Introduction to Algorithms

Lecture 8

Last time

- Red-black trees
- Rotations
- Insertion
- Deletion
- AVL-Trees

Today

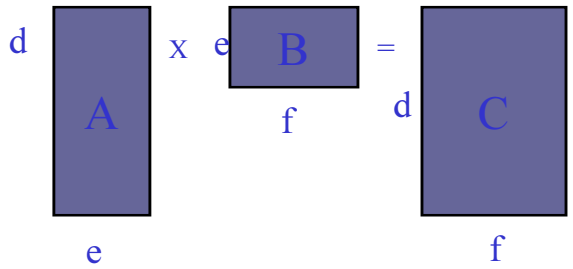
- Dynamic programming
  - Matrix Chain Multiplication
  - Longest common subsequence
  - Optimal BST

Dynamic Programming

- Dynamic programming is a metatechnique (not an algorithm) – like divide and conquer.
- Here, “programming” isn’t computer programming; word comes from table-based solution method.
- Divide & Conquer: break up into smaller problems
- Dynamic Programming: solve *all* smaller problems but only reuse optimal subproblem solutions.

Matrix Chain Multiplication - Review

- Recall how to multiply matrices.
- Given two matrices $\mathbf{A}_{(d \times e)}$ $\mathbf{B}_{(e \times f)}$



- The product is $\mathbf{C}[i, j] = \sum_{k=0}^{e-1} \mathbf{A}[i, k] \bullet \mathbf{B}[k, j]$
- Time is $O(def)$.

Matrix Chaining contd.

- What about multiplying multiple matrices?

$$\mathbf{A} = \mathbf{A}_1 \cdot \mathbf{A}_2 \cdot \mathbf{A}_3 \cdot \mathbf{A}_4 \cdot \mathbf{A}_5$$

- We know matrix multiplication is associative. Can we use this to help minimize calculations?
- Sure, consider $\mathbf{A}_{(5\times3)}$ $\mathbf{B}_{(3\times100)}$ $\mathbf{C}_{(100\times5)}$
- If we multiply $(\mathbf{A}\cdot\mathbf{B})\cdot\mathbf{C}$ we do 4000 operations
- But instead if we try $\mathbf{A}\cdot(\mathbf{B}\cdot\mathbf{C})$ we do 1575 operations

How do we parenthesize

- Brute force - try all ways to parenthesize

$$\mathbf{A} = \mathbf{A}_1 \cdot \mathbf{A}_2 \cdot \mathbf{A}_3 \cdot \cdots \cdot \mathbf{A}_n$$

- Find out which has smallest number of operations by doing multiplication, then pick the best one.
- But how many ways can we parenthesize these?
- Equivalent to the number of ways to make a binary tree with n nodes. This is $\Omega(2^n)$. see Catalan numbers.

Optimality of Subproblems

- Consider general question again, we want to parenthesize

$$\mathbf{A}_1 \cdot \mathbf{A}_2 \cdot \mathbf{A}_3 \cdot \ldots \cdot \mathbf{A}_n$$

- Now suppose somehow we new that the last multiplication we need to do was at the ith position.

$$(\mathbf{A}_1 \cdot \mathbf{A}_2 \cdot \ldots \cdot \mathbf{A}_i) \cdot (\mathbf{A}_{i+1} \cdot \ldots \cdot \mathbf{A}_n)$$

- Then the problem is the same as knowing the best way to parenthesize $(\mathbf{A}_1 \cdot \mathbf{A}_2 \cdot \ldots \cdot \mathbf{A}_i)$ and $(\mathbf{A}_{i+1} \cdot \ldots \cdot \mathbf{A}_n)$

Overlapping Subproblems

- We know how to break problem into smaller pieces, why doesn't divide and conquer work?
- Again suppose we know where the final multiply is and we want to generalize the cost with a recurrence, consider

$$N_{i,j} = \min_{i \le k < j} \left\{ N_{i,k} + N_{k+1,j} + d_i d_{k+1} d_{j+1} \right\}$$

- But notice these subproblems overlap.

Try Dynamic Programming

- Since subproblems overlap we can not use divide and conquer method.
- Instead work from the "bottom up."
- We know how to parenthesize one matrix, two matrices, we can build from there…

Matrix Chain Order

$n = length[p] - 1$

for $i = 1$ to n

  do $m[i, i] = 0$

for $l = 2$ to n

  do for $i = 1$ to $n - l + 1$

    do $j = i + l - 1$

      $m[i, j] = \infty$

      for $k = i$ to $j - 1$

        do $q = m[i, k] + m[k + 1, j] + p_{i-1} p_k p_j$

          if $q < m[i, j]$

            then $m[i.j] = q$

              $s[i, j] = k$

return m and s

What have we noticed

- Optimal solution depends on optimal subproblems
- Subproblems overlap.
- So a "top down" divide and conquer doesn't seem to work.
- Somehow we need to build up from bottom by remembering solution from subproblem.

Example

- Biologists need to measure how similar strands of DNA are to determine how closely related an organism is to another.
- They do this by considering DNA as strings of letters A,C,G,T and then comparing similarities in the strings.
- Formally they look at common subsequences in the strings.
- Example. X = AGTCAACGTT, Y=GTTCGACTGTG
- Both S = AGTG and S'=GTCACGT are subsequences
- How to do find these efficiently?

Example: Longest common subsequence (LCS)

Problem: Given two sequences $x[1..m]$ and $y[1..n]$, find a longest subsequence common to both.

$$
\begin{array}{lccccccc}
x: & A & B & C & B & D & A & B \\
 & & / & | & \backslash & & | & \\
y: & B & D & C & A & B & A &
\end{array}
$$

Brute-force algorithm: For every subsequence of x, check if it's a subsequence of y.

Worst-case running time: $\Theta(n2^m)$

(2^m subsequences of x to check; each check takes $\Theta(n)$ time)

Recursive Algorithm

Better way:

For now, compute only length, not actual sequence.

Define $c[i, j]$ = length of LCS of "prefixes" $x[1..i]$ and $y[1..j]$.

- Then $c[m, n]$ = length of LCS of x and y.

Theorem:

$$c[i, j] = \begin{cases} c[i-1, j-1]+1 & \text{if } x[i] = y[j], \\ \max\{c[i, j-1], c[i-1, j]\} & \text{otherwise.} \end{cases}$$

Proof

$$c[i,j] = \begin{cases} c[i-1, j-1] + 1 & \text{if } x[i] = y[j], \\ \max\{c[i, j-1], c[i-1, j]\} & \text{otherwise.} \end{cases}$$

Proof of case where $x[i] = y[j]$

Let $z[1..k]$ be LCS of $x[1..i]$ and $y[1..j]$

(i.e., $c[i, j] = k$).

Now, $z[k] = x[i]$ $(= y[j])$ (else could extend z by $x[i]$).

Proof

- Thus $z[1..k-1]$ is CS of $x[1..i-1]$ and $y[1..j-1]$.

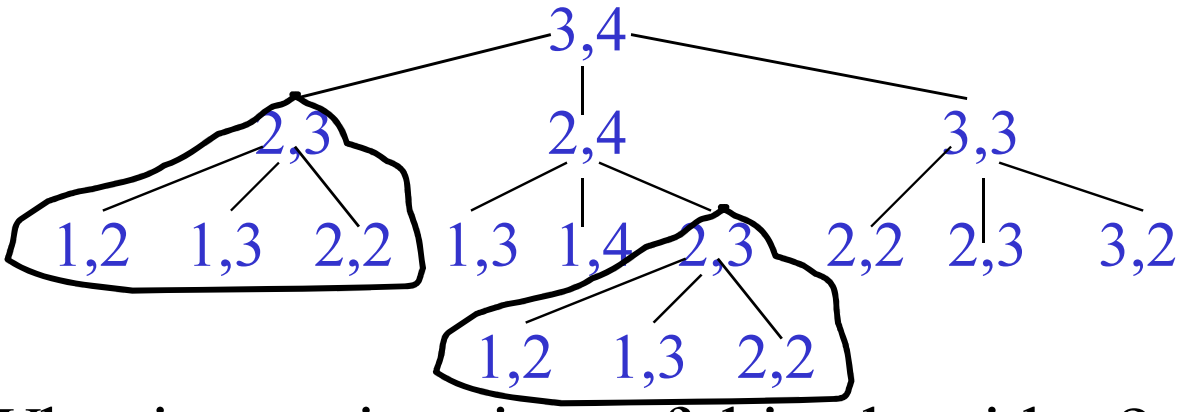
If $\exists$ a CS w longer than $z[1..k-1]$, $<w,x[i]>$ would be a CS longer than z.

Thus $z[1..k-1]$ is LCS of $x[1..i-1]$ and $y[1..j-1]$ (i.e., $c[i-1, j-1] = k - 1$).

Recursive Algorithm

Use recursive formulation directly.

Example (for length 3, length 4 sequence):



- What is running time of this algorithm? (Why?)
- How many distinct subproblems are there?
- One for each position in strings x, y, for $O(mn)$.
- Each encountered many times in recursion tree!

Dynamic Programming Algorithm

```
LCS-Length(X, Y)
m ← length[X]
n ← length[Y]
for i ← 1 to m
    do c[i, 0] ← 0
for j ← 0 to n
    do c[0, j] ← 0
for i ← 1 to m
     do for j ← 1 to n
            do if x_i = y_j
                  then c[i, j] ← c[i − 1, j − 1] + 1
                       b[i, j] ← "↖"
                  else if c[i − 1, j] ≥ c[i, j − 1]
                          then c[i, j] ← c[i − 1, j]
                               b[i, j] ← "↑"
                          else c[i, j] ← c[i, j − 1]
                               b[i, j] ← "←"
return c and b
```

Example

$X = \langle A, B, C, B \rangle$

$Y = \langle B, D, C, A, B \rangle$

Which table entries must be known to compute $c[i, j]$?

i	j	0	1	2	3	4	5
		y_j	B	D	C	A	B
0	x_i	0	0	0	0	0	0
1	A	0	0	0	0	1	1
2	B	0	1	1	1	1	2
3	C	0	1	1	2	2	2
4	B	0	1	1	2	2	3

Reconstructing the Sequence

```
Print-LCS(b,X,i,j)
if i = 0 or j = 0
    then return
if b[i,j] = "↖"
    then Print-LCS(b,X,i–1,j–1)
            print x_i
elseif b[i,j] = "↑"
    then Print-LCS(b,X,i–1,j)
else Print-LCS(b,X,i,j–1)
```

Dynamic Programming

Why can you apply it?

- Computational task must have recursive formulation.
- No cycles in formulation (usually, problem should be reduced to "smaller" problems).
- Total number of problem instances to be solved must be small, say N.
- Then running time is $O(N$ * time to compute recursion rule).

Example: Optimal BST (Sedgewick)

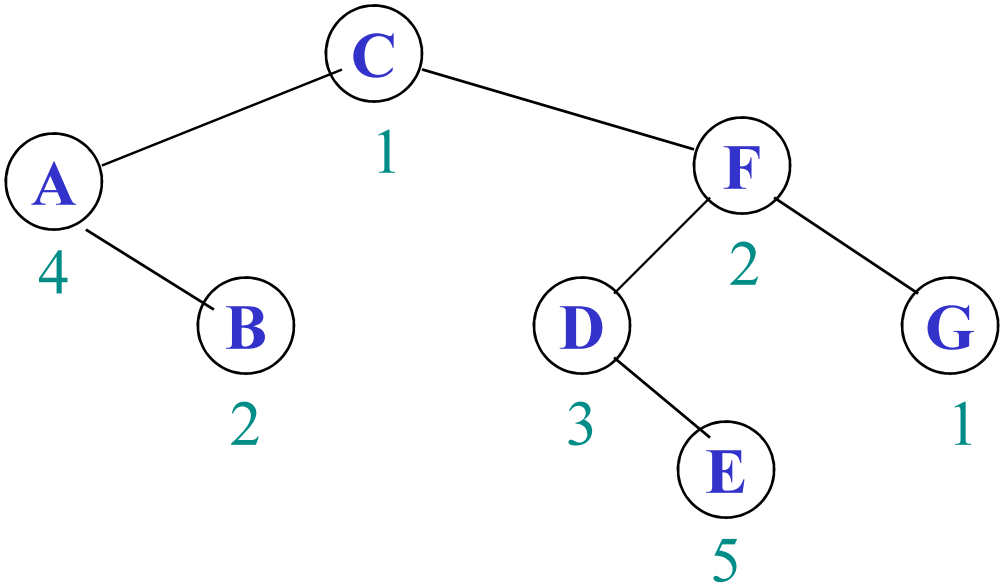
Suppose we are doing string searching, and know relative key frequencies

Example: spell; cc

Use Dynamic Programming to optimize BST.

Weighted Internal Path Length

Example BST, with search frequencies



Define "cost" of tree as frequency-weighted sum of node distance from root.

This is WIPL; here

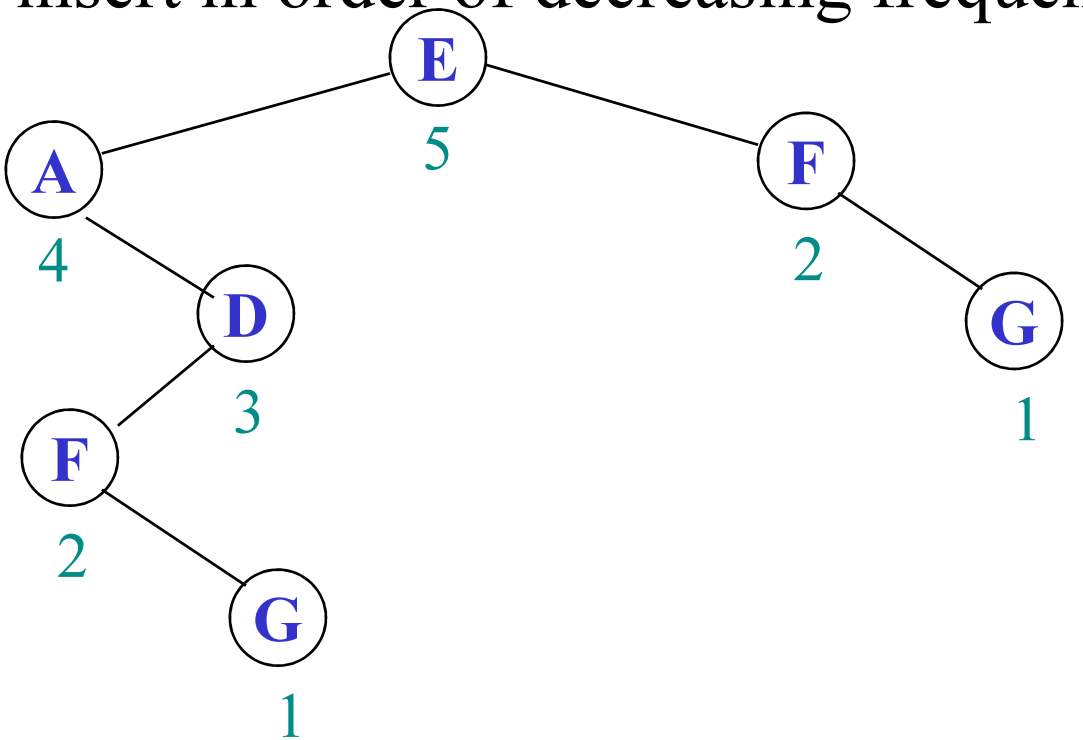
1*1 + 4*2 + 2*2 + 2*3 + 3*3 +1*3 + 5*4 = 51

Intuition

Want to put highest-frequency keys near root

But must balance against cost of increased path lengths

First try: insert in order of decreasing frequency



WIPL is

1*5 + 4*2 + 2*2 + 3*3 + 1*3 + 2*4 + 5*1 =42

Algorithm

Given: keys $K_1 < K_2 < \ldots < K_n$, $1 \leq i \leq n$ and frequencies f_i, $1 \leq i \leq n$.

Find BST that minimizes, over all keys, of frequency times distance from root (access cost).

Idea: for subsequences of length $1,2,3,\ldots,N\text{-}1$, Find Optimal BST for search subsequence.

How?

Algorithm

Idea:

- For each k, $i \le k \le i+j$ [$j+1$ is length]
- Place K_k at root of T
- Lookup optimal BST L of $K_i \ldots K_{k-1}$ (left)
- Lookup optimal BST R of $K_{k+1} \ldots K_{i+j}$ (right)
- WIPL(T) = WIPL(L) + WIPL(R) + $\sum_{i'=i}^{i+j} f_{i'}$

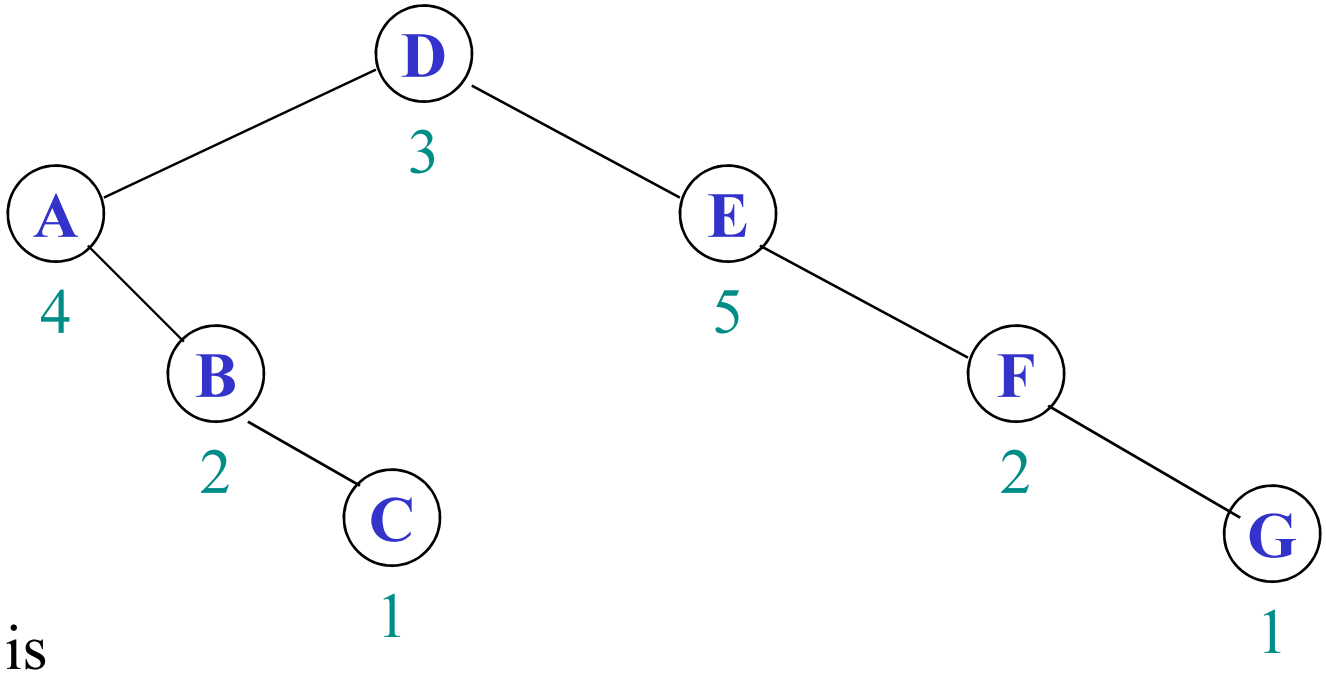
  so compare to current optimum and update

Optimal BST

Puts high-frequency keys near root,

But…keeps tree reasonably balanced!

$N = 7; f_k = 4,2,1,3,5,2,1$



WIPL is

1*3 + 4*2 + 5*2 + 2*3 + 2*3 +1*4 + 1*4 = 41

Optimal substructure

- A problem exhibits optimal substructure if an optimal solution contains optimal solutions to its subproblems.
- Build an optimal solution from optimal solutions to subproblems
- Example - Matrix-chain multiplication: An optimal parenthesization of $A_i A_{i+1} \ldots A_j$ that splits the product between A_k and A_{k+1} contains within it optimal solutions to the problem of parenthesizing $A_i A_{i+1} \ldots A_k$ and $A_{k+1} A_{k+2} \ldots A_j$.

Illustration of Optimal Substructure

Minimal
$Cost_A_{1..6} + Cost_A_{7..9} + p_0 p_6 p_9$

$A_1A_2A_3A_4A_5A_6A_7A_8A_9$

Suppose $((A_1A_2)(A_3((A_4A_5)A_6)))\ ((A_7A_8)A_9)$ is optimal

Then $(A_1A_2)\ (A_3((A_4A_5)A_6))$ must be optimal for $A_1A_2A_3A_4A_5A_6$

Otherwise, if $(A_1(A_2A_3))\ ((A_4A_5)A_6)$ is optimal for $A_1A_2A_3A_4A_5A_6$

Then $((A_1(A_2A_3))\ ((A_4A_5)A_6))\ ((A_7A_8)A_9)$ will be better than

$((A_1A_2)(A_3((A_4A_5)A_6)))\ ((A_7A_8)A_9)$

Recognizing subproblems

- Show a solution to the problem consists of making a choice. Making the choice leaves one or more subproblems to be solved.
- Suppose that for a given problem, the choice that leads to an optimal solution is available.
- Notice something in common with a greedy solution.

Excerise

- 15.1-3, 15.2-5, 15.3-1, 15.3-3, 15.4-2, 15.5-2, 15.5-3