

Homework 5 Due Tuesday Oct 26

- CLRS 14-2.2 (rb tree black height)
- CLRS 14-2.3 (rb tree depth)
- CLRS 14-1 (point of maximum overlap)
- CLRS 15.1-4 (assembly line space requirement)

Chapter 15: Dynamic programming

Dynamic programming is a method for designing efficient algorithms for recursively solvable problems with the following properties:

1. **Optimal Substructure:** An optimal solution to an instance contains an optimal solution to its sub-instances.
2. **Overlapping Subproblems:** The number of subproblems is small so during the recursion same instances are referred to over and over again.

Four steps in solving a problem using the dynamic programming technique

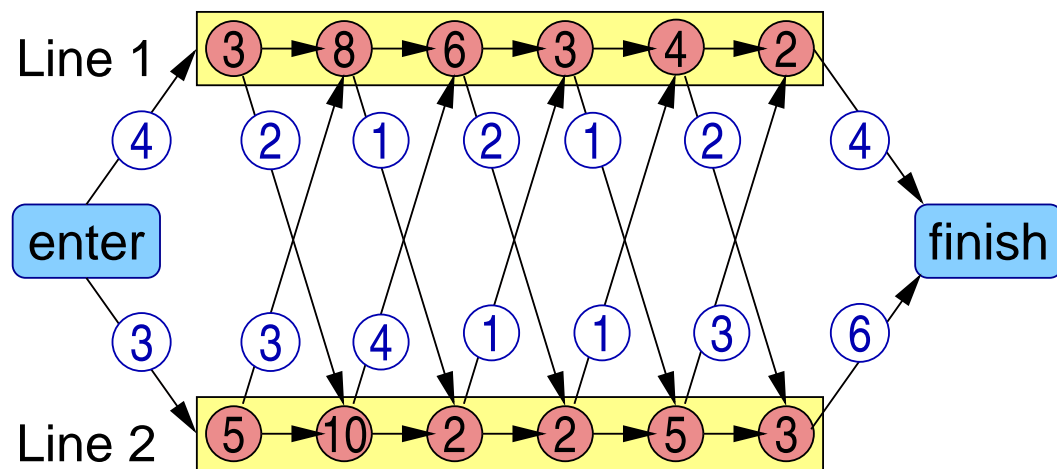
1. Characterize the structure of an optimal solution
2. Recursively define the value of an optimal solution
3. Compute the value of an optimal solution in a bottom-up fashion
4. Construct an optimal solution from computed information

The Problems to be Studied

1. Assembly-line Scheduling ... the problem of finding the best choices for stations in two assembly lines
2. Matrix-chain Multiplication ... the problem of finding the ordering of matrix-multiplication that minimizes the total number of scalar multiplications.
3. Longest Common Subsequence ... the problem of finding the longest sequence that appears commonly in a pair of strings.
4. Optimal Binary Tree ... Finding the arrangement of nodes that minimizes the average search time

I. Assembly-line Scheduling

Assume you own a factory for car assembling. The car you will be producing has n parts and the parts need to be put on the chassis in a fixed order. There are two different assembly lines. Each line consists of n stations, where for each i , $1 \leq i \leq n$, the i th station is for putting the i th part. The time required for a station varies. When a chassis leaves a station for the next part it is possible to move the chassis to the other line, but that takes extra time depending on which station the chassis is at the moment. Also, each line has a certain entry time and an exit time. What are the choice of the stations so as to minimize the production time?



How about testing all possible paths?

How about testing all possible paths?

*There are 2^n possible paths.
For large n exhaustive search
is not going to work.*

*There is an $O(n)$ -time
solution to this problem.*

*The trick is to find the fastest
path to each station.*

Mathematical Formulation

For each $i \in \{1, 2\}$ and for each j , $1 \leq j \leq n$, let $S_{i,j}$ denote the j th station in line i .

For each $i \in \{1, 2\}$, define the following quantities:

- e_i is the entry time into line i .
- x_i is the exit time from line i .
- For each j , $1 \leq j \leq n - 1$, $t_{i,j}$ is the time that it takes for moving from $S_{i,j}$ to $S_{3-i,j+1}$.
- For each j , $1 \leq j \leq n$, $a_{i,j}$ is the time required for station $S_{i,j}$.

Step 1: Characterizing structure of the optimal solution

To compute the fastest assembly time, we only need to know the fastest time to $S_{1,n}$ and the fastest time to $S_{2,n}$, including the assembly time for the n th part.

Then we choose between the two exiting points by taking into consideration the extra time required, x_1 and x_2 .

To compute the fastest time to $S_{1,n}$ we only need to know the fastest time to $S_{1,n-1}$ and to $S_{2,n-1}$. Then there are only two choices...

Step 2: A recursive definition of the values to be computed

For each $i \in \{1, 2\}$ and for each j , $1 \leq j \leq n$, let $f_i[j]$ be the fastest possible time to get to station $S_{i,j}$, including the assemble time at $S_{i,j}$.

Let f^* be the fastest time for the entire assembly. Then

$$f^* = \min(f_1[n] + x_1, f_2[n] + x_2).$$

For all j , $2 \leq j \leq n$, we have $f_1[j] = \min(f_1[j-1] + a_{1,j}, f_2[j-1] + t_{2,j-1} + a_{2,j})$ and $f_2[j] = \min(f_1[j-1] + t_{1,j-1} + a_{1,j}, f_2[j-1] + a_{2,j})$.

Step 3: Computing the fastest time

First, set $f_1[1] = e_1 + a_{1,1}$ and $f_2[1] = e_2 + a_{1,2}$.

Then, for $j \leftarrow 2$ to n , compute $f_1[j]$ as $\min(f_1[j-1] + a_{1,j}, f_2[j-1] + t_{2,j-1} + a_{1,j})$ and $f_2[j]$ as $\min(f_1[j-1] + t_{1,j-1} + a_{2,j}, f_2[j-1] + a_{2,j})$.

Finally, compute f^* as $\min(f_1[n] + x_1, f_2[n] + x_2)$.

Step 4: Computing the fastest path

For each $i \in \{1, 2\}$, and for each j , $2 \leq j \leq n$, compute as $l_i[j]$ as the choice made for $f_i[j]$ (whether the first or the second term gives the minimum). Also, compute the choice for f^* as l^* .

Then we have only to trace back the choices to find the fastest path.

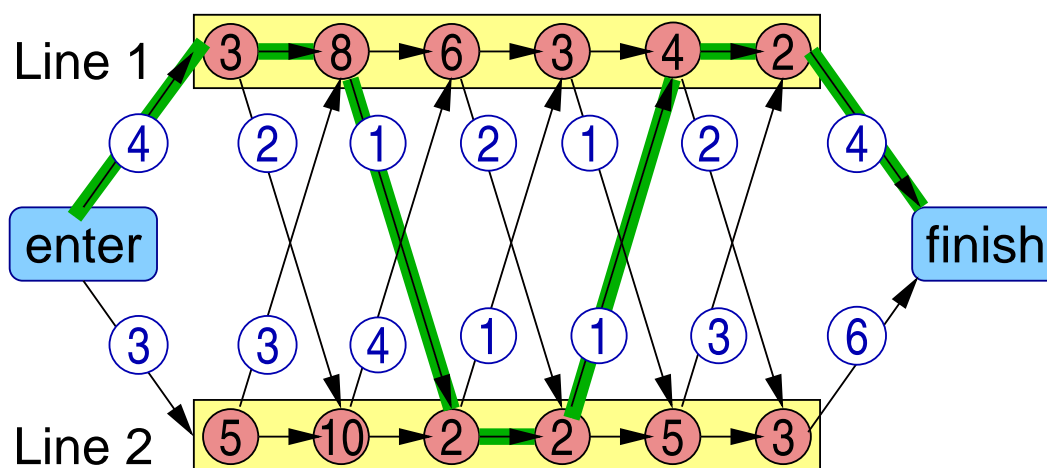
Fastest-Way(a, t, e, x, n)

```
1:  $f_1[1] \leftarrow e_1 + a_{1,1}$ 
2:  $f_2[1] \leftarrow e_2 + a_{1,2}$ 
3: for  $j \leftarrow 2$  to  $n$  do {
4:     if  $f_1[j-1] + a_{1,j} \leq f_2[j-1] + t_{2,j-1} + a_{1,j}$ 
5:     then {  $f_1[j] \leftarrow f_1[j-1] + a_{1,j}$ 
6:          $l_1[j] \leftarrow 1$  }
7:     else {  $f_1[j] \leftarrow f_2[j-1] + t_{2,j-1} + a_{1,j}$ 
8:          $l_1[j] \leftarrow 2$  }
9:     if  $f_2[j-1] + a_{2,j} \leq f_1[j-1] + t_{1,j-1} + a_{2,j}$ 
10:    then {  $f_2[j] \leftarrow f_2[j-1] + a_{2,j}$ 
11:         $l_2[j] \leftarrow 2$  }
12:    else {  $f_2[j] \leftarrow f_1[j-1] + t_{1,j-1} + a_{2,j}$ 
13:         $l_2[j] \leftarrow 1$  }
14: if  $f_1[n] + x_1 \leq f_2[n] + x_2$  then {
15:      $f^* \leftarrow f_1[n] + x_1$ 
16:      $l^* \leftarrow 1$  }
17: else {  $f^* \leftarrow f_2[n] + x_2$ 
18:      $l^* \leftarrow 2$  }
```

Example

| j | 1 | 2 | 3 | 4 | 5 | 6 |
|----------|---|----|----|----|----|----|
| $f_1[j]$ | 7 | 15 | 21 | 22 | 25 | 27 |
| $l_1[j]$ | | 1 | 1 | 2 | 2 | 1 |
| $f_2[j]$ | 8 | 18 | 18 | 20 | 25 | 28 |
| $l_2[j]$ | | 2 | 1 | 2 | 2 | 2 |

$$f^* = 31 \text{ and } l^* = 1$$



II. Matrix-Chain Multiplication

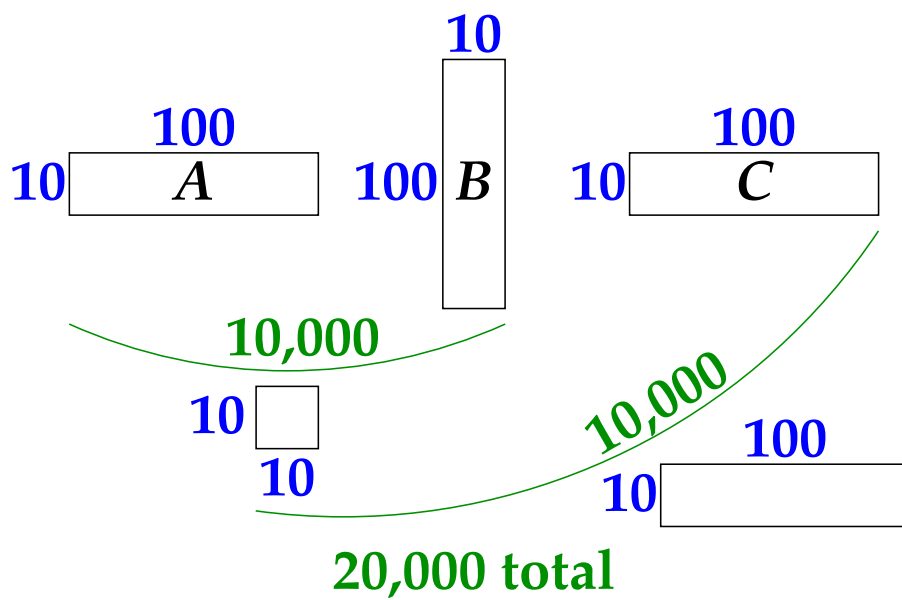
Suppose that we need to compute the product $M = A_1 \cdots A_n$ of matrices A_1, \dots, A_n . In the standard matrix multiplication, to compute the product of two matrices of dimension $p \times q$ and $q \times r$, pqr scalar multiplications are needed.

The multiplication over matrices is an **associative** operation. So, there are many different ways to compute the product. Use parentheses to describe the order. If the sizes of the matrix are not uniform, the cost of computing the product may be dependent on the order in which the matrices are multiplied.

The **matrix-chain multiplication problem** is the problem of, given a sequence of matrices, finding the order of multiplications that minimizes the total cost.

Example Suppose we need to compute ABC , where A is 10×100 , B is 100×10 , and C is 10×100

*How many operations for
 $A(BC)$?*



Parenthesization of Matrix Chain

A chain of matrices is **fully parenthesized** if it is either a single matrix or the product of two fully parenthesized matrix products.

*How many different fully
parenthesizations are there for
 $ABCD$?*

There are five:

$(A(B(CD))), (A((BC)D)),$
 $((A(BC))D), ((AB)(CD)),$
and $((AB)C)D).$

*Then how many are there for
 n matrices?*

The Number of Full Parenthesizations

For each $n \geq 1$, let $P(n)$ be the number of distinct full parenthesizations of a chain of n matrices. Then

$$P(n) = \begin{cases} 1 & \text{if } n = 1, \\ \sum_{k=1}^{n-1} P(k)P(n-k) & \text{if } n \geq 2. \end{cases}$$

Solving this, we obtain $P(n) = C(n-1)$, where

$$C(n) = \frac{1}{n+1} \binom{2n}{n}$$

Redefining the Problem

Using the concept of full parenthesization the problem can be redefined as follows:

Given a list $p = (p_0, p_1, \dots, p_n)$ of positive integers, compute the optimal-cost full-parenthesization of any chain (A_1, A_2, \dots, A_n) , such that for all i , $1 \leq i \leq n$, the dimension of the i^{th} matrix is dimension $p_{i-1} \times p_i$, where the cost is measured by the total number of scalar multiplications when the standard matrix multiplication is used.

Inefficiency of Brute-force Search

One cannot use **brute-force search** to solve this problem, because

$$C(n) = \frac{1}{n+1} \binom{2n}{n} = \Omega(4^n/n^{3/2}).$$

However, there is a solution with $O(n^3)$ running time.

Step 1: Characterization of the structure

The outermost pair of parentheses splits the matrix sequence into two. Suppose that the split is between A_1, \dots, A_k and A_{k+1}, \dots, A_n . Then to evaluate the product via this split, we compute $B(k) = A_1 \cdots A_k$ and $C(k) = A_{k+1} \cdots A_n$, then $B(k)C(k)$.

Suppose *the optimal cost of computing $B(k)$ and $C(k)$ is known for all $k, 1 \leq k \leq n - 1$.*

Then we can compute the optimal cost for the entire product by finding a k that minimizes

$$\begin{aligned} &\text{"the optimal cost for computing } B(k)\text{"} + \\ &\text{"the optimal cost for computing } C(k)\text{"} + \\ &p_0 p_k p_n. \end{aligned}$$

This suggests a **bottom-up** approach for computing the optimal costs.

Step 2: A recursive solution

For each i , $1 \leq i \leq n$, and each j , $1 \leq i \leq j \leq n$, let $m[i, j]$ be the optimal cost for computing $A_i \cdots A_j$.

Then for all i , $1 \leq i \leq n$, $m[i, i] = 0$ and for all i and j , $1 \leq i < j \leq n$, $m[i, j]$ is the minimum of

$$m[i, k] + m[k + 1, j] + p_{i-1}p_kp_j,$$

where $i \leq k \leq j - 1$

Step 3: Computing the optimal cost

1. For $i = 1, \dots, n$, set $m[i, i] = 0$.
2. For $\ell \leftarrow 2$ to n , and for all i and j such that $j - i + 1 = \ell$, compute $m[i, j]$.

Algorithm

```
1: for  $i \leftarrow 1$  to  $n$  do  $m[i, i] \leftarrow 0$ 
2: for  $\ell \leftarrow 2$  to  $n$  do
3:   for  $i \leftarrow 1$  to  $n - \ell + 1$  do {
4:      $j \leftarrow i + \ell - 1$ 
5:      $m[i, j] \leftarrow +\infty$ 
6:     for  $k \leftarrow i$  to  $j - 1$  do {
7:        $y \leftarrow m[i, k] + m[k + 1, j] + p_{i-1}p_kp_j$ 
8:       if  $y < m[i, j]$ 
9:         then {
10:            $m[i, j] \leftarrow y$ 
11:            $s[i, j] \leftarrow k$ 
12:         }
13:     }
```

*What is the running time of
this algorithm?*

*What is the running time of
this algorithm?*

There are $\Theta(n^3)$
combinations of i , j , and k so
the running time is $O(n^3)$.

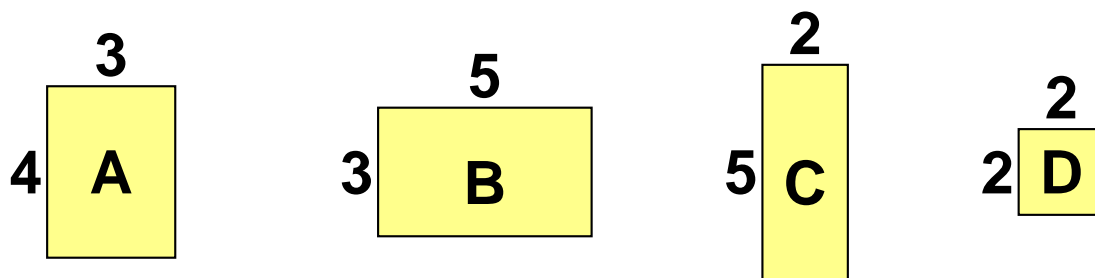
Step 4: Computing the optimal parenthesization

For each i , $1 \leq i \leq n - 1$, and for each j , $i + 1 \leq j \leq n$, let $s[i, j]$ be the smallest t , $i \leq t \leq j - 1$, such that

$$m[i, k] + m[k + 1, j] + p_{i-1}p_kp_j$$

is minimized at $k = t$.

When determining an m value memorize the choice as $s[i, j]$.



| j | 1 | 2 | 3 | 4 | i |
|---|---|----|----|----|---|
| | 0 | 60 | | | 1 |
| | | 0 | 30 | | 2 |
| | | | 0 | 20 | 3 |
| | | | | 0 | 4 |

Memoization

The same $O(n^3)$ efficiency can be achieved by keeping the recursive algorithm but remembering all the m -values that have been already computed.

Such a strategy is called **memoization**.

Matrix-Chain'

```
1: for  $i \leftarrow 1$  to  $n$  do  $m[i, i] \leftarrow 0$ 
2: for  $i \leftarrow 1$  to  $n - 1$  do
3:   for  $j \leftarrow i + 1$  to  $n$  do
4:      $m[i, j] \leftarrow +\infty$ 
5: Matrix-Chain-Memoized( $1, n$ )
```


Matrix-Chain-Memoized(c, d)

```
1: if  $m[c, d] \neq \infty$  return  $m[c, d]$ 
2:  $z \leftarrow \infty$ 
3: for  $i \leftarrow c$  to  $d - 1$  do {
4:      $u \leftarrow \text{Matrix-Chain-Memoized}(c, i)$ 
5:      $v \leftarrow \text{Matrix-Chain-Memoized}(i + 1, d)$ 
6:      $w \leftarrow u + v + p_{c-1}p_i p_d$ 
7:     if  $w < z$  then {
8:          $z \leftarrow w$ 
9:          $s[i, j] \leftarrow i$ 
10:    }
11: }
12:  $m[c, d] \leftarrow z$ 
```

Once all the entries have been computed, the optimal parenthesization can be recovered from the s -table

Print-Chain(i, j)

- 1: \triangleright print the parenthesization for $A_i \cdots A_j$
- 2: **Print**(" (")
- 3: **Print-Chain**($i, s[i, j]$)
- 4: **Print-Chain**($s[i, j] + 1, j$)
- 5: **Print**(") ")

III. Longest Common Subsequence

Let $Z = \langle z_1, z_2, \dots, z_k \rangle$ and $X = \langle x_1, x_2, \dots, x_m \rangle$ be strings over an alphabet. We say that Z is a **subsequence** of X if Z can be generated by striking out some (or none) elements from X .

For example, $\langle b, c, d, b \rangle$ is a subsequence of $\langle a, b, c, a, d, c, a, b \rangle$.

The **longest common subsequence problem (LCS)** is the problem of finding, given two sequences $X = \langle x_1, x_2, \dots, x_m \rangle$ and $Y = \langle y_1, y_2, \dots, y_n \rangle$, a maximum-length common subsequence of X and Y .

Step 1: Characteristics of the problem

Brute-force search for LCS requires exponentially many steps since there are $\sum_{i=1}^m \binom{n}{i}$ candidate subsequences.


The **optimal-substructure** of LCS

For a sequence $Z = \langle z_1, z_2, \dots, z_k \rangle$ and i , $1 \leq i \leq k$, let Z_i denote the prefix of Z having length i , namely, $Z_i = \langle z_1, z_2, \dots, z_i \rangle$.

Theorem A Let $X = \langle x_1, x_2, \dots, x_m \rangle$ and $Y = \langle y_1, y_2, \dots, y_n \rangle$.

1. If $x_m = y_n$, then an LCS of X_{m-1} and Y_{n-1} can be constructed by appending x_m ($= y_n$) to an LCS of X and Y .
2. If $x_m \neq y_n$, then an LCS of X and Y is either an LCS of X_{m-1} and Y or an LCS of X and Y_{n-1} .

Proof (1) Suppose x_m and y_n are the same symbol, say σ . Take an LCS Z of X and Y . Generation of Z **should need either x_m or y_n** . O.w., appending σ to Z would make a longer common sequence. If necessary, modify the production of Z from X (from Y) so that its last element is x_m (y_n). Then Z is a common subsequence W of X_{m-1} and Y_{n-1} followed by a σ . By the maximality of Z , W should be an LCS.

(2) If $x_m \neq y_n$, then for any LCS Z of X and Y , generation of Z **cannot use both x_m and y_n** . So, Z is either an LCS of X and Y_{n-1} or an LCS of X_{m-1} and Y . 

Step 2: A recursive definition

If $x_m = y_n$, then append x_m to an LCS of X_{m-1} and Y_{n-1} . Otherwise, compare an LCS of X and Y_{n-1} and an LCS of X_{m-1} and Y and pick the longer.

Let $c[i, j]$ be the length of an LCS of X_i and Y_j . We get the recurrence:

$$c[i, j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0, \\ c[i - 1, j - 1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j, \\ \max(c[i, j - 1], c[i - 1, j]) & \text{if } i, j > 0 \text{ and } x_i \neq y_j. \end{cases}$$

Let $b[i, j]$ be the choice made for (X_i, Y_j) .
With the b -table we can reconstruct an LCS.

| i | j | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|-----|-------------------|-------|-------------------|-----|-------------------|-----|-------------------|-------------------|
| | | y_j | \textcircled{B} | D | \textcircled{C} | A | \textcircled{B} | \textcircled{A} |
| 0 | x_i | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | A | 0 | 0 | 0 | 0 | 1 | 1 | 1 |
| 2 | \textcircled{B} | 0 | 1 | 1 | 1 | 1 | 2 | 2 |
| 3 | \textcircled{C} | 0 | 1 | 1 | 2 | 2 | 2 | 2 |
| 4 | \textcircled{B} | 0 | 1 | 1 | 2 | 2 | 3 | 3 |
| 5 | D | 0 | 1 | 2 | 2 | 2 | 3 | 3 |
| 6 | \textcircled{A} | 0 | 1 | 2 | 2 | 3 | 3 | 4 |
| 7 | B | 0 | 1 | 2 | 2 | 3 | 4 | 4 |

Here numeric entries are c -values and arrows are b -values.

The LCS problem possesses the other characteristic of dynamic programming:

- **overlapping subproblems:**

For all i, j, i' , and j' , such that $i' \leq i \leq j \leq j'$, the value of $c[i, j]$ will be referenced to in the evaluation of $c[i', j']$.

4. Optimal Binary Trees

Suppose that we want to arrange a sorted array of n elements, k_1, \dots, k_n , in an n -node binary search tree. Each k_i is associated with a value p_i , the frequency that k_i is searched for. Elements not in the list may be searched for. Let d_0, \dots, d_n be the $n + 1$ regions that are “in-between” the n keys. These are represented by **nil**’s when the n keys are arranged in a tree.

To these “dummy” keys frequencies q_0, \dots, q_n are assigned. The sum of all the $2n + 1$ frequencies is 1.

Suppose that the cost of search is the number of nodes visited. Our goal is to find a binary tree that minimizes the average search cost defined as follows:

Let T be an n -node binary tree that holds the $2n + 1$ keys, k_1, \dots, k_n and d_0, \dots, d_n . Then the average search time on T is

$$\sum_{i=1}^n p_i(\text{depth}_T(k_i) + 1) + \sum_{i=0}^n q_i(\text{depth}_T(d_i) + 1).$$

This is equal to

$$1 + \sum_{i=1}^n p_i \text{depth}_T(k_i) + \sum_{i=0}^n q_i \text{depth}_T(d_i).$$

*How good is examining all
possible binary search trees of
 n nodes?*

That should be really bad...

*That's right. The number of
 n -node binary trees is $C(n)$.*

Characterization

If k_r is chosen as the root, then we arrange $d_0, k_1, d_1, \dots, d_{r-2}, k_{r-1}, d_{r-1}$ as the left subtree of k_r and arrange $d_r, k_{r+1}, d_{r+1}, \dots, d_{n-1}, k_n, d_n$ as the right subtree of k_r . So, for each of the two groups, we need to find the “best” arrangement.

The Table

For each i , $0 \leq i \leq n$, and for each j , $i \leq j \leq n$, let $S[i, j]$ be the optimal average search cost for arranging the nodes $d_i, k_{i+1}, d_{i+1}, \dots, d_{j-1}, k_j, d_j$ in a binary tree, where the cost for searching the rest of the nodes is considered to be 0.

Then for all i , $0 \leq i \leq n$, $S[i, i] = q_i$.

Recursion

For all i and j , $0 \leq i < j \leq n$, $S[i, j]$ is the minimum of

$$S[i, t] + S[t + 1, j] + p_t + L + R,$$

where t ranges between i and $j - 1$, L and R are respectively the sum of the frequencies to the left and to the right of p_t , i.e.,

$$L = \sum_{m=i}^{t-1} p_m + \sum_{m=i-1}^{t-1} q_m$$

and

$$R = \sum_{m=t+1}^j p_m + \sum_{m=t}^j q_m.$$

So,

$$L + R + p_t = \sum_{m=i}^j p_m + \sum_{m=i-1}^j q_m.$$

The Dynamic Programming Algorithm

For each ℓ , $1 \leq \ell \leq n + 1$, for each i ,
 $0 \leq i \leq n - \ell + 1$, compute $S[i, i + \ell]$ as the
minimum of

$$S[i, t] + S[t + 1, j] + \left(\sum_{m=i}^j p_m + \sum_{m=i-1}^j q_m \right).$$

Record in $C[i, i + \ell]$ the t that gives the
minimum.