Lecture 13: April 3, 2001

Today:

- Dynamic programming
  - Longest common subsequence
  - Optimal BST
  - (Memoizing)

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

Example: Longest common subsequence (LCS)

§16.2, 16.3

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

```
x: A B C B D A B
     /  |  \  |
y: B D C A B A
```

$\Rightarrow$ B C B A.

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

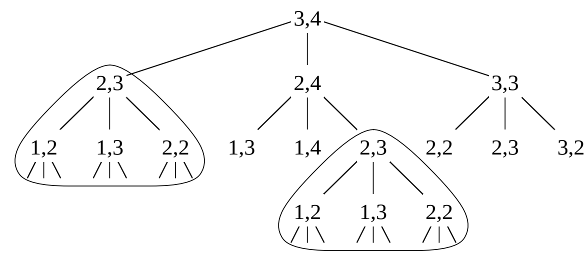
Theorem:

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

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 $x[i]$).
Thus $z[1\,..\,k-1]$ is CS of $x[1\,..\,i-1]$ and $y[1\,..\,j-1]$.
If $\exists$ a CS w longer than $z[1\,..\,k-1]$,
  $\langle w, x[i]\rangle$ 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

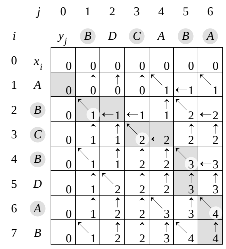
```
LCS-LENGTH(X, Y)
m ← length[X]
n ← length[Y]
for i ← 1 to m
      do c[i, 0] ← 0
for j ← 0 to n
      do c[0, j] ← 0
for i ← 1 to m
      do for j ← 1 to n
              do if x_i = y_j
                    then c[i, j] ← c[i − 1, j − 1] + 1
                         b[i, j] ← "↖"
                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

$X = \langle A, B, C, B, D, A, B\rangle$
$Y = \langle B, D, C, A, B, A\rangle$
Which table entries must be known to compute $c[i,j]$?

	j	0	1	2	3	4	5	6
i		y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
2	B	0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
3	C	0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
4	B	0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
5	D	0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
6	A	0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4
7	B	0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4

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$(b, X, i, j-1)$

Prints "BCBA"

Memoizing

Same idea, different implementation:
store optimal subanswers directly

MEMOIZED-LCS-LENGTH(X, Y)

$m \leftarrow$ *length*$[X]$
$n \leftarrow$ *length*$[Y]$
for $i \leftarrow 0$ to m
  do for $j \leftarrow 0$ to n
    do $c[i, j] \leftarrow -1$
return LOOKUP-LCS(m, n)

LOOKUP-LCS(i, j)

if $c[i, j] \geq 0$
  then return $c[i, j]$
if $x_i = y_j$
  then $c[i, j] \leftarrow$ LOOKUP-LCS$(i-1, j-1) + 1$
  else $c[i, j] \leftarrow \max($LOOKUP-LCS$(i-1, j)$,
    LOOKUP-LCS$(i, j-1))$
return $c[i, j]$

Dynamic Programming

When can you apply it?

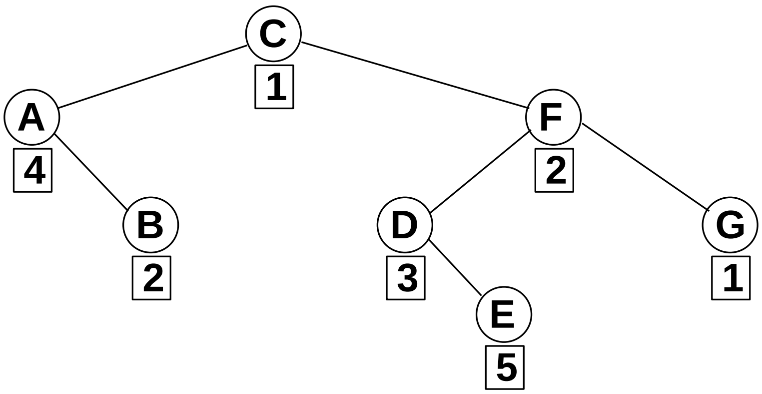
- Computational task must have recursive formulation.
- No cycles in formulation (usually, problem should be reduced to "smaller" problems).
- Total number of problem instances to be solved must be small, say N.
- Then running time is $O(N*$ time to compute recursion rule).

Example 2: 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

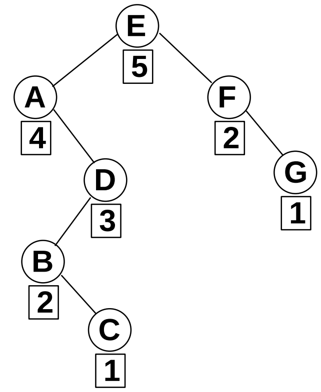


Define "cost" of tree as frequency-weighted
sum of node distances from root
This is WIPL; here

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

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

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

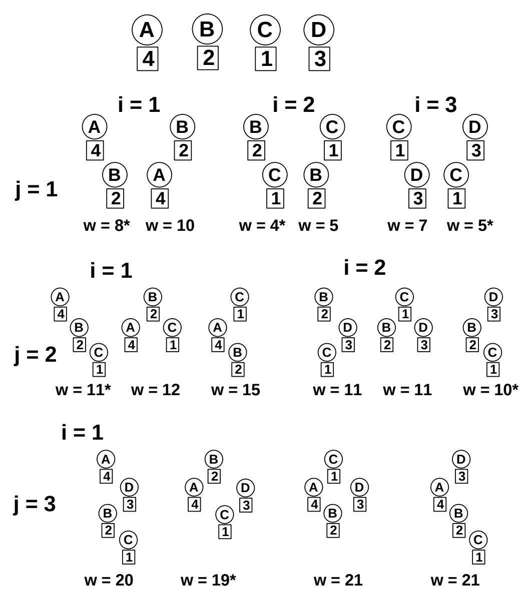
Can we do better?

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, ... N-1
Find optimal BST for each subsequence. How?

- For each $k, i \leq k \leq i + j$ [$j + 1$ 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)
- $\mathrm{WIPL}(T) = \mathrm{WIPL}(L) + \mathrm{WIPL}(R) + \Sigma_{i'=i}^{i+j} f_{i'}$ (why?) so compare to current optimum and update

Example (N = 4)



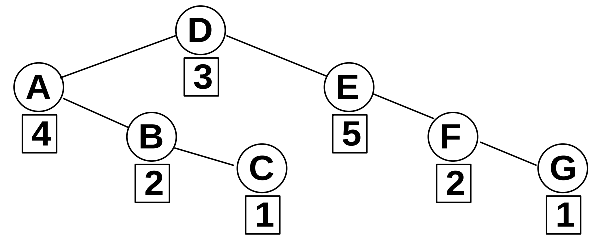
Which trees **are** enumerated?
Which trees are **not** enumerated?

Pseudocode

```
int f[1..N];              // search frequency of ith key
int cost[0..N+1][0..N+1]; // cost[a][b] = min WILP(f_a..f_b)
int i, j, k, t;

for ( i = 1; i <= N; i++ )
  for ( j = i+1; j <= N+1; j++ )
    cost[i][j] = MAXINT; // initial maximum costs

for ( i = 1; i <= N; i++ )
  cost[i][i] = f[i];     // cost of singleton OBST

for ( i = 1; i <= N+1; i++ )
  cost[i][i-1] = 0;      // handle nodes with 1 child

// invariant: min WILP of OBSTs length 1..j already computed
for ( j = 1; j <= N-1; j++ ) // for each BST of j+1 elements
  for ( i = 1; i <= N-j; i++ ) { // loop over first elements
    for ( k = i; k <= i+j; k++ ) { // minimize cost[i][i+j]
      t = cost[i][k-1] + cost[k+1][i+j]; // sum L + R costs
      if ( t < cost[i][i+j] ) {    // found better minimum for
        cost[i][i+j] = t;          //   tree of keys i .. i+j
        best[i][i+j] = k;          // store index of best root
        } // if
      } // for k
    for ( k = i; k <= i+j; k++ )   // total weight added to every
      cost[i][i+j] += f[k];        // cost, so not needed above
    } // for i
```


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

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

When to use dynamic programming?

Optimal substructure: optimal solution to problem instance contains optimal solutions to subinstances.
Overlapping subproblems: total number of **distinct** subproblems small compared to recursive run time.
(Alternative: “Memoizing,” a top-down approach.)