Lecture 13: Dynamic Programming II

Michael Dinitz

October 7, 2025

601.433/633 Introduction to Algorithms

Introduction

Today: two more examples of dynamic programming

- *Longest Common Subsequence* (strings)
- *Optimal Binary Search Tree* (trees)

Important problems, but really: more examples of dynamic programming

Both in CLRS (unlike Weighted Interval Scheduling)

Longest Common Subsequence

Definitions

String: Sequence of elements of some *alphabet* ($\{0, 1\}$, or $\{A - Z\} \cup \{a - z\}$, etc.)

Definition: A sequence $Z = (z_1, \ldots, z_k)$ is a *subsequence* of $X = (x_1, \ldots, x_m)$ if there exists a strictly increasing sequence $(i_1, i_2, \ldots, i_k)$ such that $x_{i_j} = z_j$ for all $j \in \{1, 2, \ldots, k\}$.

Example: (B, C, D, B) is a subsequence of (A, B, C, B, D, A, B)

- Allowed to skip positions, unlike substring!

Definitions

String: Sequence of elements of some *alphabet* ($\{0, 1\}$, or $\{A - Z\} \cup \{a - z\}$, etc.)

Definition: A sequence $Z = (z_1, \ldots, z_k)$ is a *subsequence* of $X = (x_1, \ldots, x_m)$ if there exists a strictly increasing sequence $(i_1, i_2, \ldots, i_k)$ such that $x_{i_j} = z_j$ for all $j \in \{1, 2, \ldots, k\}$.

Example: (B, C, D, B) is a subsequence of (A, B, C, B, D, A, B)

- Allowed to skip positions, unlike substring!

Definition: In *Longest Common Subsequence* problem (LCS) we are given two strings $X = (x_1, \ldots, x_m)$ and $Y = (y_1, \ldots y_n)$. Need to find the longest Z which is a subsequence of both X and Y.

Subproblems

First and most important step of dynamic programming: define subproblems!

- Not obvious: $\boldsymbol{X}$ and $\boldsymbol{Y}$ might not even be same length!

Subproblems

First and most important step of dynamic programming: define subproblems!

- Not obvious: $\boldsymbol{X}$ and $\boldsymbol{Y}$ might not even be same length!

Prefixes of strings

- $\boldsymbol{X_i = (x_1, x_2, \ldots, x_i)}$ (so $\boldsymbol{X = X_m}$)
- $\boldsymbol{Y_j = (y_1, y_2, \ldots, y_j)}$ (so $\boldsymbol{Y = Y_n}$)

Subproblems

First and most important step of dynamic programming: define subproblems!

- Not obvious: X and Y might not even be same length!

Prefixes of strings

- $X_i = (x_1, x_2, \ldots, x_i)$ (so $X = X_m$)
- $Y_j = (y_1, y_2, \ldots, y_j)$ (so $Y = Y_n$)

Definition: Let $OPT(i, j)$ be longest common subsequence of X_i and Y_j

So looking for optimal solution $OPT = OPT(m, n)$

- Last time OPT denotes value of solution, here denotes solution. Be flexible in notation

Subproblems

First and most important step of dynamic programming: define subproblems!

- Not obvious: X and Y might not even be same length!

Prefixes of strings

- $X_i = (x_1, x_2, \ldots, x_i)$ (so $X = X_m$)
- $Y_j = (y_1, y_2, \ldots, y_j)$ (so $Y = Y_n$)

Definition: Let $OPT(i,j)$ be longest common subsequence of X_i and Y_j

So looking for optimal solution $OPT = OPT(m,n)$

- Last time OPT denotes value of solution, here denotes solution. Be flexible in notation

Two-dimensional table!

Optimal Substructure

Second step of dynamic programming: prove optimal substructure

- Relationship between subproblems: show that solution to subproblem can be found from solutions to smaller subproblems

Optimal Substructure

Second step of dynamic programming: prove optimal substructure

- Relationship between subproblems: show that solution to subproblem can be found from solutions to smaller subproblems

Theorem

Let $\boldsymbol{Z} = (\boldsymbol{z_1}, \ldots, \boldsymbol{z_k})$ be an LCS of $\boldsymbol{X_i}$ and $\boldsymbol{Y_j}$ (so $\boldsymbol{Z} = \boldsymbol{OPT(i, j)}$).

1. *If $\boldsymbol{x_i} = \boldsymbol{y_j}$:*

Optimal Substructure

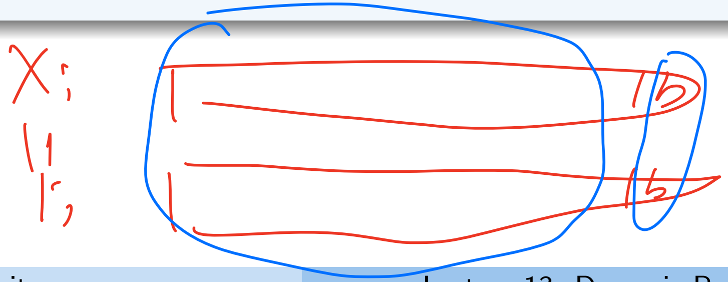
Second step of dynamic programming: prove optimal substructure

- Relationship between subproblems: show that solution to subproblem can be found from solutions to smaller subproblems

Theorem

Let $\boldsymbol{Z} = (\boldsymbol{z_1}, \dots, \boldsymbol{z_k})$ be an LCS of $\boldsymbol{X_i}$ and $\boldsymbol{Y_j}$ (so $\boldsymbol{Z} = \boldsymbol{OPT(i, j)}$).

1. *If $\boldsymbol{x_i} = \boldsymbol{y_j}$: then $\boldsymbol{z_k} = \boldsymbol{x_i} = \boldsymbol{y_j}$ and $\boldsymbol{Z_{k-1}} = \boldsymbol{OPT(i-1, j-1)}$*



Optimal Substructure

Second step of dynamic programming: prove optimal substructure

- Relationship between subproblems: show that solution to subproblem can be found from solutions to smaller subproblems

Theorem

Let $\boldsymbol{Z = (z_1, \ldots, z_k)}$ *be an LCS of* $\boldsymbol{X_i}$ *and* $\boldsymbol{Y_j}$ *(so* $\boldsymbol{Z = OPT(i,j)}$*).*

1. *If* $\boldsymbol{x_i = y_j}$*: then* $\boldsymbol{z_k = x_i = y_j}$ *and* $\boldsymbol{Z_{k-1} = OPT(i-1, j-1)}$
2. *If* $\boldsymbol{x_i \neq y_j}$ *and* $\boldsymbol{z_k \neq x_i}$*:*

Optimal Substructure

Second step of dynamic programming: prove optimal substructure

- Relationship between subproblems: show that solution to subproblem can be found from solutions to smaller subproblems

Theorem

Let $\boldsymbol{Z} = (\boldsymbol{z_1}, \ldots, \boldsymbol{z_k})$ be an LCS of $\boldsymbol{X_i}$ and $\boldsymbol{Y_j}$ (so $\boldsymbol{Z} = \boldsymbol{OPT(i, j)}$).

1. *If $\boldsymbol{x_i} = \boldsymbol{y_j}$: then $\boldsymbol{z_k} = \boldsymbol{x_i} = \boldsymbol{y_j}$ and $\boldsymbol{Z_{k-1}} = \boldsymbol{OPT(i-1, j-1)}$*
2. *If $\boldsymbol{x_i} \neq \boldsymbol{y_j}$ and $\boldsymbol{z_k} \neq \boldsymbol{x_i}$: then $\boldsymbol{Z} = \boldsymbol{OPT(i-1, j)}$*

Optimal Substructure

Second step of dynamic programming: prove optimal substructure

- Relationship between subproblems: show that solution to subproblem can be found from solutions to smaller subproblems

Theorem

Let $\mathbf{Z} = (\mathbf{z_1}, \ldots, \mathbf{z_k})$ be an LCS of $\mathbf{X_i}$ and $\mathbf{Y_j}$ (so $\mathbf{Z} = \mathbf{OPT(i, j)}$).

1. *If $\mathbf{x_i} = \mathbf{y_j}$: then $\mathbf{z_k} = \mathbf{x_i} = \mathbf{y_j}$ and $\mathbf{Z_{k-1}} = \mathbf{OPT(i-1, j-1)}$*
2. *If $\mathbf{x_i} \neq \mathbf{y_j}$ and $\mathbf{z_k} \neq \mathbf{x_i}$: then $\mathbf{Z} = \mathbf{OPT(i-1, j)}$*
3. *If $\mathbf{x_i} \neq \mathbf{y_j}$ and $\mathbf{z_k} \neq \mathbf{y_j}$:*

Optimal Substructure

Second step of dynamic programming: prove optimal substructure

- Relationship between subproblems: show that solution to subproblem can be found from solutions to smaller subproblems

Theorem

Let $\boldsymbol{Z} = (z_1, \ldots, z_k)$ be an LCS of $\boldsymbol{X_i}$ and $\boldsymbol{Y_j}$ (so $\boldsymbol{Z} = \boldsymbol{OPT(i,j)}$).

1. *If $\boldsymbol{x_i = y_j}$: then $\boldsymbol{z_k = x_i = y_j}$ and $\boldsymbol{Z_{k-1} = OPT(i-1, j-1)}$*
2. *If $\boldsymbol{x_i \neq y_j}$ and $\boldsymbol{z_k \neq x_i}$: then $\boldsymbol{Z = OPT(i-1, j)}$*
3. *If $\boldsymbol{x_i \neq y_j}$ and $\boldsymbol{z_k \neq y_j}$: then $\boldsymbol{Z = OPT(i, j-1)}$*

Optimal Substructure: Proof (I)

Case 1: If $x_i = y_j$, then $z_k = x_i = y_j$ and $Z_{k-1} = OPT(i-1, j-1)$

Proof Sketch.

Contradiction.

Optimal Substructure: Proof (I)

Case 1: If $x_i = y_j$, then $z_k = x_i = y_j$ and $Z_{k-1} = OPT(i-1, j-1)$

Proof Sketch.

Contradiction.

Part 1: Suppose $x_i = y_j = a$, but $z_k \neq a$.

Optimal Substructure: Proof (I)

Case 1: If $x_i = y_j$, then $z_k = x_i = y_j$ and $Z_{k-1} = OPT(i-1, j-1)$

Proof Sketch.

Contradiction.

Part 1: Suppose $x_i = y_j = a$, but $z_k \neq a$. Add a to end of Z, still have common subsequence, longer than LCS. Contradiction

Optimal Substructure: Proof (I)

Case 1: If $x_i = y_j$, then $z_k = x_i = y_j$ and $Z_{k-1} = OPT(i-1, j-1)$

Proof Sketch.

Contradiction.

Part 1: Suppose $x_i = y_j = a$, but $z_k \neq a$. Add a to end of Z, still have common subsequence, longer than LCS. Contradiction

Part 2: Suppose $Z_{k-1} \neq OPT(i-1, j-1)$.

Optimal Substructure: Proof (I)

Case 1: If $x_i = y_j$, then $z_k = x_i = y_j$ and $Z_{k-1} = OPT(i-1, j-1)$

Proof Sketch.

Contradiction.

Part 1: Suppose $x_i = y_j = a$, but $z_k \neq a$. Add a to end of Z, still have common subsequence, longer than LCS. Contradiction

Part 2: Suppose $Z_{k-1} \neq OPT(i-1, j-1)$.

$\implies$ $\exists W$ LCS of X_{i-1}, Y_{j-1} of length $> k-1 \implies \geq k$

$\implies$ (W, a) common subsequence of X_i, Y_j of length $> k$

- Contradiction to Z being LCS of X_i and Y_j $\square$

Optimal Substructure: Proof (II)

Case 2: If $x_i \neq y_j$ and $z_k \neq x_i$ then $Z = OPT(i-1, j)$

Optimal Substructure: Proof (II)

Case 2: If $x_i \neq y_j$ and $z_k \neq x_i$ then $Z = OPT(i-1, j)$

Proof.

Since $z_k \neq x_i$, Z a common subsequence of $X_{i-1}, Y_j \implies |Z| \leq |OPT(i-1, j)|$

Optimal Substructure: Proof (II)

Case 2: If $x_i \neq y_j$ and $z_k \neq x_i$ then $Z = OPT(i-1, j)$

Proof.

Since $z_k \neq x_i$, Z a common subsequence of $X_{i-1}, Y_j \implies |Z| \leq |OPT(i-1,j)|$

$OPT(i-1,j)$ a common subsequence of X_i, Y_j

$\implies |OPT(i-1,j)| \leq |OPT(i,j)| = |Z|$ (def of $OPT(i,j)$ and Z)

Optimal Substructure: Proof (II)

Case 2: If $x_i \neq y_j$ and $z_k \neq x_i$ then $Z = OPT(i-1, j)$

Proof.

Since $z_k \neq x_i$, Z a common subsequence of $X_{i-1}, Y_j \implies |Z| \leq |OPT(i-1, j)|$

$OPT(i-1, j)$ a common subsequence of X_i, Y_j

$\implies |OPT(i-1,j)| \leq |OPT(i,j)| = |Z|$ (def of $OPT(i,j)$ and Z)

$\implies Z = OPT(i-1, j)$ □

Optimal Substructure: Proof (III)

Case 3: If $x_i \neq y_j$ and $z_k \neq y_j$ then $Z = OPT(i, j-1)$

Proof.

Symmetric to Case 2. $\square$

Structure Corollary

$(OPT(i-1, j-1), x_i)$

Corollary

ϕ

$$OPT(i,j) = \begin{cases} \emptyset & \text{if } i = 0 \text{ or } j = 0, \\ OPT(i-1,j-1) \circ x_i & \text{if } i,j > 0 \text{ and } x_i = y_j \\ \text{arg}\max(|OPT(i,j-1)|, |OPT(i-1,j)|) & \text{if } i,j > 0 \text{ and } x_i \neq y_j \end{cases}$$

Structure Corollary

Corollary

$$OPT(i,j) = \begin{cases} \varnothing & \text{if } i = 0 \text{ or } j = 0, \\ OPT(i-1,j-1) \circ x_i & \text{if } i,j > 0 \text{ and } x_i = y_j \\ \max(OPT(i,j-1), OPT(i-1,j)) & \text{if } i,j > 0 \text{ and } x_i \neq y_j \end{cases}$$

Gives obvious recursive algorithm

- Can take exponential time (good exercise at home!)

Dynamic Programming!

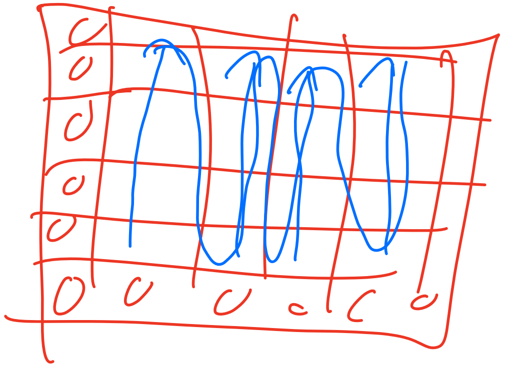
- Top-Down: are problems getting "smaller"? What does "smaller" mean?
- Bottom-Up: two-dimensional table! What order to fill it in?

Dynamic Programming Algorithm

LCS(X,Y) {
    for($i = 0$ to m) $M[i, 0] = 0$;
    for($j = 0$ to n) $M[0, j] = 0$;
    for($i = 1$ to m) {
        for($j = 1$ to n) {
            if($x_i = y_j$)
                $M[i, j] = 1 + M[i-1, j-1]$;
            else
                $M[i, j] = \max(M[i, j-1], M[i-1, j])$;
        }
    }
    return $M[m, n]$;
}

Dynamic Programming Algorithm

```
LCS(X,Y) {
    for(i = 0 to m) M[i,0] = 0;
    for(j = 0 to n) M[0,j] = 0;
    for(i = 1 to m) {
        for(j = 1 to n) {
            if(x_i = y_j)
                M[i,j] = 1 + M[i-1,j-1];
            else
                M[i,j] = max(M[i,j-1], M[i-1,j]);
            }
        }
    return M[m,n];
}
```



Running Time: $O(mn)$

Correctness

Theorem

$M[i,j] = |OPT(i,j)|$

Correctness

Theorem

$M[i, j] = |OPT(i, j)|$

Proof.

Induction on $i + j$ (or could do on iterations in the algorithm)

Correctness

Theorem

$M[i,j] = |OPT(i,j)|$

Proof.

Induction on $i + j$ (or could do on iterations in the algorithm)

Base Case: $i + j = 0 \implies i = j = 0 \implies M[i,j] = 0 = |OPT(i,j)|$

Correctness

Theorem

$M[i,j] = |OPT(i,j)|$

Proof.

Induction on $i + j$ (or could do on iterations in the algorithm)

Base Case: $i + j = 0 \implies i = j = 0 \implies M[i,j] = 0 = |OPT(i,j)|$

Inductive Step: Divide into three cases

1. If $i = 0$ or $j = 0$, then $M[i,j] = 0 = |OPT(i,j)|$

Correctness

Theorem

$M[i,j] = |OPT(i,j)|$

Proof.

Induction on $i + j$ (or could do on iterations in the algorithm)

Base Case: $i + j = 0 \implies i = j = 0 \implies M[i,j] = 0 = |OPT(i,j)|$

Inductive Step: Divide into three cases

1. If $i = 0$ or $j = 0$, then $M[i,j] = 0 = |OPT(i,j)|$
2. If $x_i = y_j$, then $M[i,j] = 1 + M[i-1,j-1] = 1 + |OPT(i-1,j-1)| = |OPT(i,j)|$

alg   induction   ST

Correctness

Theorem

$M[i,j] = |OPT(i,j)|$

Proof.

Induction on $i + j$ (or could do on iterations in the algorithm)

Base Case: $i + j = 0 \implies i = j = 0 \implies M[i,j] = 0 = |OPT(i,j)|$

Inductive Step: Divide into three cases

1. If $i = 0$ or $j = 0$, then $M[i,j] = 0 = |OPT(i,j)|$
2. If $x_i = y_j$, then $M[i,j] = 1 + M[i-1,j-1] = 1 + |OPT(i-1,j-1)| = |OPT(i,j)|$
3. If $x_i \neq y_j$, then

$$\begin{aligned} M[i,j] &= \max(M[i,j-1], M[i-1,j]) && \text{(def of algorithm)} \\ &= \max(|OPT(i,j-1)|, |OPT(i-1,j)|) && \text{(induction)} \\ &= |OPT(i,j)| && \text{(structure thm/corollary)} \end{aligned}$$

Computing a Solution

Like we talked about last lecture: backtrack through dynamic programming table.

Details in CLRS 14.4

Optimal Binary Search Trees

Problem Definition

Input: probability distribution / search frequency of keys

- n distinct keys $k_1 < k_2 < \cdots < k_n$
- For each $i \in [n]$, probability p_i that we search for k_i (so $\sum_{i=1}^{n} p_i = 1$)

What's the best binary search tree for these keys and frequencies?

Problem Definition

Input: probability distribution / search frequency of keys

- n distinct keys $k_1 < k_2 < \cdots < k_n$
- For each $i \in [n]$, probability p_i that we search for k_i (so $\sum_{i=1}^{n} p_i = 1$)

What's the best binary search tree for these keys and frequencies?

Cost of searching for k_i in tree T is $depth_T(k_i) + 1$ (say depth of root $= 0$)

$\implies E[\text{cost of search in } T] = \sum_{i=1}^{n} p_i(depth_T(k_i) + 1)$

Problem Definition

Input: probability distribution / search frequency of keys

- n distinct keys $k_1 < k_2 < \cdots < k_n$
- For each $i \in [n]$, probability p_i that we search for k_i (so $\sum_{i=1}^{n} p_i = 1$)

What's the best binary search tree for these keys and frequencies?

Cost of searching for k_i in tree T is $depth_T(k_i) + 1$ (say depth of root $= 0$)

$\implies E[\text{cost of search in } T] = \sum_{i=1}^{n} p_i(depth_T(k_i) + 1)$

Definition: $c(T) = \sum_{i=1}^{n} p_i(depth_T(k_i) + 1)$

Problem: Find search tree T minimizing cost.

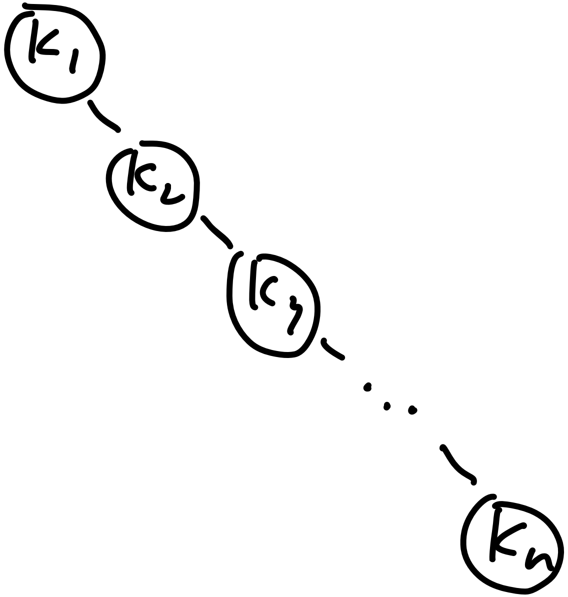
Obvious Approach

Natural approach: greedy (make highest probability key the root). Does this work?

Obvious Approach

Natural approach: greedy (make highest probability key the root). Does this work?

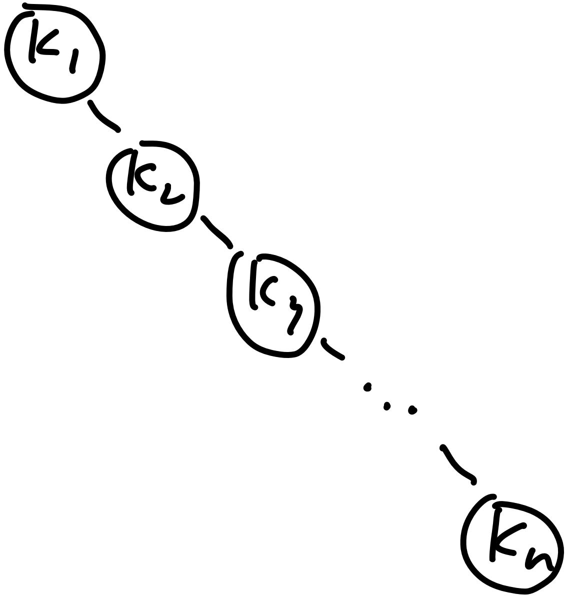
Set $p_1 > p_2 > \ldots p_n$, but with $p_i - p_{i+1}$ extremely small (say $\mathbf{1/2^n}$)



Obvious Approach

Natural approach: greedy (make highest probability key the root). Does this work?

Set $p_1 > p_2 > \ldots p_n$, but with $p_i - p_{i+1}$ extremely small (say $1/2^n$)

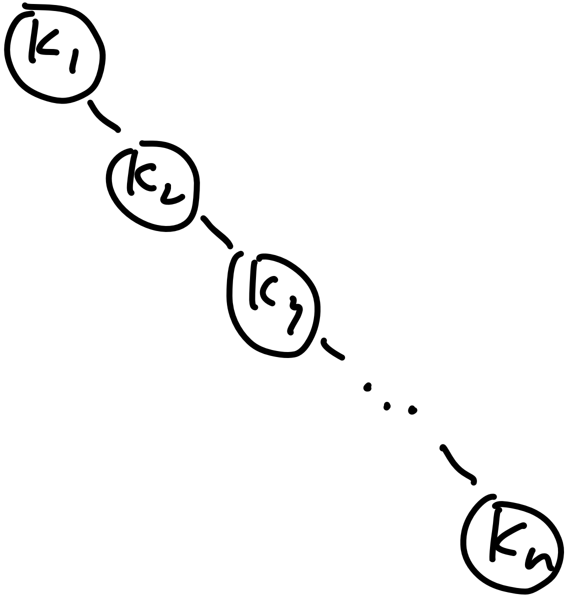


$\boldsymbol{E}[\text{cost of search in } \boldsymbol{T}] \geq \Omega(n)$

Obvious Approach

Natural approach: greedy (make highest probability key the root). Does this work?

Set $\boldsymbol{p_1 > p_2 > \dots p_n}$, but with $\boldsymbol{p_i - p_{i+1}}$ extremely small (say $\mathbf{1/2^n}$)

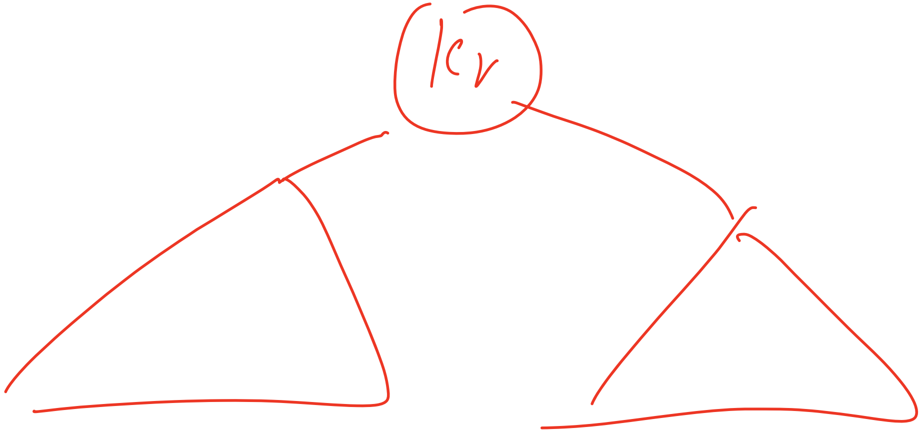


$\boldsymbol{E}[\text{cost of search in } \boldsymbol{T}] \geq \boldsymbol{\Omega(n)}$

Balanced search tree: $\boldsymbol{E}[\text{cost}] \leq \boldsymbol{O(\log n)}$

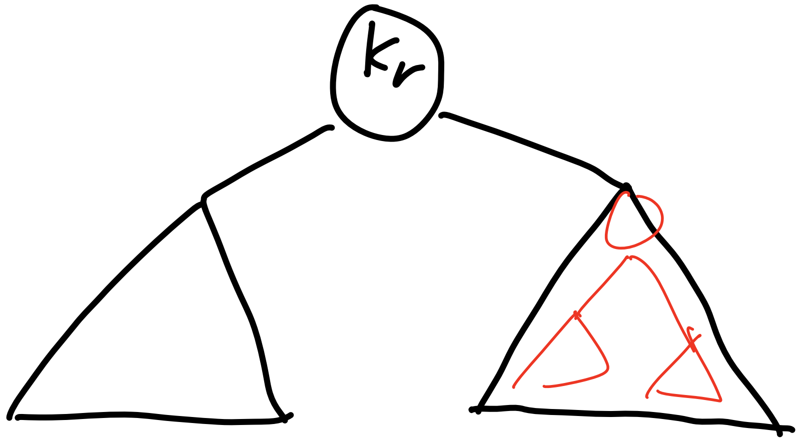
Intuition

Suppose root is $\boldsymbol{k_r}$. What does optimal tree look like?



Intuition

Suppose root is $\boldsymbol{k_r}$. What does optimal tree look like?



OPT for $\{k_1, \ldots, k_{r-1}\}$

OPT for $\{k_{r+1}, \ldots, k_n\}$

Subproblems

Definition

Let $OPT(i,j)$ with $i \le j$ be optimal tree for keys $\{k_i, k_{i+1}, \ldots, k_j\}$: tree T minimizing $c(T) = \sum_{a=i}^{j} p_a(depth_T(k_a) + 1)$

By convention, if $i > j$ then $OPT(i,j)$ empty
So overall goal is to find $OPT(1,n)$.

Subproblems

Definition

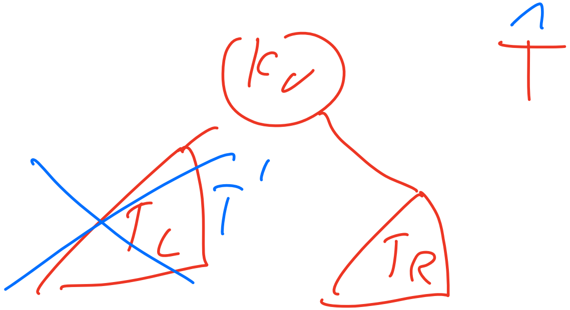
Let $OPT(i,j)$ with $i \leq j$ be optimal tree for keys $\{k_i, k_{i+1}, \ldots, k_j\}$: tree T minimizing $c(T) = \sum_{a=i}^{j} p_a(depth_T(k_a) + 1)$

By convention, if $i > j$ then $OPT(i,j)$ empty
So overall goal is to find $OPT(1,n)$.

Theorem (Optimal Substructure)

Let k_r be the root of $OPT(i,j)$. Then the left subtree of $OPT(i,j)$ is $OPT(i,r-1)$, and the right subtree of $OPT(i,j)$ is $OPT(r+1,j)$.

Proof Sketch of Optimal Substructure



Definitions:

- Let $T = OPT(i,j)$, T_L its left subtree, T_R its right subtree.
- Suppose for contradiction $T_L \neq OPT(i, r-1)$, let $T' = OPT(i, r-1)$

  $\implies$ $c(T') < c(T_L)$ (def of $OPT(i, r-1)$)
- Let $\hat{T}$ be tree get by replacing T_L with T'

Proof Sketch of Optimal Substructure

Definitions:

- Let $T = OPT(i,j)$, T_L its left subtree, T_R its right subtree.
- Suppose for contradiction $T_L \neq OPT(i, r-1)$, let $T' = OPT(i, r-1)$

  $\implies c(T') < c(T_L)$ (def of $OPT(i, r-1)$)
- Let $\hat{T}$ be tree get by replacing T_L with T'

Whole bunch of math (see lecture notes): get that $c(\hat{T}) < c(T)$

Contradicts $T = OPT(i,j)$

Proof Sketch of Optimal Substructure

Definitions:

- Let $T = OPT(i,j)$, T_L its left subtree, T_R its right subtree.
- Suppose for contradiction $T_L \neq OPT(i, r-1)$, let $T' = OPT(i, r-1)$

  $\implies c(T') < c(T_L)$ (def of $OPT(i, r-1)$)
- Let $\hat{T}$ be tree get by replacing T_L with T'

Whole bunch of math (see lecture notes): get that $c(\hat{T}) < c(T)$

Contradicts $T = OPT(i,j)$

Symmetric argument works for $T_R = OPT(r+1, j)$

Cost Corollary

Corollary

$c(OPT(i,j)) = \sum_{a=i}^{j} p_a + \min_{i \le r \le j}(c(OPT(i,r-1)) + c(OPT(r+1,j)))$

Let k_r be root of $OPT(i,j)$

$$c(OPT(i,j)) = \sum_{a=i}^{j} p_a(depth_{OPT(i,j)}(k_a) + 1) \quad \text{def}$$

$$\text{ST} \rightarrow \quad = \sum_{a=i}^{r-1}(p_a(depth_{OPT(i,r-1)}(k_a) + 2)) + p_r + \sum_{a=r+1}^{j} p_a(depth_{OPT(r+1,j)}(k_a) + 2)$$

$$\text{algebra} \rightarrow \quad = \sum_{a=i}^{j} p_a + \sum_{a=i}^{r-1}(p_a(depth_{OPT(i,r-1)}(k_a) + 1)) + \sum_{a=r+1}^{j} p_a(depth_{OPT(r+1,j)}(k_a) + 1)$$

$$\text{def of cost} \rightarrow \quad = \sum_{a=i}^{j} p_a + c(OPT(i,r-1)) + c(OPT(r+1,j)).$$

Cost Corollary

Corollary

$c(OPT(i,j)) = \sum_{a=i}^{j} p_a + \min_{i \le r \le j}(c(OPT(i, r-1)) + c(OPT(r+1, j)))$

Let k_r be root of $OPT(i,j)$

$$
\begin{aligned}
c(OPT(i,j)) &= \sum_{a=i}^{j} p_a(depth_{OPT(i,j)}(k_a) + 1) \\
&= \sum_{a=i}^{r-1}(p_a(depth_{OPT(i,r-1)}(k_a) + 2)) + p_r + \sum_{a=r+1}^{j} p_a(depth_{OPT(r+1,j)}(k_a) + 2) \\
&= \sum_{a=i}^{j} p_a + \sum_{a=i}^{r-1}(p_a(depth_{OPT(i,r-1)}(k_a) + 1)) + \sum_{a=r+1}^{j} p_a(depth_{OPT(r+1,j)}(k_a) + 1) \\
&= \sum_{a=i}^{j} p_a + c(OPT(i, r-1)) + c(OPT(r+1, j)).
\end{aligned}
$$

Same logic holds for any possible root $\implies$ take min

Algorithm

Fill in table M:

$$M[i,j] = \begin{cases} 0 & \text{if } i > j \\ \min_{i \le r \le j} \left(\sum_{a=i}^{j} p_a + M[i, r-1] + M[r+1, j] \right) & \text{if } i \le j \end{cases}$$

Algorithm

Fill in table $\boldsymbol{M}$:

$$\boldsymbol{M}[\boldsymbol{i},\boldsymbol{j}] = \begin{cases} \mathbf{0} & \text{if } \boldsymbol{i} > \boldsymbol{j} \\ \mathbf{min}_{\boldsymbol{i} \le \boldsymbol{r} \le \boldsymbol{j}} \left(\sum_{\boldsymbol{a}=\boldsymbol{i}}^{\boldsymbol{j}} \boldsymbol{p}_{\boldsymbol{a}} + \boldsymbol{M}[\boldsymbol{i}, \boldsymbol{r}-\mathbf{1}] + \boldsymbol{M}[\boldsymbol{r}+\mathbf{1}, \boldsymbol{j}] \right) & \text{if } \boldsymbol{i} \le \boldsymbol{j} \end{cases}$$

Top-Down (memoization): are problems getting smaller?

Algorithm

Fill in table $\boldsymbol{M}$:

$$\boldsymbol{M[i,j]} = \begin{cases} \mathbf{0} & \text{if } \boldsymbol{i > j} \\ \mathbf{min}_{\boldsymbol{i \le r \le j}} \left(\sum_{\boldsymbol{a=i}}^{\boldsymbol{j}} \boldsymbol{p_a} + \boldsymbol{M[i, r-1]} + \boldsymbol{M[r+1, j]} \right) & \text{if } \boldsymbol{i \le j} \end{cases}$$

Top-Down (memoization): are problems getting smaller? Yes! $\boldsymbol{j - i}$ decreases in every recursive call.

Algorithm

Fill in table M:

$$M[i,j] = \begin{cases} 0 & \text{if } i > j \\ \min_{i \le r \le j} \left(\sum_{a=i}^{j} p_a + M[i, r-1] + M[r+1, j] \right) & \text{if } i \le j \end{cases}$$

Top-Down (memoization): are problems getting smaller? Yes! $j - i$ decreases in every recursive call.

Correctness. Claim $M[i,j] = c(OPT(i,j))$. Induction on $j - i$.

Algorithm

Fill in table M:

$$M[i,j] = \begin{cases} 0 & \text{if } i > j \\ \min_{i \le r \le j} \left(\sum_{a=i}^{j} p_a + M[i, r-1] + M[r+1, j] \right) & \text{if } i \le j \end{cases}$$

Top-Down (memoization): are problems getting smaller? Yes! $j - i$ decreases in every recursive call.

Correctness. Claim $M[i,j] = c(OPT(i,j))$. Induction on $j - i$.

- Base case: if $j - i < 0$ then $M[i,j] = OPT(i,j) = 0$

Algorithm

Fill in table M:

$$M[i,j] = \begin{cases} 0 & \text{if } i > j \\ \min_{i \le r \le j} \left(\sum_{a=i}^{j} p_a + M[i, r-1] + M[r+1, j] \right) & \text{if } i \le j \end{cases}$$

Top-Down (memoization): are problems getting smaller? Yes! $j - i$ decreases in every recursive call.

Correctness. Claim $M[i,j] = c(OPT(i,j))$. Induction on $j - i$.

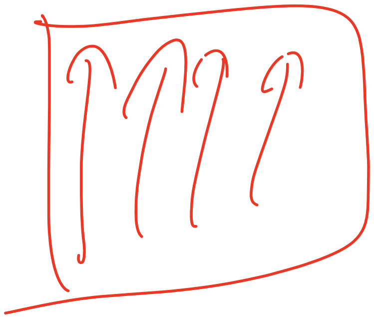
- Base case: if $j - i < 0$ then $M[i,j] = OPT(i,j) = 0$
- Inductive step:

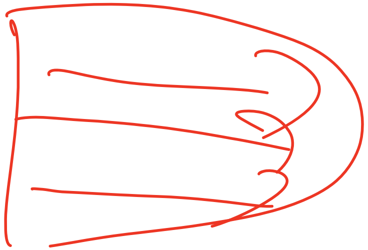
$$\begin{aligned} M[i,j] &= \min_{i \le r \le j} \left(\sum_{a=i}^{j} p_a + M[i, r-1] + M[r+1, j] \right) && \text{(alg def)} \\ &= \min_{i \le r \le j} \left(\sum_{a=i}^{j} p_a + c(OPT(i, r-1)) + c(OPT(r+1, j)) \right) && \text{(induction)} \\ &= c(OPT(i,j)) && \text{(cost corollary)} \end{aligned}$$

Algorithm: Bottom-up

What order to fill the table in?

- Obvious approach: for($\boldsymbol{i} = \mathbf{1}$ to $\boldsymbol{n} - \mathbf{1}$) for($\boldsymbol{j} = \boldsymbol{i} + \mathbf{1}$ to $\boldsymbol{n}$) Doesn't work!

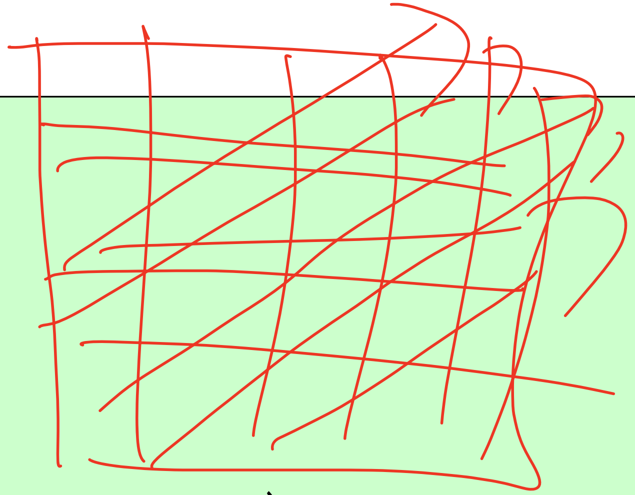




Algorithm: Bottom-up

What order to fill the table in?

- Obvious approach: for($i = 1$ to $n - 1$) for($j = i + 1$ to n) Doesn't work!
- Take hint from induction: $j - i$



OBST {
    Set $M[i, j] = 0$ for all $j > i$;
    Set $M[i, i] = p_i$ for all i
    for($\ell = 1$ to $n - 1$) {
        for($i = 1$ to $n - \ell$) {
            $j = i + \ell$
            $M[i, j] = \min_{i \le r \le j} \left(\sum_{a=i}^{j} p_a + M[i, r-1] + M[r+1, j] \right)$;
        }
    }
    return $M[1, n]$;
}

Analysis

Correctness: same as top-down

Running Time:

Analysis

Correctness: same as top-down

Running Time:

- # table entries:

Analysis

Correctness: same as top-down

Running Time:

- # table entries: $O(n^2)$

Analysis

Correctness: same as top-down

Running Time:

- # table entries: $O(n^2)$
- Time to compute table entry $M[i,j]$:

Analysis

Correctness: same as top-down

Running Time:

- # table entries: $O(n^2)$
- Time to compute table entry $M[i,j]$: $O(j - i) = O(n)$

Analysis

Correctness: same as top-down

Running Time:

- # table entries: $O(n^2)$
- Time to compute table entry $M[i,j]$: $O(j-i) = O(n)$

Total running time: $O(n^3)$

Bonus Content

Obvious Question: Robustness.

- What if given distribution is *wrong*?

Want algorithm that gives a solution with cost a function of true optimal cost, “distance” between given distribution and true distribution.

Dinitz, Im, Lavastida, Moseley, Niaparast, Vassilvitskii. *Binary Search Trees with Distributional Predictions*. NeurIPS ’24