15-210
Parallel and Sequential Algorithms and Data Structures

Lecture 22

Dynamic Programming

Synopsis

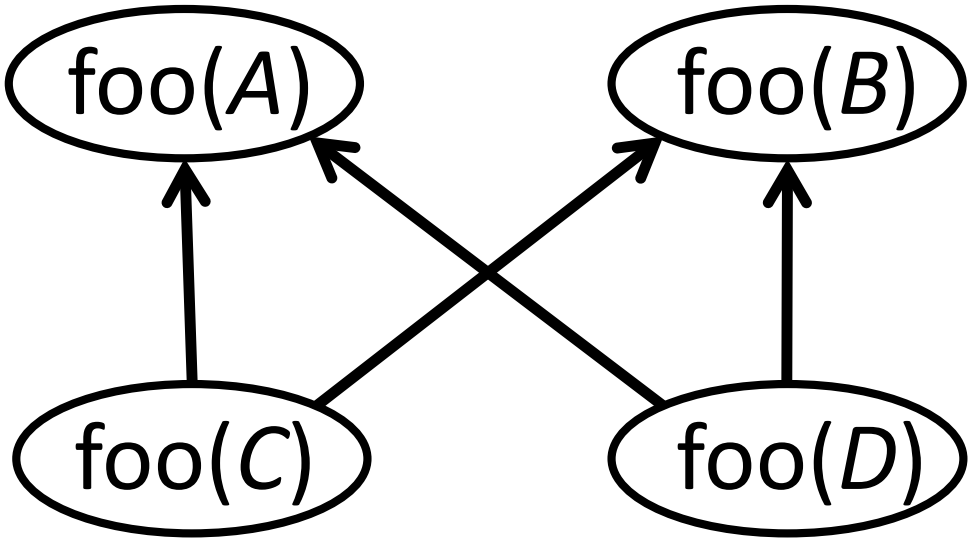
- Dynamic Programming
- Subset Sum Problem
- Minimum Edit Distance Problem
- Additional example applications

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 remember the solutions to the smaller subproblems.

Reusing Solutions

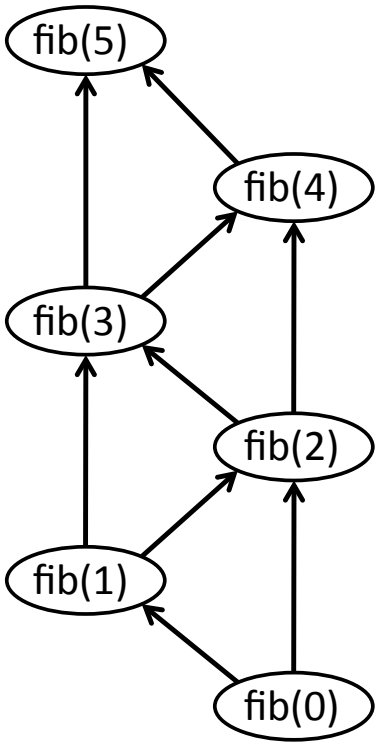
- How much work does this code need?

```
1  fun fib(n) =
2      if (n ≤ 1) then 1
3      else fib(n − 1) + fib(n − 2)
```

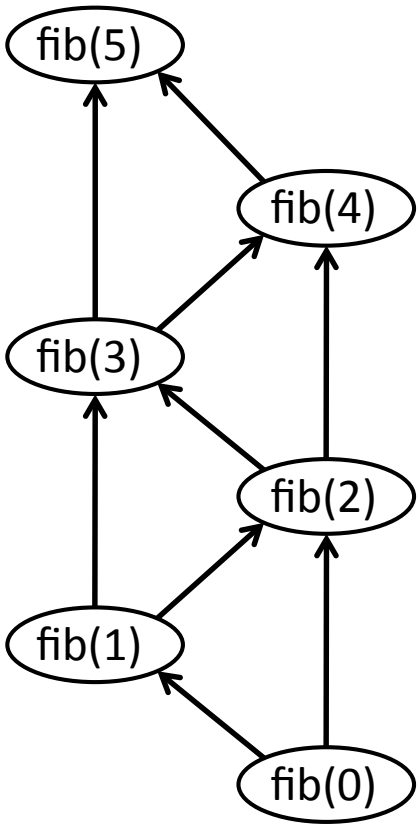
- It turns out $W_{fib}(n) = O(c^n)$ (Why?)

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 DAG 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

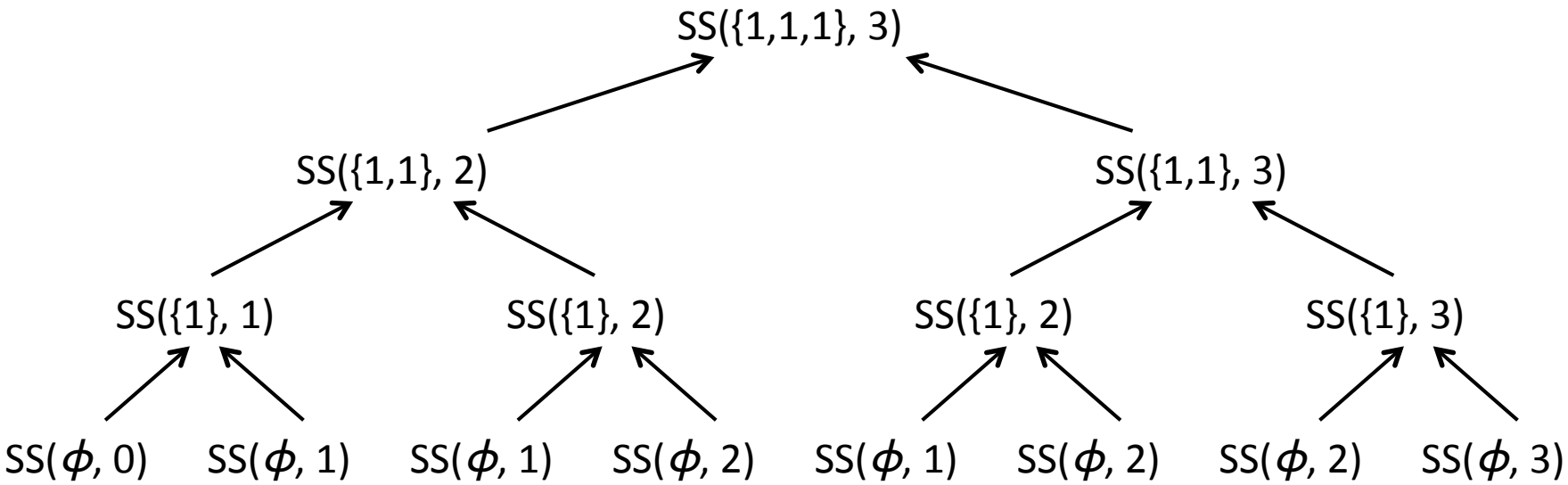
- Brute force: Consider all 2^n subset for a total work of $O(n2^n)$.
- Divide and Conquer: also ends up being exponential work without any sharing!
- Sharing solutions however works.

The Subset Sum Problem

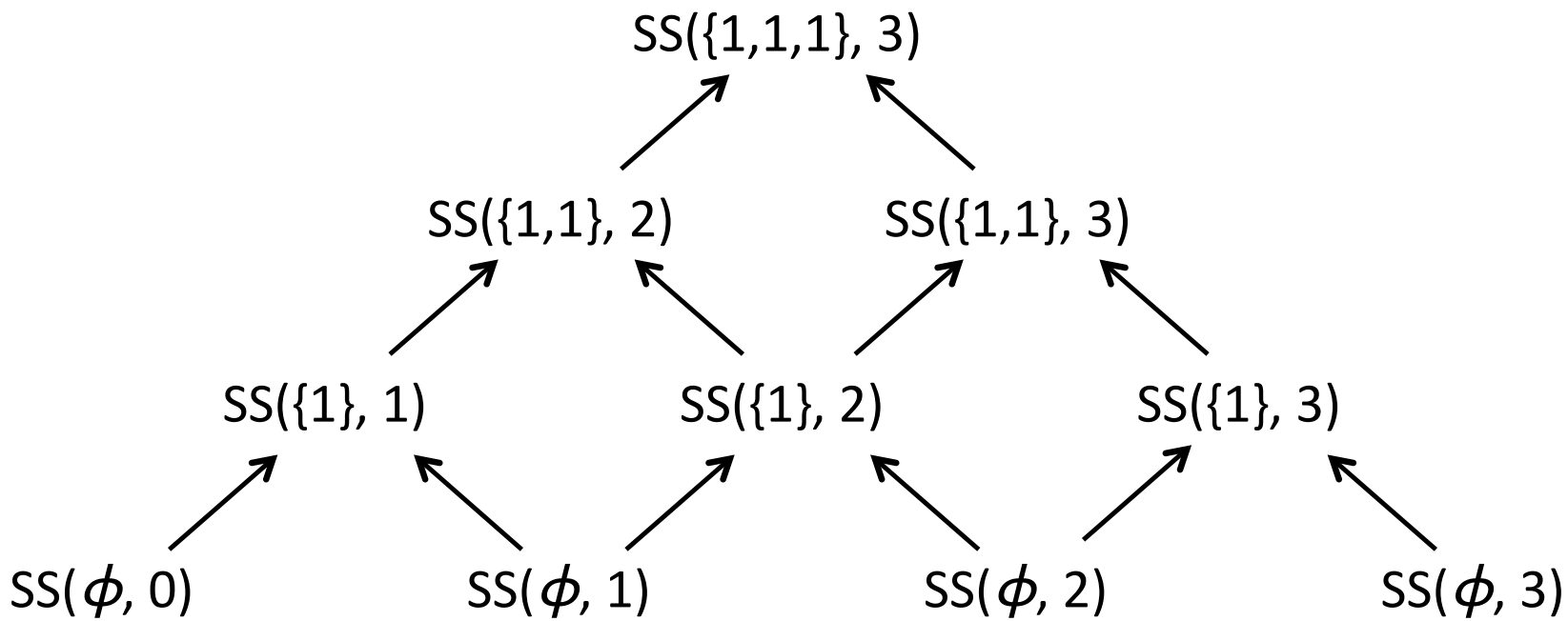
- To solve $SS(S, k)$, pick some element $a \in S$
- Solve (recursively) $SS(S \setminus \{a\}, k - a)$
  - If there is a solution, we are done.
- If not, solve $SS(S \setminus \{a\}, k)$.

```
1  fun SS(S, k) =
2    case (showl(S), k) of
3      (_, 0) ⇒ true
4    | (NIL, _) ⇒ false
5    | (CONS(a, R), _) ⇒
6        if (a > k) then SS(R, k)
7        else (SS(R, k − a) orelse SS(R, k))
```

The Subset Sum Problem DAG



The Subset Sum Problem DAG



- How many distinct subproblems do we need to solve?

The Subset Sum Problem

- For $SS(S, k)$, there are only $|S| + 1$ 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

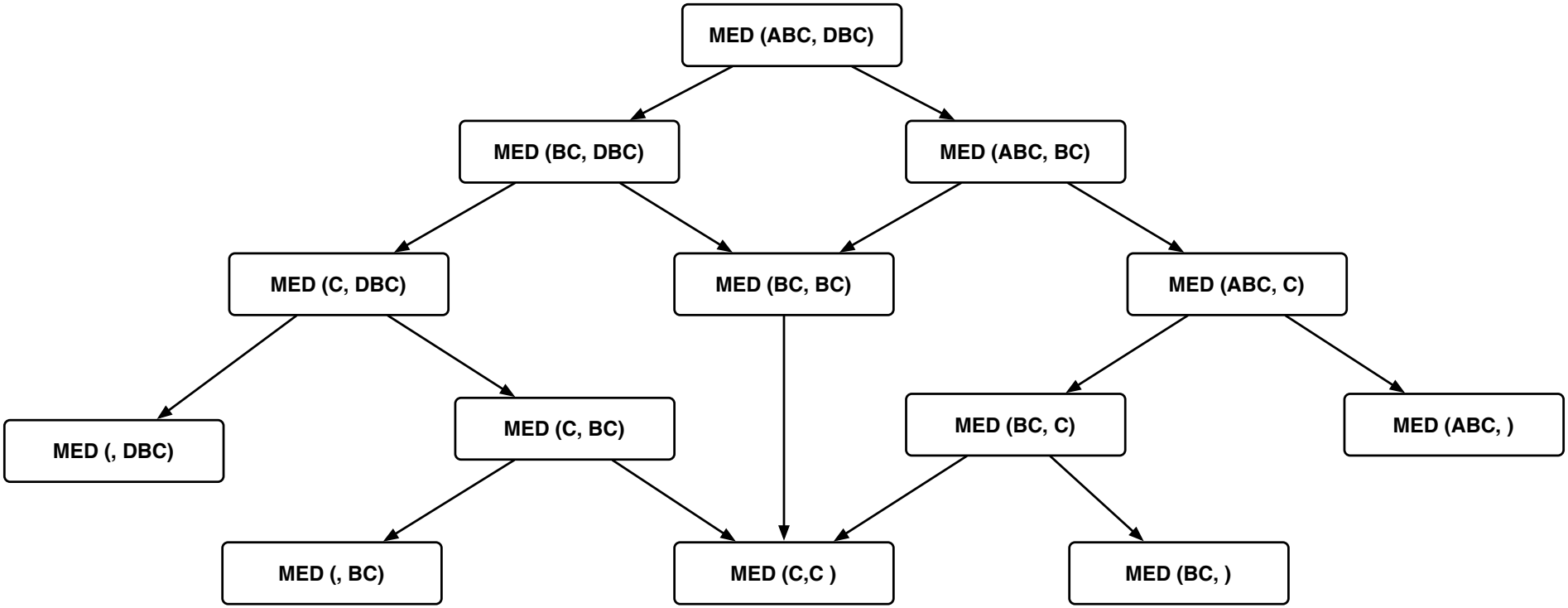
- Given $S = s :: S'$ and $T = t :: T'$
- If $s = t$, $MED(S, T)$ is determined by S' and T'
- Otherwise we have two subproblems:
  - Find $MED(S, T')$ – consider a deletion from T to get T'
  - Find $MED(S', T)$ – consider a deletion to S to get S'
- Find the minimum and add 1.

Minimum Edit Distance

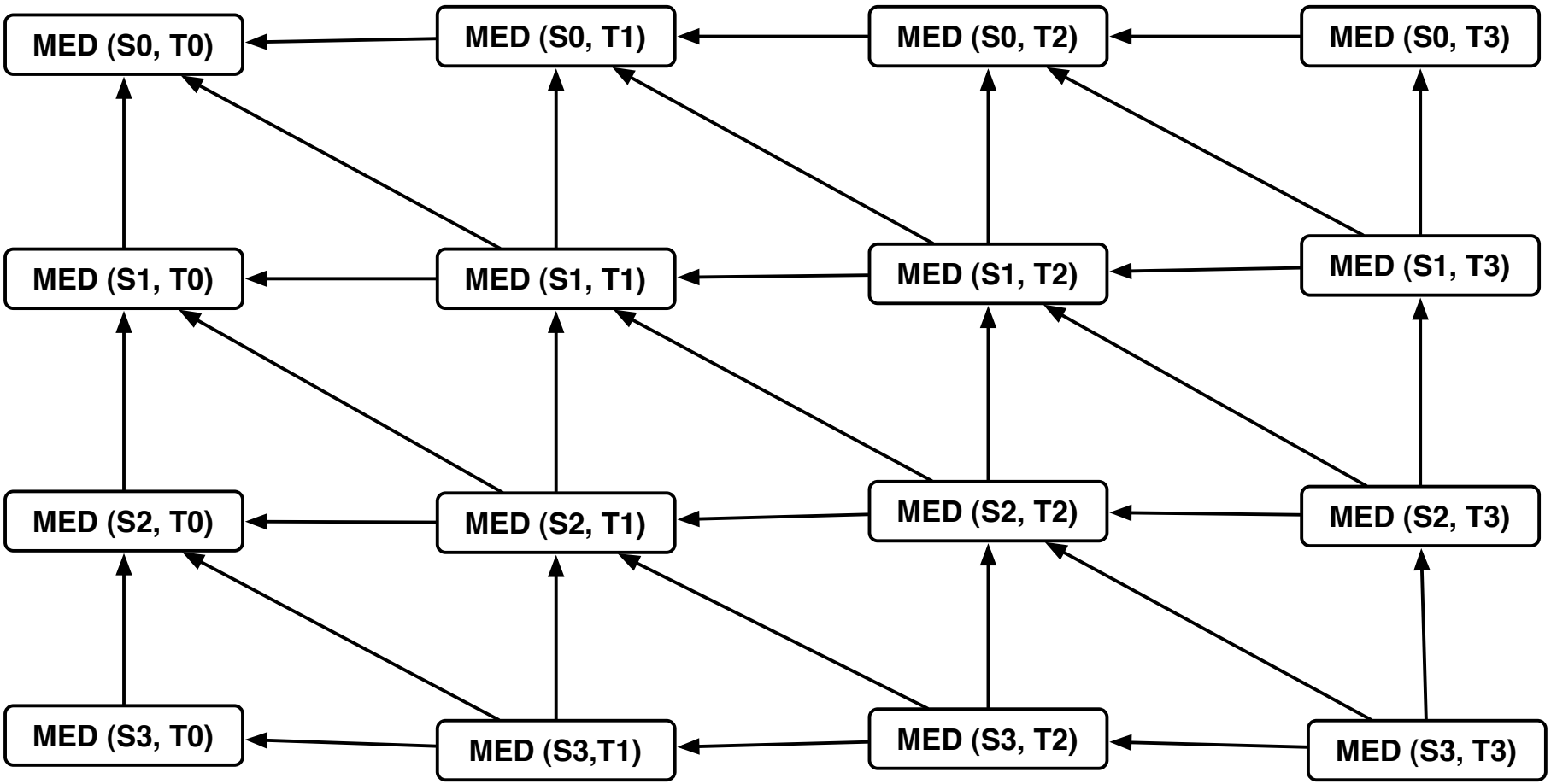
```
1  fun MED(S, T) =
2    case (showl(S), showl(T)) of
3      (_, NIL) ⇒ |S|
4    | (NIL, _) ⇒ |T|
5    | (CONS(s, S′), CONS(t, T′)) ⇒
6        if (s = t) then MED(S′, T′)
7        else 1 + min(MED(S, T′), MED(S′, T))
```

- If run recursively, this would take exponential work.
  - Binary tree with linear depth!
- But there is significant sharing!

Subproblem Sharing in MED



Bottom-up Dependency Graph



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 $C_1, C_2, \ldots C_n = 1$ (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 $i = 1, \ldots \lfloor K/2 \rfloor$
- 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)