CMPUT 498/501: Advanced Algorithms **Fall 2025**

Lecture 3 (Sep 10, 2025): Advanced Dynamic Programming

Lecturer: Mohammad R. Salavatipour *Scribe: Mohsen Mohammadi, Parsa Zarezadeh*

3.1 Optimal Binary Search Trees: A Recap

We recall the Optimal Binary Search Tree (BST) problem. We are given a set of keys $k_1 < k_2 < \ldots < k_n$ and "dummy" keys $d_0 < d_1 < \ldots < d_n$, where d_i represents values between k_i and k_{i+1}. We are also given frequencies:

- p_i: the frequency of searching for key k_i.
- q_i: the frequency of searching for a value in the range of d_i.

The goal is to construct a BST that minimizes the expected search cost. If $d_T(x)$ is the depth of a node x in a tree T (with the root at depth 0), the expected cost is:

$$E[\text{cost}] = \sum_{i=1}^{n}(d_T(k_i)+1)\cdot p_i + \sum_{i=0}^{n}(d_T(d_i)+1)\cdot q_i$$

i	0	1	2	3	4	5
p_i		0.15	0.20	0.05	0.15	0.10
q_i	0.05	0.05	0.05	0.05	0.05	0.10

Table 3.1: Example frequencies for keys and dummy keys.

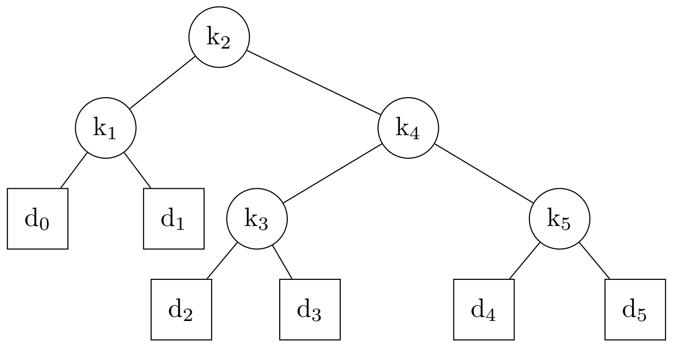

Figure 3.1: Optimal BST structures for the given frequencies.

Let $f[i,j] = \sum_{l=i}^{j} p_l + \sum_{l=i-1}^{j} q_l$ and let $O[i,j]$ be the minimum expected search cost for a BST on the keys $k_i, \ldots, k_j$ (and dummy keys $d_{i-1}, \ldots, d_j$). If we choose k_r as the root for this subtree, its left child will be the root of an optimal BST for keys $k_i, \ldots, k_{r-1}$ and its right child will be the root of an optimal BST for keys $k_{r+1}, \ldots, k_j$. So the cost of this subtree is:

$$\begin{aligned} O[i,j] &= p_r + (O[i,r-1] + f[i,r-1]) + (O[r+1,j] + f[r+1,j]) \\ &= O[i,r-1] + O[r+1,j] + f[i,j] \end{aligned}$$

When k_r is the root, all nodes in its left and right subtrees increase their depth by 1. The cost increase is the sum of all frequencies in that subtree. The recurrence relation is:

$$O[i,j] = \min_{i \le r \le j} \{O[i,r-1] + O[r+1,j] + f[i,j]\}$$

The base case is $O[i,i-1] = q_{i-1}$ for $1 \le i \le n+1$.

Computing the entire table for $O[i,j]$ involves filling $O(n^2)$ entries. For each entry, we must check $O(n)$ possible roots k_r. This leads to a total time complexity of $O(n^3)$.

We also define the ROOT$[i,j]$ table, where ROOT$[i,j]$ stores the index r of the key k_r that yields the minimum expected cost for the subproblem on keys $k_i, \ldots, k_j$. This table allows us to reconstruct the optimal BST structure.

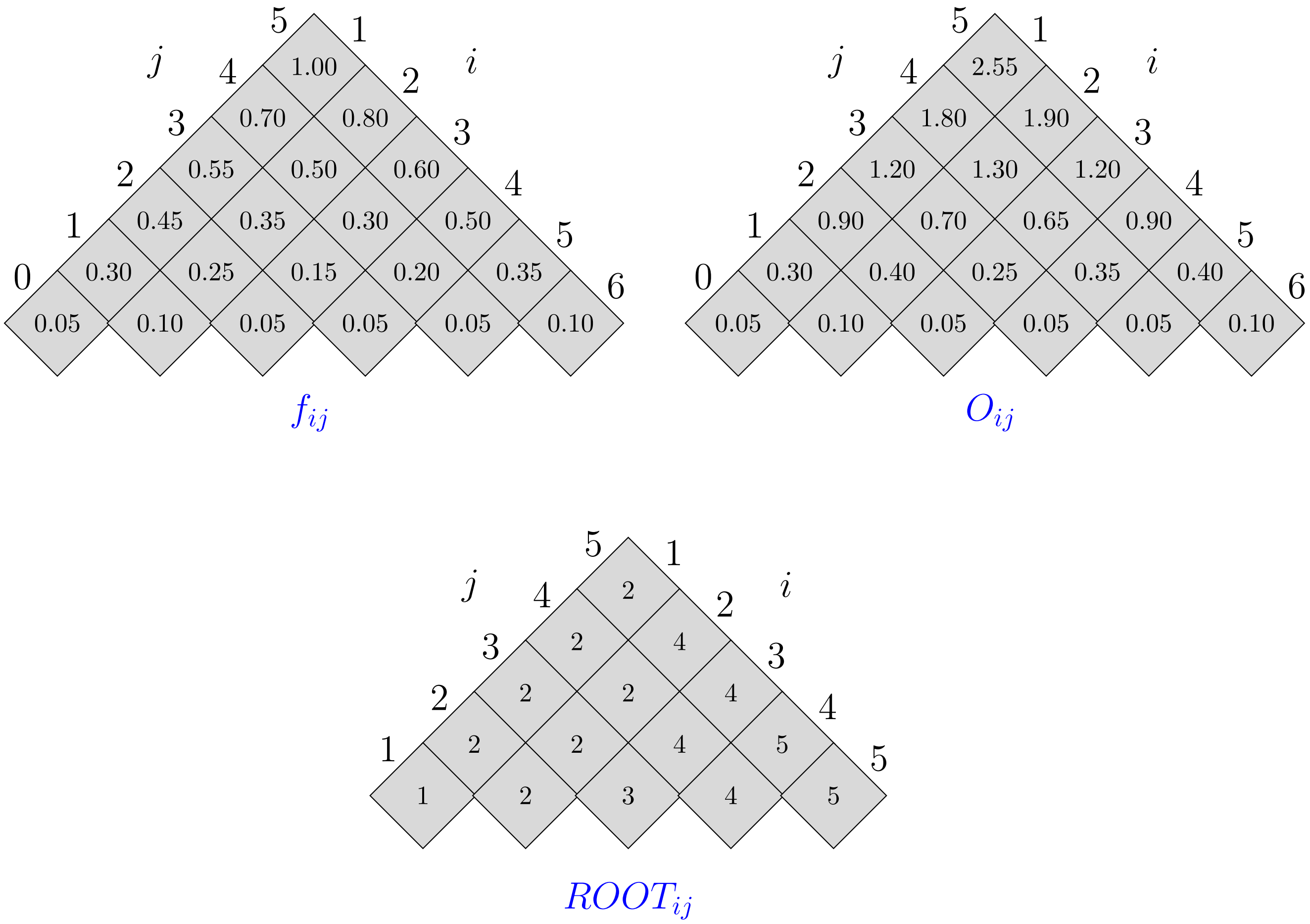

Figure 3.2: Tables for f_{ij}, O_{ij}, and ROOT$_{ij}$ for the given frequencies.

3.2 Improving Time with Monotonicity

3.2.1 The Knuth-Yao Speedup for Optimal BST

The $O(n^3)$ complexity can be improved. In 1971, Knuth proved a monotonicity property for the optimal root indices.

Lemma 1 (Knuth '71) *For all valid i, j, the optimal root index satisfies the following inequality:*

$$\textsc{root}[i, j-1] \le \textsc{root}[i, j] \le \textsc{root}[i+1, j]$$

This property implies that every row and column in the $\textsc{root}[\cdot, \cdot]$ table is sorted. We can leverage this to significantly reduce the search space for the optimal root r. Instead of searching for r in the range $[i, j]$, we only need to search in the much smaller range $[\textsc{root}[i, j-1], \textsc{root}[i+1, j]]$.

The improved algorithm, Faster-Opt-BST, is as follows:

Algorithm 1 Faster-Opt-BST

```
1: Compute all $f[i,j]$ values.
2: for $i \leftarrow 1$ to $n+1$ do
3:     $O[i, i-1] \leftarrow q_{i-1}$
4: for $l \leftarrow 1$ to $n$ do
5:     for $i \leftarrow 1$ to $n-l+1$ do
6:         $j \leftarrow i+l-1$
7:         $O[i,j] \leftarrow -\infty$
8:         for $r \leftarrow \textsc{root}[i, j-1]$ to $\textsc{root}[i+1, j]$ do
9:             $temp \leftarrow O[i, r-1] + O[r+1, j] + f[i,j]$
10:            if $O[i,j] > temp$ then
11:                $O[i,j] \leftarrow temp$
12:                $\textsc{root}[i,j] \leftarrow r$
   return $O[1,n]$
```

The total work for the innermost loop, for a fixed diagonal length $l = j - i$, is:

$$\sum_{i=1}^{n-l} (\textsc{root}[i+1, i+l-1] - \textsc{root}[i, i+l-2] + 1)$$

This sum telescopes, resulting in a complexity of $O(n)$ for each diagonal. Since there are $O(n)$ diagonals, the total time complexity is reduced to $O(n^2)$. Further improvements by Hu & Tucker have reduced the complexity to $O(n \log n)$.

3.2.2 Finding row minimums Problem

This speedup is an instance of a more general principle. Consider the problem of finding the minimum element in each row of an $n \times m$ matrix M. The naive approach takes $O(nm)$ time. However, if the matrix has a special structure, we can do better.

Definition 1 *A matrix M is **monotone** if the index of the leftmost minimum element in any row is not to the left of the index of the leftmost minimum of the previous row. If $LM[i]$ is the column index of the leftmost minimum in row i, then for all i:*

$$LM[i] \leq LM[i+1]$$

$$\begin{bmatrix} 8 & 21 & 35 & 15 & 29 \\ 74 & 13 & 19 & 29 & 53 \\ 21 & 7 & 27 & 19 & 75 \\ 63 & 43 & 19 & 11 & 75 \\ 10 & 7 & 19 & 2 & 4 \end{bmatrix}$$

Figure 3.3: A monotone matrix with leftmost minimum highlighted.

For monotone matrices, we can find all row minimum more efficiently. A divide-and-conquer approach works as follows:

1. Recursively find the row minimum for all odd-numbered rows.
2. For each even row $2i$, we know from monotonicity that $LM[2i-1] \leq LM[2i] \leq LM[2i+1]$. We can find the minimum for row $2i$ by searching only in the range of columns $[LM[2i-1], LM[2i+1]]$.

The total time to compute minimum for even rows is $\sum_{i=1}^{n/2}(LM[2i+1] - LM[2i-1] + 1) = O(n+m)$. The recurrence is $T(n,m) = T(n/2,m) + O(n+m)$, which solves to $O(m + n \log m)$.

3.3 The SMAWK Algorithm

The SMAWK algorithm provides an even faster $O(n+m)$ solution for a class of matrices known as totally monotone matrices.

3.3.1 Monotonicity and Monge Arrays

Definition 2 *A matrix is **totally monotone** if every 2×2 submatrix is monotone.*

A related and stronger property is the Monge property.

Definition 3 *A matrix M has the **Monge property** if for all $i < i'$ and $j < j'$:*

$$M[i,j] + M[i',j'] \leq M[i,j'] + M[i',j]$$

Lemma 2 *Every Monge matrix is totally monotone.*

Proof. Consider a 2×2 submatrix with rows i, i' and columns j, j'. If it's not monotone, then $M[i,j] > M[i,j']$ and $M[i',j] \leq M[i',j']$. Rearranging these gives $M[i,j] - M[i,j'] > 0$ and $M[i',j] - M[i',j'] \leq 0$. Summing these inequalities violates the Monge property. ■

$$\begin{bmatrix} 8 & 21 & 35 & 15 & 29 \\ 74 & 13 & 19 & 29 & 53 \\ 21 & 7 & 27 & 19 & 75 \\ 63 & 43 & 19 & 11 & 75 \\ 10 & 7 & 19 & 2 & 4 \end{bmatrix}$$

Figure 3.4: 2×2 submatrices highlighted is not totally monotone.

3.3.2 Overview of the SMAWK Algorithm

The SMAWK algorithm runs in $O(n+m)$ time. It consists of two main procedures that reduce the problem size.

- **SPARSIFY**: If the matrix is tall $(n > m)$, this procedure eliminates rows. It recursively finds the minimum for odd rows, then uses monotonicity to find minimum for even rows in $O(n+m)$ time. This reduces the problem from size $n \times m$ to $n/2 \times m$.
- **REDUCE**: If the matrix is wide $(n \leq m)$, this procedure eliminates columns that are guaranteed not to contain a row minimum. It reduces an $n \times m$ matrix to an $n \times n$ matrix in $O(n+m)$ time.

The time complexity follows the recurrence:

$$T(n,m) \leq \begin{cases} O(n+m) + T(n/2, m) & \text{if } n > m \\ O(n+m) + T(n,n) & \text{if } n \leq m \end{cases}$$

This recurrence solves to $O(n+m)$.

3.3.3 The REDUCE Procedure

The Reduce procedure works by identifying and eliminating "dead" columns. The key observation comes from comparing two entries in the same row, $M[i,p]$ and $M[i,q]$ with $p < q$.

- If $M[i,p] \geq M[i,q]$, then due to the total monotonicity property, for all rows $j \geq i$, we must have $M[j,p] \geq M[j,q]$. This means column p cannot contain the leftmost minimum for any row from i onwards. We say column p is "dead" for these rows.
- If $M[i,p] < M[i,q]$, then for all rows $h < i$, we must have $M[h,p] < M[h,q]$. This means column q cannot be the leftmost minimum for any row from 1 to i.

Algorithm 2 REDUCE(M)

```
1: $S \leftarrow$ empty stack
2: for $k \leftarrow 1$ to $m$ do
3:     while stack is not empty AND $M[\text{size}(S), S.\text{top}()] \geq M[\text{size}(S), k]$ do
4:         $S$.pop()
5:     if size($S$) $< n$ then
6:         $S$.push($k$)
   return columns in $S$
```

REDUCE maintains a stack of "surviving" column indices. It iterates through columns $k = 1, \ldots, m$, deciding whether to pop from the stack or push the new column k. The algorithm works as follows. Let t be the number of items on the stack, and let the top of the stack be column $S[t]$. When considering a new column k:

1. We compare $M[t, S[t]]$ with $M[t, k]$.
2. If $M[t, S[t]] \geq M[t, k]$, then column $S[t]$ is dead for row t and all subsequent rows. We can pop it and repeat the comparison with the new top of the stack.
3. If $M[t, S[t]] < M[t, k]$, column $S[t]$ might still be the minimum for row t. Column k cannot be the minimum for row t (or any previous row), but it could be for a subsequent row. We stop popping and push k onto the stack (if there is still room, i.e., $t < n$).

Each column is pushed once and popped at most once. The number of comparisons is therefore $O(n + m)$. The final stack contains at most n columns, which are guaranteed to contain all the row minimum.

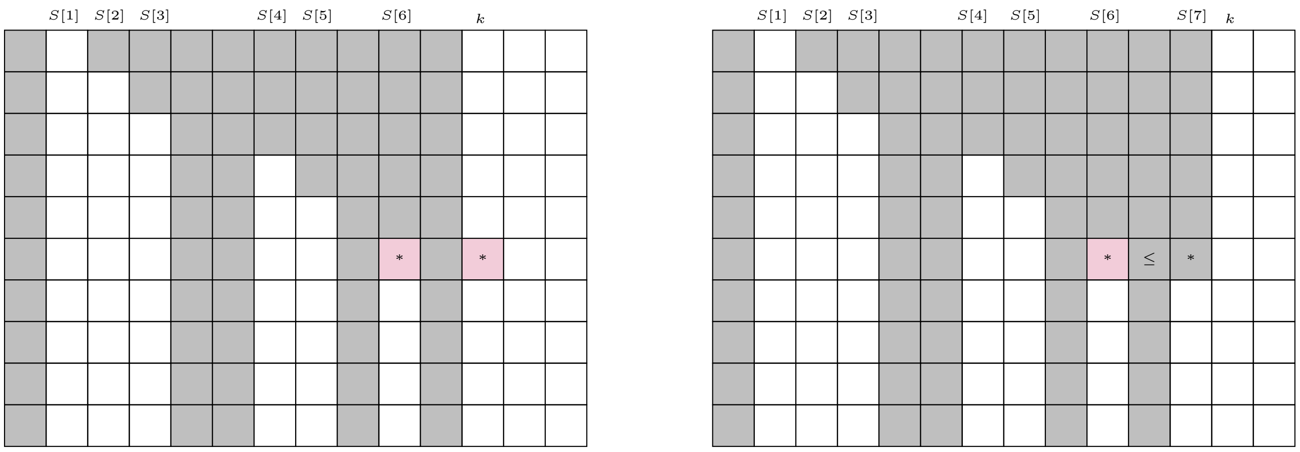

Figure 3.5: updating the stack in REDUCE. The first table shows the state before considering column k. The second table shows the state after processing column k.