15-210
Parallel and Sequential Algorithms and Data Structures

Lecture 23

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

```
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))
```

MED WITH SURROGATES

```
1   fun MED(S, T) = let
2       fun MED'(i, 0) = i
3         | MED'(0, j) = j
4         | MED'(i, j) =case (S_i = T_j) of
5                           true ⇒ MED'(i − 1, j − 1)
6                         | false ⇒ 1 + min(MED'(i, j − 1),
7                                           MED'(i − 1, j))
8   in
9       MED'(|S|, |T|)
10  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!

```
1  fun memo f (M, a) =
2     case find(M, a) of
3        SOME(v) ⇒ (M, v)
4      | NONE ⇒ let
5                 (M′, v) = f(M, a)
6        in
7           (update(M′, a, v), v)
8        end
```

Memoized MED

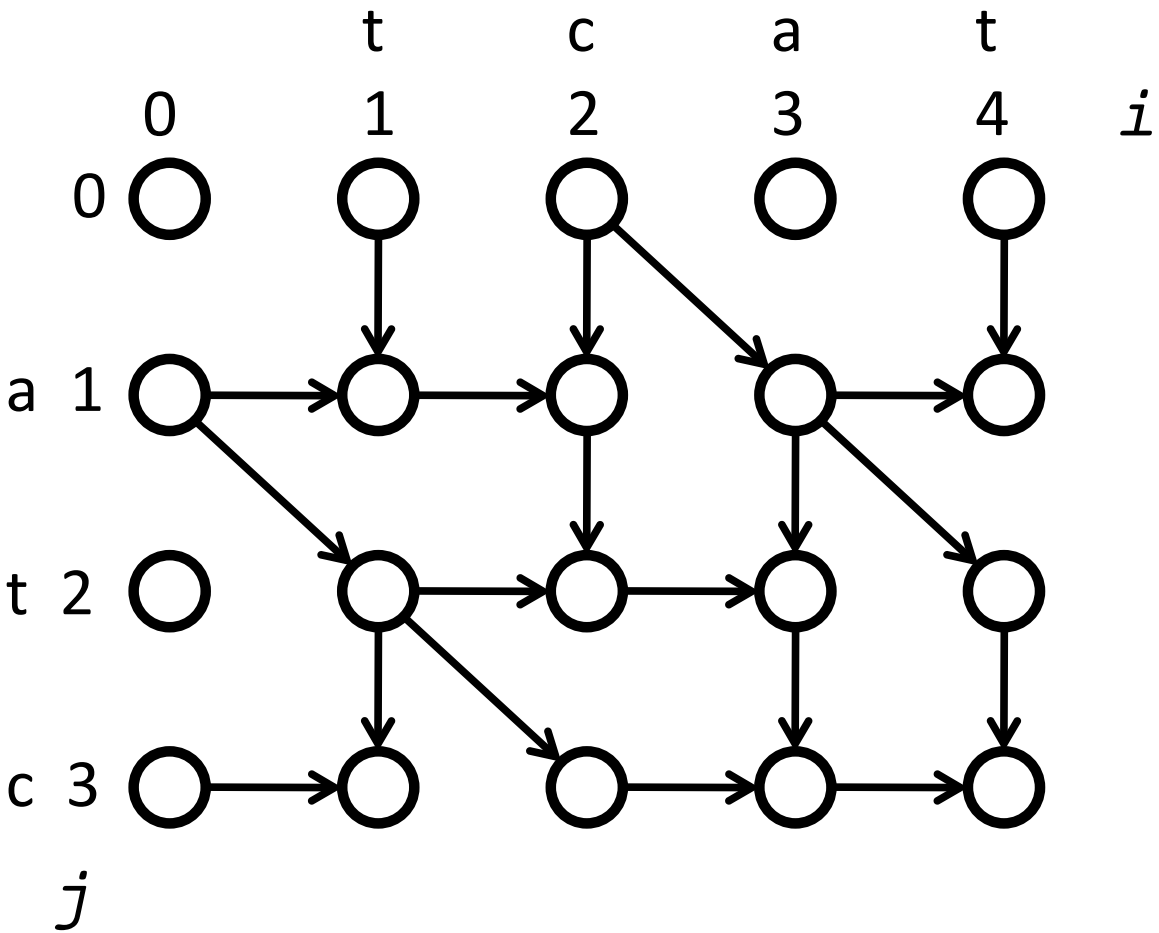
```
1  fun MED(S, T) = let
2      fun MED'(M, (i, 0)) = (M, i)
3        | MED'(M, (0, j)) = (M, j)
4        | MED'(M, (i, j)) =case (S_i = T_j) of
5                              true ⇒ MED''(M, (i − 1, j − 1))
6                            | false ⇒ let
7                                  (M', v_1) = MED''(M, (i, j − 1))
8                                  (M'', v_2) = MED''(M', (i − 1, j))
9                                in (M'', 1 + min(v_1, v_2)) end
10     and MED''(M, (i, j)) = memo MED' (M, (i, j))
11 in
12     MED'({}, (|S|, |T|))
13 end
```

- Purely functional
- but highly sequential

Bottom-up DP

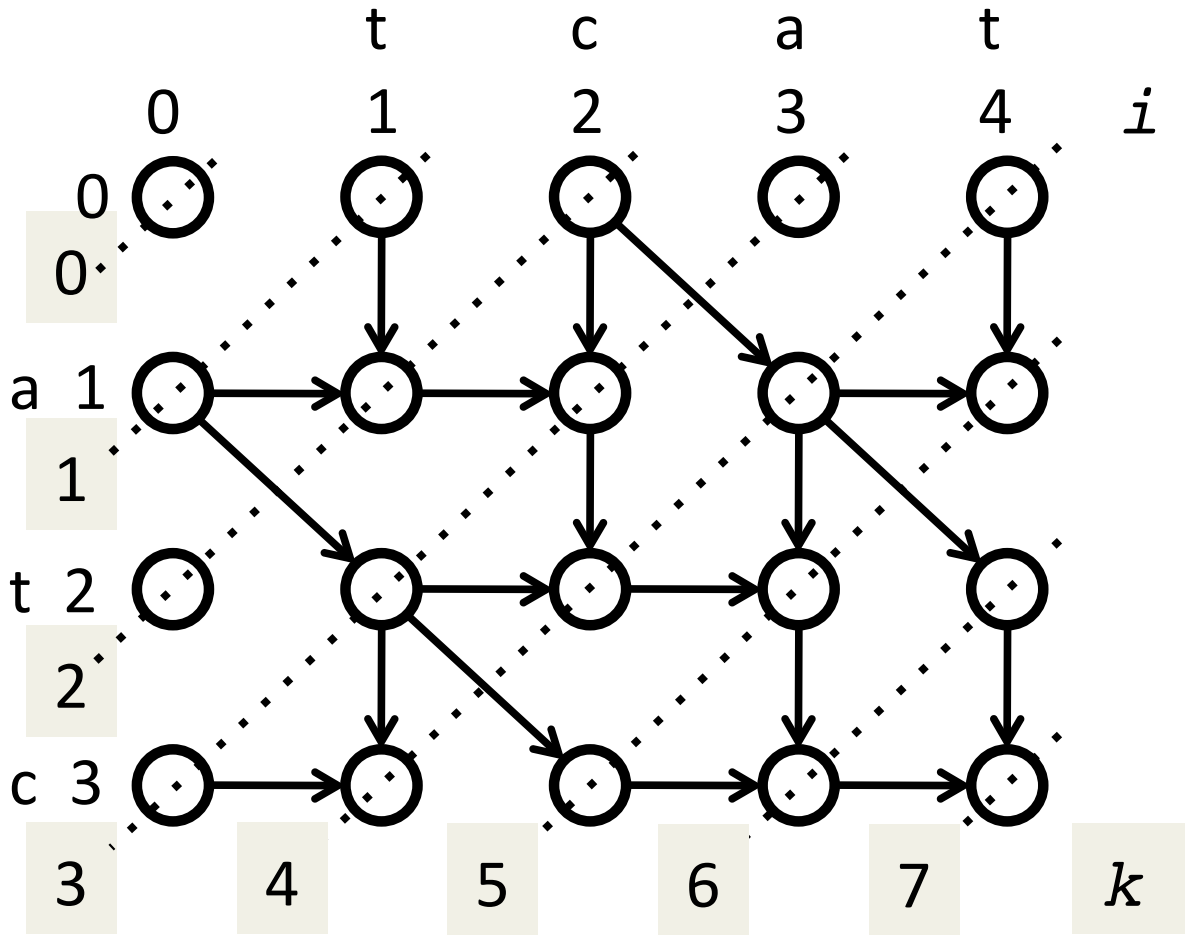
- Start with the leaves
- Works through the subproblems consistent with the DAG
  - if (u, v) is a dependency edge in the DAG, compute u before v, for all such u.
  - All values will be available for v when they are needed!
- Uses a memo table.
- Understanding the DAG structure is important

Bottom-up DP for MED



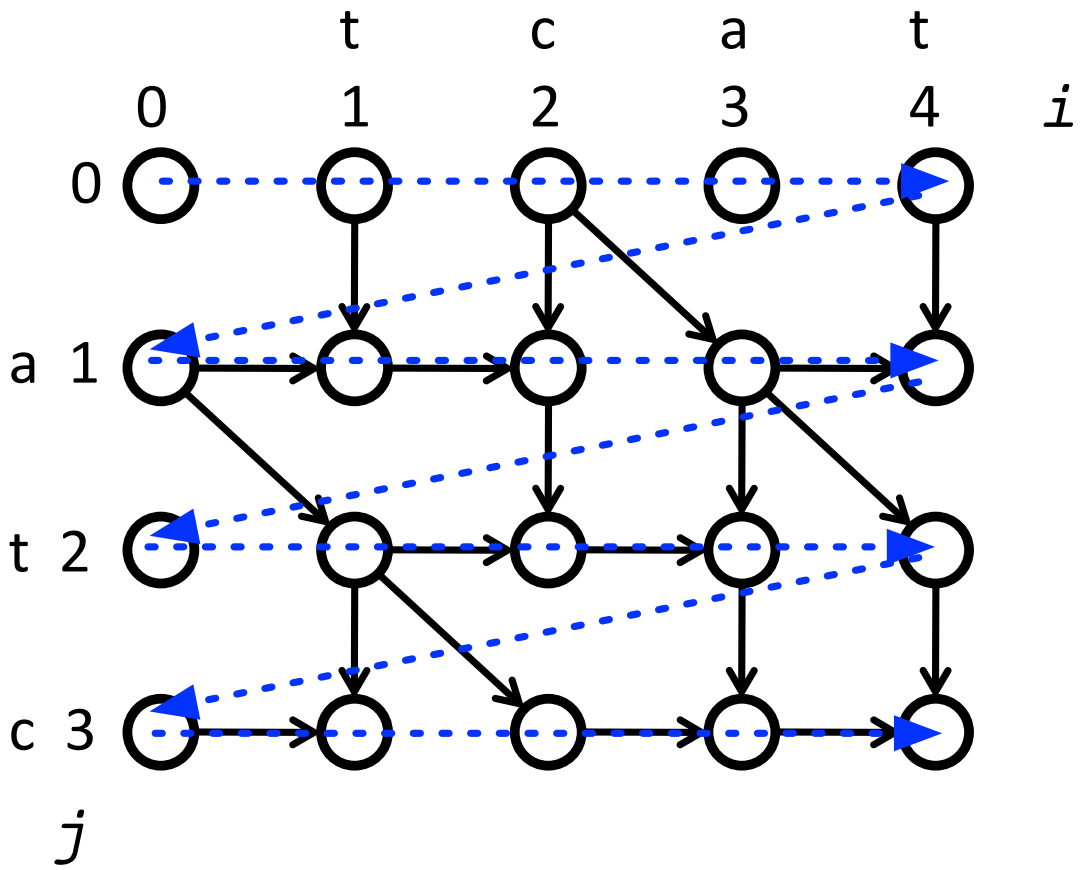
Dag for *MED*(" *tcat*" , " *atc*")

Bottom-up DP for MED



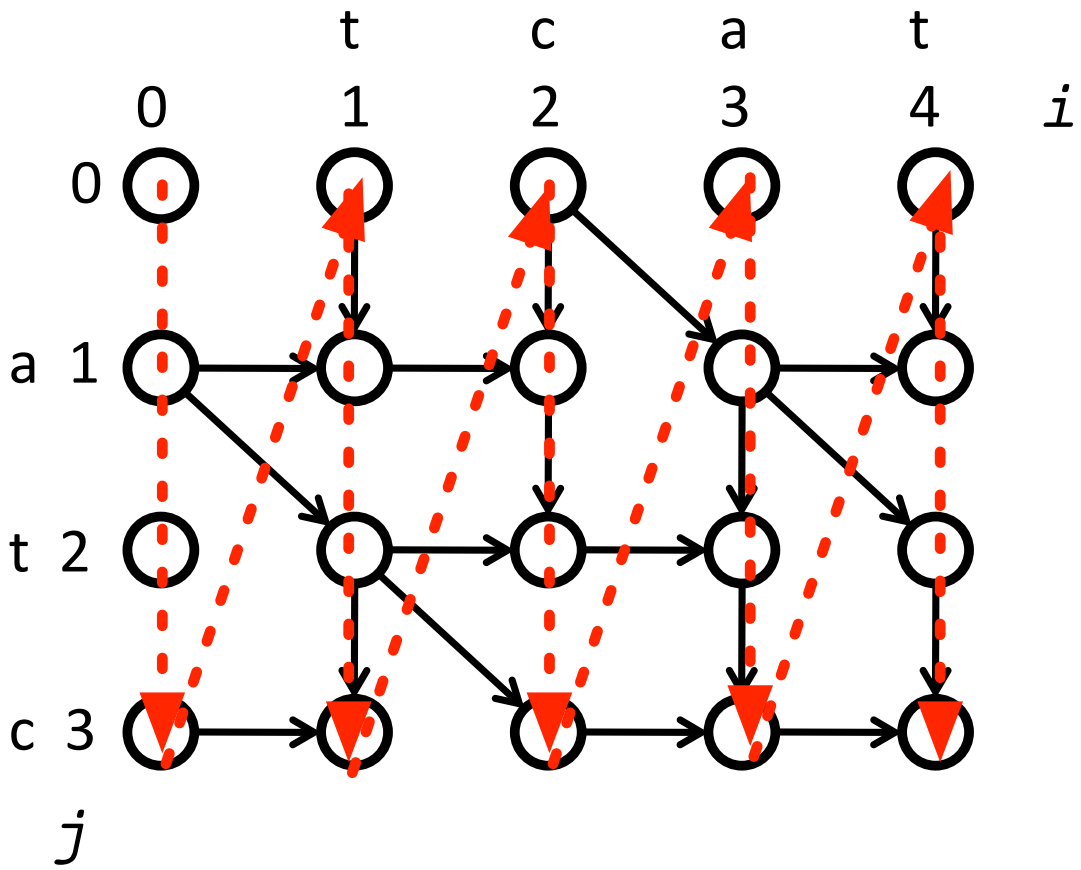
We can go by diagonals.

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

```
1  fun MED(S, T) = let
2    fun MED'(M, (i, 0)) = i
3      | MED'(M, (0, j)) = j
4      | MED'(M, (i, j)) =case (S_i = T_j) of
5                            true ⇒ M_{i-1,j-1}
6                          | false ⇒ 1 + min(M_{i,j-1}, M_{i-1,j})

7    fun diagonals(M, k) =
8      if (k > |S| + |T|) then M
9      else let
10       s = max(0, k − |T|)
11       e = min(k, |S|)
12       M' = M ∪ {(i, k − i) ↦ MED'(M, (i, k − i)) : i ∈ {s, ..., e}}
13     in
14       diagonals(M', k + 1)
15     end

16 in
17   diagonals({}, 0)
18 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

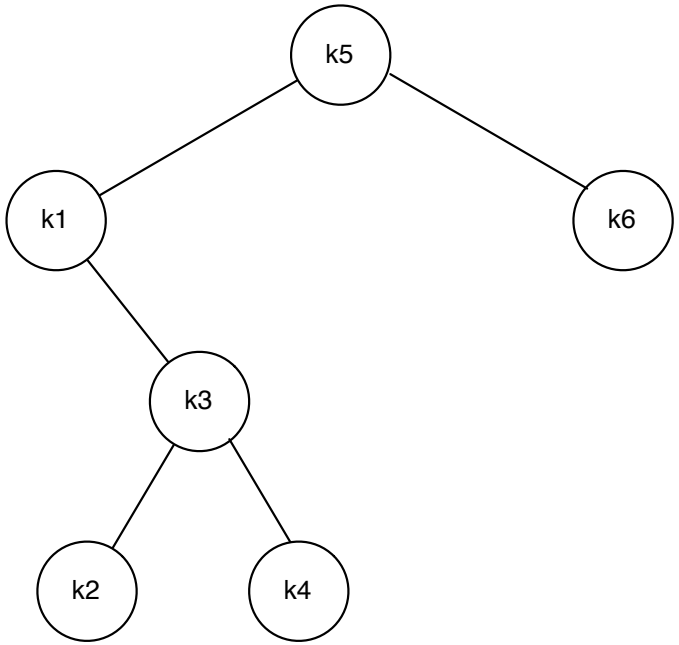
The *optimal binary search tree* (OBST) problem is given an ordered set of keys S and a probability function $p : S \to [0 : 1]$, to find $\hat{T}$

$$\hat{T} = arg\,min_{T \in Trees(S)} \left(\sum_{s \in S} d(s, T) \cdot p(s) \right)$$

where $Trees(S)$ 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(key)	1/8	1/32	1/16	1/32	1/4	1/2



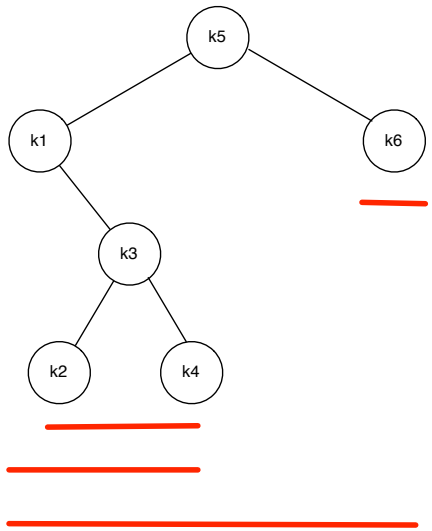
$$Cost = \frac{1}{8}\times 2+\frac{1}{32}\times 4+\frac{1}{16}\times 3+\frac{1}{32}\times 4+\frac{1}{4}\times 1+\frac{1}{2}\times 2 = \frac{31}{16}$$

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

```
1  fun OBST(S) =
2      if |S| = 0 then 0
3      else (∑_{s∈S} p(s)) + min_{i∈⟨1...|S|⟩}(OBST(S_{1,i−1}) +
4                                              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

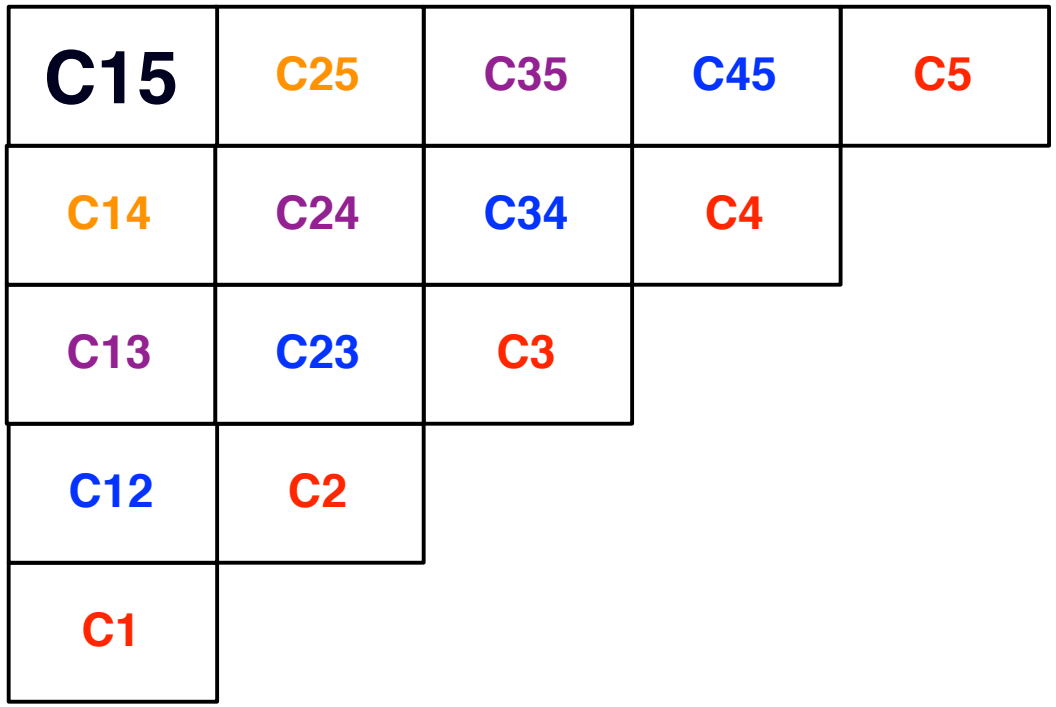
- Cost of each subproblem is not uniform! (Why?)
- Each subproblem has $O(n)$ work and $O(\log n)$ span (Why?)
- We get total $O(n^3)$ work and $O(n \log n)$ span. (Why?)

Code for Optimal BST

```
1  fun OBST(S) = let
2      fun OBST'(i, l) =
3          if l = 0 then 0
4          else ∑_{k=0}^{l-1} p(S_{i+k}) + min_{k=0}^{l-1}(OBST'(i, k)+
5                                                  OBST'(i + k + 1, l − k − 1))
6  in
7      OBST'(1, |S|)
8  end
```

Bottom-up Optimal BST

- For a bottom up version, a triangular table is sufficient



cij = optimal cost of the tree covering Sij