A$
    $\mid (\texttt{Leaf}, \_) \Rightarrow B$
    $\mid (\texttt{Node}(k_a, L_a, R_a), \texttt{Node}(k_b, L_b, R_b)) \Rightarrow$
        **case** $\texttt{Key.compare}\ (k_a, k_b)$ **of**
            $\texttt{LESS} \Rightarrow \texttt{Node}(k_a, L_a, \texttt{meld}(R_a, B))$
          $\mid \_ \Rightarrow \texttt{Node}(k_b, L_b, \texttt{meld}(A, R_b))$
```

- Traverses the right spines of the trees
- Could be $\Theta(|A| + |B|)$ in the worst case.

LEFTIST HEAPS

- When melding, keep trees *deeper* on the left.
- Define

$$\text{rank}(x) = \#\text{ of nodes on the right}$$
$$\text{spine of the subtree rooted at } x,$$

- For all nodes, rank can be inductively defined

$$\begin{aligned}\text{rank}(\text{leaf}) &= 0\\ \text{rank}(\text{node}(\_, \_, R)) &= 1 + \text{rank}(R)\end{aligned}$$

Leftist Property

- For all node x in a leftist heap,

$$\text{rank}(L(x)) \geq \text{rank}(R(x))$$

    - $L(x)$ and $R(x)$ are the left and child children of x
- Allows for

```
          o 1
         /
        o 2
       /
      o 3
     .
    .
  o n
```

- But this is OK (Why?)

LEFTIST HEAPS

- Most items pile to the left
- Right spine is relatively short!

LEMMA

In a leftist heap with n entries, the rank of the root node is at most $\log_2(n+1)$.

LEFTIST HEAPS

```
datatype PQ = Leaf | Node of (int × key × PQ × PQ)
fun rank Leaf = 0
  | rank (Node(r, _, _, _)) = r
fun makeLeftistNode (v, L, R) =
    if (rank(L) < rank(R))
    then Node(1 + rank(L), v, R, L)
    else Node(1 + rank(R), v, L, R)
```

- Puts lower rank subtree to the right!

Leftist Heaps

```
fun meld (A, B) =
  case (A, B) of
      (_, Leaf) ⇒ A
    | (Leaf, _) ⇒ B
    | (Node(_, $k_a$, $L_a$, $R_a$), Node(_, $k_b$, $L_b$, $R_b$)) ⇒
        case Key.compare($k_a$, $k_b$) of
            LESS ⇒ makeLeftistNode ($k_a$, $L_a$, meld($R_a$, B))
          | _ ⇒ makeLeftistNode ($k_b$, $L_b$, meld(A, $R_b$))
```

LEFTIST HEAPS

THEOREM

If A and B are leftists heaps then

- the `meld`(A, B) algorithm runs in $O(\log(|A|) + \log(|B|))$ work, and
- returns a leftist heap containing the union of A and B.

- Code traverses the right spines, one node at a time
  - so needs at most rank(A) + rank(B) steps
  - Each step needs constant work
- `makeLeftistNode` guarantees leftist result

PROVING THE LEMMA

CLAIM

If a heap has rank r, it contains at least $2^r - 1$ entries.

- $n(r) \equiv$ nodes in the smallest heap of rank r
  - Monotone: if $r' \geq r$, then $n(r') \geq n(r)$
  - $n(0) = 0$
- $\text{rank}(L(x)) \geq \text{rank}(R(x)) = r - 1$

$$\begin{aligned} n(r) &= 1 + n(\text{rank}(L(x))) + n(\text{rank}(R(x))) \\ &\geq 1 + n(r-1) + n(r-1) = 1 + 2 \cdot n(r-1). \end{aligned}$$

- $n(r) \geq 2^r - 1$

PROVING THE LEMMA

- Apply the claim
- Suppose leftist heap of n nodes has rank r
- $n \geq n(r) \geq 2^r - 1$
- $2^r \leq n + 1 \Rightarrow r \leq \log_2(n+1)$
- Rank of a leftist node of n nodes is at most $\log_2(n+1)$

SUMMARY OF PRIORITY QUEUES

Implementation	`insert`	`findMin`	`deleteMin`	`meld`
(Unsorted) Sequence	$O(n)$	$O(n)$	$O(n)$	$O(m+n)$
Sorted Sequence	$O(n)$	$O(1)$	$O(n)$	$O(m+n)$
Balanced Tree	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(m \log(1+\frac{n}{m}))$
Leftist Heap	$O(\log n)$	$O(1)$	$O(\log n)$	$O(\log m + \log n)$