Algorithm Design Techniques part I

Divide-and-Conquer.
Dynamic Programming

Some Algorithm Design Techniques

- Top-Down Algorithms: Divide-and-Conquer
- Bottom-Up Algorithms: Dynamic Programming
- Brute-Force and Greedy Algorithms
- Backtracking Algorithms
- Local Search Algorithms

Divide and Conquer

- Divide-and-conquer algorithms:
  - Solve a given problem, by dividing it into one or more subproblems each of which is similar to the given problem.
  - Each subproblem is solved independently.
  - Finally, the solutions to the subproblems are combined in order to obtain the solution to the original problem.
  - Often implemented using recursion (note that not all recursive functions are divide-and-conquer algorithms)
  - Generally, the subproblems solved by a divide-and-conquer algorithm are *non-overlapping*.

Divide and Conquer

- *Divide and conquer* method for algorithm design:
  - **Divide**: If the input size is too large to deal with in a straightforward manner, divide the problem into two or more disjoint subproblems
  - **Conquer**: Use divide and conquer recursively to solve the subproblems
  - **Combine**: Take the solutions to the subproblems and "merge" these solutions into a solution for the original problem

Integer Multiplication

- Algorithm: Multiply two n-bit integers x and y.
  - Divide step: Split x and y into high-order and low-order bits

$$m = \lfloor n/2 \rfloor$$

x			y	
a	b		c	d

  - We can then define $x \times y$ by multiplying the parts and adding:

$$(10^m a + b)(10^m c + d) = 10^{2m} ac + 10^m (bc + ad) + bd$$

  - So, $T(n) = 4T(n/2) + n$, which implies $T(n)$ is $O(n^2)$.
  - But that is no better than the algorithm we learned in grade school.

Integer Multiplication (2)

MULTIPLY(x, y, n)

if $n = 1$
  then return $x \cdot y$
  else $m \leftarrow \lceil n/2 \rceil$
    $a \leftarrow \lfloor x/10^m \rfloor$
    $b \leftarrow x \mod 10^m$
    $d \leftarrow \lfloor y/10^m \rfloor$
    $c \leftarrow y \mod 10^m$
    $e \leftarrow$ MULTIPLY(a, c, m)
    $f \leftarrow$ MULTIPLY(b, d, m)
    $g \leftarrow$ MULTIPLY(b, c, m)
    $h \leftarrow$ MULTIPLY(a, d, m)
return $10^{2m}e + 10^m(g + h) + f$

An Improved Integer Multiplication Algorithm

- Algorithm: Multiply two n-bit integers x and y.
  - Divide step: Split x and y into high-order and low-order bits

$$m = \lfloor n/2 \rfloor$$

x

a	b

y

c	d

  - Observe that there is a different way to multiply parts:

$$ac + bd - (a-b)(c-d) = bc + ad$$

  - So, $T(n) = 3T(n/2) + n$, which implies $T(n)$ is $O(n^{\log_2 3})$, by the Master Theorem.
  - Thus, $T(n)$ is $O(n^{1.585})$
  - This algorithm is superior to the elementary school algorithm for $n > 500$

Improved Integer Multiplication (2)

FASTMULTIPLY(x, y, n)

if $n = 1$
  then return $x \cdot y$
  else $m \leftarrow \lceil n/2 \rceil$
    $a \leftarrow \lfloor x/10^m \rfloor$
    $b \leftarrow x \mod 10^m$
    $d \leftarrow \lfloor y/10^m \rfloor$
    $c \leftarrow y \mod 10^m$
    $e \leftarrow$ MULTIPLY(a, c, m)
    $f \leftarrow$ MULTIPLY(b, d, m)
    $g \leftarrow$ MULTIPLY$(a - b, c - d, m)$
return $10^{2m}e + 10^m(e + f - g) + f$

Exponentiation

SlowPower(a, n)

for $i \leftarrow 2$ **to** n
    do $x \leftarrow x \cdot a$
return x

FastPower(a, n)

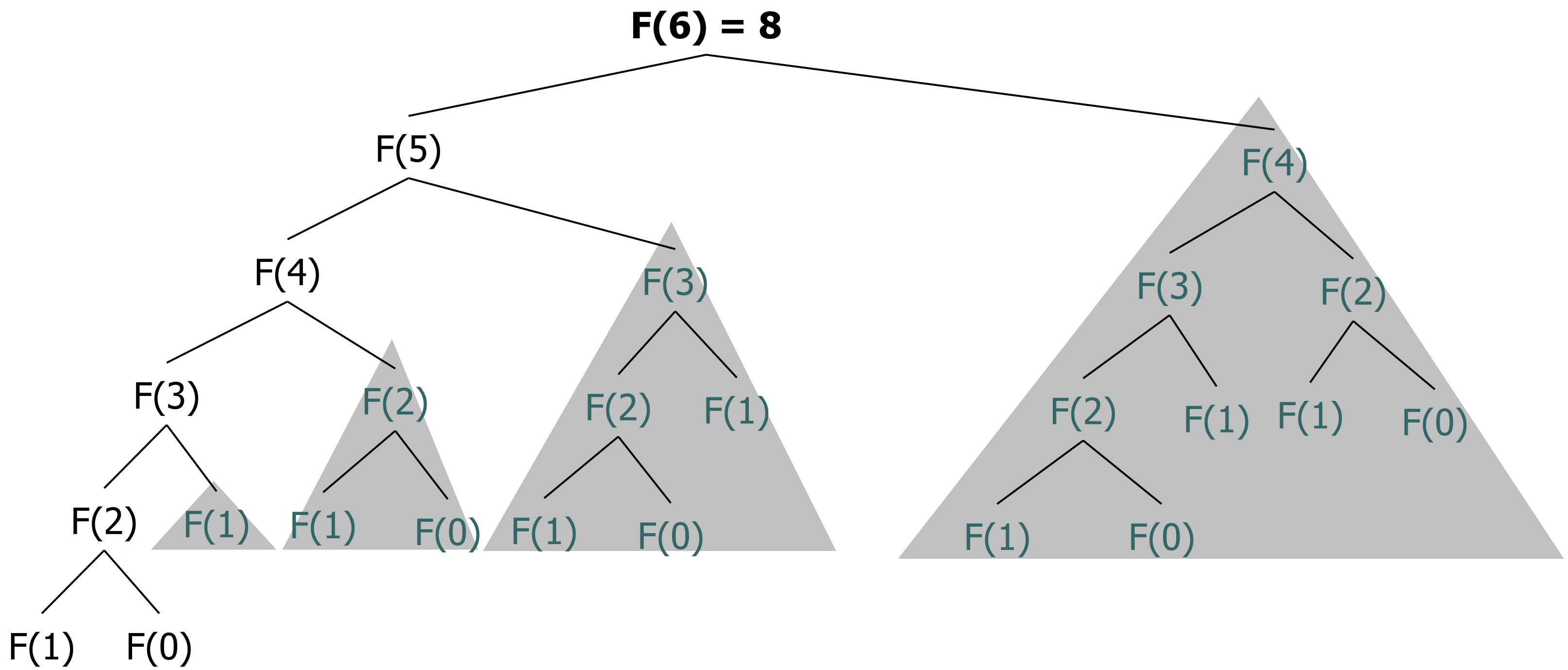
if $n = 1$
  then return a
  else $x \leftarrow$ FastPower($a, \lfloor n/2 \rfloor$)
    if n is even
      then return $x \cdot x$
      else return $x \cdot x \cdot a$

- Compare the two algorithms
  - How many steps for each algorithm?
  - How much space?

Fibonacci Numbers

- $F_n = F_{n-1} + F_{n-2}$
- $F_0 = 0,\ F_1 = 1$
  - 0, 1, 1, 2, 3, 5, 8, 13, 21, 34 …
- Straightforward recursive procedure is slow!
- Why? How slow?
- We can see that by drawing the recursion tree

Fibonacci Numbers (2)



- We keep calculating the same value over and over!

Fibonacci Numbers (3)

- How many summations are there?
- Golden ratio $\frac{F_{n+1}}{F_n} \approx \phi = \frac{1+\sqrt{5}}{2} \approx 1.61803...$
- Thus $\boldsymbol{F_n} \approx 1.6^n$
- Our recursion tree has only 0s and 1s as leaves, thus we have $\approx 1.6^n$ summations
- Running time is *exponential!*

Fibonacci Numbers (4)

- We can calculate F_n in *linear* time by remembering solutions to the solved subproblems – *dynamic programming*
- Compute solution in a bottom-up fashion
- Trade space for time!
  - In this case, only two values need to be remembered at any time (probably less than the depth of your recursion stack!)

```
Fibonacci(n)
   F_0 ← 0
   F_1 ← 1
   for i ← 1 to n do
      F_i ← F_{i-1} + F_{i-2}
```

Dynamic Programming

- *Divide-and-conquer* algorithms partition the problem into *independent subproblems*, solve the subproblems recursively, and then combine the solutions to solve the original problem.
- Dynamic programming:
  - Applicable when the <u>*subproblems are not independent*</u>.
  - Every subproblem is solved only once and the result stored in a table for avoiding the work of re-computing it. Consequence: there must be only relatively few subproblems for the table to be efficiently computable.
  - Allows an exponential-time algorithm to be transformed to a polynomial-time algorithm.
  - The name ‘dynamic programming’ refers to computing the table.

Dynamic Programming in Optimization

- Optimization problems:
  - Can have many possible solutions.
  - Each solution has a value and the task is to find the solution with the optimal (minimal or maximal) value
- Development of a dynamic programming algorithm involves four steps:
  - Characterize the structure of an optimal solution.
  - Recursively define the value of an optimal solution.
  - Compute the value of an optimal solution in a bottom-up fashion.
  - Construct the optimal solution.

Longest Common Subsequence

- Two text strings are given: X and Y
- There is a need to quantify how similar they are:
  - Comparing DNA sequences in studies of evolution of different species
  - Spell checkers
- One of the measures of similarity is the length of a Longest Common Subsequence (LCS)

LCS: Definition

- Z is a subsequence of X, if it is possible to generate Z by skipping some (possibly none) characters from X
- For example: X ="CEIIVVC", Y="EIVCV", $LCS(X,Y)$ = "EIVC" or "EIVV"
- To solve LCS problem we have to find "skips" that generate $LCS(X,Y)$ from X, and "skips" that generate $LCS(X,Y)$ from Y

LCS: Optimal Substructure

- We make Z to be empty and proceed from the ends of X_m="$x_1 x_2 \ldots x_m$" and Y_n="$y_1 y_2 \ldots y_n$"
  - If $x_m = y_n$, append this symbol to the beginning of Z, and find optimally $LCS(X_{m-1}, Y_{n-1})$
  - If $x_m \neq y_n$,
    - Skip either a letter from X
    - or a letter from Y
    - Decide which decision to do by comparing $LCS(X_m, Y_{n-1})$ and $LCS(X_{m-1}, Y_n)$
  - "Cut-and-paste" argument

LCS: Reccurence

- The algorithm could be easily extended by allowing more "editing" operations in addition to *copying* and *skipping* (e.g., changing a letter)
- Let $\boldsymbol{c[i, j]} = \boldsymbol{LCS(X_i, Y_j)}$

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

- Observe: conditions in the problem restrict sub-problems (What is the total number of sub-problems?)

LCS: Compute the Optimum

```
LCS(X, Y)
m ← length[X]
n ← length[Y]
for i ← 1 to m                                    ▷ initialize column 0
    do c[i, 0] ← 0
       b[i, 0] ← SKIPX
for j ← 0 to n                                    ▷ initialize row 0
    do c[0, j] ← 0
       b[0, j] ← SKIPY
for i ← 1 to m                                    ▷ fill rest of table
    do for j ← 1 to n
           do if x_i = y_j                        ▷ take x_i (=y_j) for LCS
                 then c[i, j] ← c[i − 1, j − 1] + 1
                      b[i, j] ← ADDXY             ▷ ADDXY ≡ ↖
              elseif c[i − 1, j] ≥ c[i, j − 1]    ▷ x_i ∉ LCS
                 then c[i, j] ← c[i − 1, j]
                      b[i, j] ← SKIPX             ▷ SKIPX ≡ ↑
              else c[i, j] ← c[i, j − 1]          ▷ y_i ∉ LCS
                      b[i, j] ← SKIPY             ▷ SKIPY ≡ ←
return c, b
```

LCS Example

	Y:	0	1	2	3	4	5	6	7	8 $=n$
			A	B	D	C	B	A	B	C
0		0	0	0	0	0	0	0	0	0
1	A	0	1	1	1	1	1	1	1	1
2	D	0	1	1	2	2	2	2	2	2
3	B	0	1	2	2	2	3	3	3	3
X: 4	C	0	1	2	2	3	3	3	3	4
5	D	0	1	2	3	3	3	3	3	4
6	B	0	1	2	3	3	4	4	4	4
7	A	0	1	2	3	3	4	5	5	5
$m =$ 8	B	0	1	2	3	3	4	5	6	6

LCS: Getting the sequence

GETLCS(X, Y, b)

$LCS \leftarrow \langle \rangle$
$i \leftarrow length[X]$
$j \leftarrow length[Y]$
while $i \neq 0 \land j \neq 0$
    do if $b[i, j] =$ ADDXY
        then add x_i (or y_j) to front of LCS
            $i \leftarrow i - 1$
            $j \leftarrow j - 1$
      elseif $b[i, j] =$ SKIPX
        then $i \leftarrow i - 1$
        else $j \leftarrow j - 1$ $\triangleright\ b[i, j] =$ SKIPY
return LCS

Matrix-Chain Multiplication

- Suppose we like to multiply a whole sequence of n matrices:

$$A_1 \times A_2 \times \ldots \times A_n$$

- Multiplying $p \times q$ matrix A with $q \times r$ matrix B takes $p \times q \times r$ scalar multiplications:

```
MatrixMultiply(A, B)

if columns[A] ≠ rows[B]
    then error "incompatible dimensions"
    else  for i ← 1 to rows[A]
              do for j ← 1 to columns[B]
                     do C[i, j] ← 0
                        for k ← 1 to columns[A]
                            do C[i, j] ← C[i, j] + A[i, k] × B[k, j]
          return C
```

- However, depending dimensions of the matrices, a different order might need significantly fewer multiplications.

Matrix-Chain Multiplication Example

- Suppose the dimensions for matrices $\mathbf{A_1, A_2, A_3, A_4}$ are:

$\mathbf{A_1}$: 15×5 $\mathbf{A_2}$: 5×10 $\mathbf{A_3}$: 10×20 $\mathbf{A_4}$: 20×25

Parenthesization:	**Number of scalar multiplications:**
$((A_1 \times A_2) \times A_3) \times A_4$	$15 \times 5 \times 10 + 15 \times 10 \times 20 + 15 \times 20 \times 25 = 11250$
$(A_1 \times A_2) \times (A_3 \times A_4)$	$15 \times 5 \times 10 + 10 \times 20 \times 25 + 15 \times 10 \times 25 = 13250$
$(A_1 \times (A_2 \times A_3)) \times A_4$	$5 \times 10 \times 20 + 15 \times 5 \times 20 + 15 \times 20 \times 25 = 10000$
$A_1 \times ((A_2 \times A_3) \times A_4)$	$5 \times 10 \times 20 + 5 \times 20 \times 25 + 15 \times 5 \times 25 = 5375$
$A_1 \times (A_2 \times (A_3 \times A_4))$	$10 \times 20 \times 25 + 15 \times 10 \times 25 + 15 \times 5 \times 25 = 8125$

- Problem: find a sequence $\langle A_1, \ldots, A_n \rangle$ of n matrices with dimensions $p_{i-1} \times p_i$, for $1 \le i \le n$, a parenthesization that minimizes the number of scalar multiplications

A Naive Algorithm

- For a single matrix, we have only one parenthesization. For a sequence of n matrices, we can split it between the k-th and $(k+1)$-st matrices and parenthesize the subsequences recursively. Thus for the number $P(n)$ of parenthesizations of n matrices we get:

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

- The solution of this recurrence is : $P(n) = C(n-1)$

$C(n)$ is the n-th Catalan number $\longrightarrow$ $C(n) = 1/(n+1)\binom{2n}{n}$

By applying Stirling's formula $\longrightarrow$ $= \Omega(4^n / n^{3/2})$

- Thus the number of solutions is exponential in n, so an exhaustive search for the optimal solution will quickly fail.

A Recursive Solution

- Let $\boldsymbol{m(i, j)}$ be the *minimum* number of multiplications necessary to compute $\prod_{k=i}^{j} A_k$
- Key observations:
  - The outermost parenthesis partition the chain of matrices $(\boldsymbol{i, j})$ at some $\boldsymbol{k}$, $(\boldsymbol{i \le k < j})$:

$$\boldsymbol{(A_i \ldots A_k)(A_{k+1} \ldots A_j)}$$

  - The optimal parenthesization of matrices $(\boldsymbol{i, j})$ has optimal parenthesizations on either side of $\boldsymbol{k}$: for matrices $(\boldsymbol{i, k})$ and $(\boldsymbol{k}+1, \boldsymbol{j})$

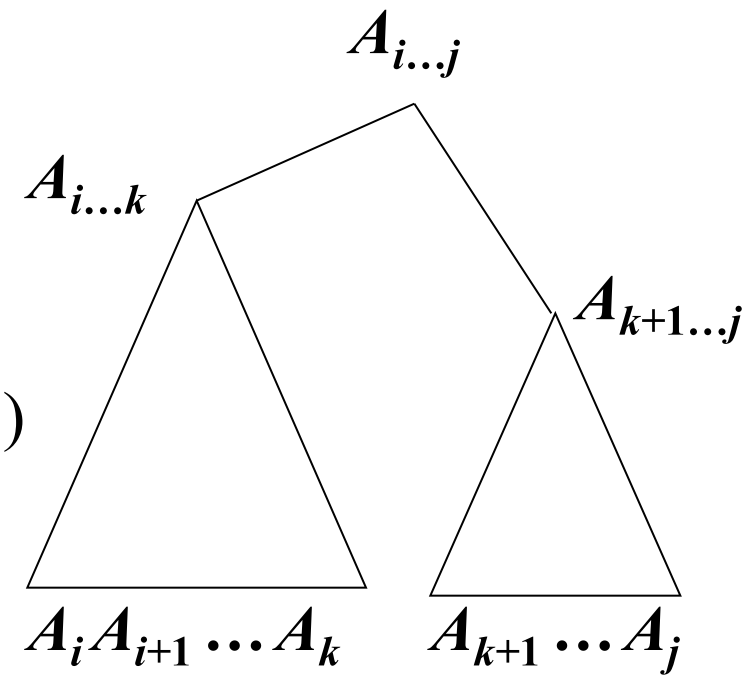
A Recursive Solution

- We try out all possible $\boldsymbol{k}$.
  Recurrence:

$$\begin{cases} m[i,i] = 0 & \text{if } i = j \\ m[i,j] = \min\limits_{i \le k \le j} \{ m[i,k] + m[k+1,j] + p_{i-1} p_k p_j & \text{if } i < j \end{cases}$$

- A direct recursive implementation is exponential – there is a lot of duplicated work (why?)

- But there are only $\binom{n}{2} + n = \Theta(n^2)$ different subproblems $(\boldsymbol{i, j})$, where $1 \le \boldsymbol{i} \le \boldsymbol{j} \le \boldsymbol{n}$



Computing the Optimal Cost

- The idea of dynamic programming is, rather than computing m recursively, computing it bottom-up: a recursive computation takes exponential time, a bottom-up computation in the order of n^3.
- Let $s[i, k]$ (for 'split') be the value of k such that

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

- The entries $s[i, j]$ are the values of k for an optimal parenthesization of $A_i \times \ldots \times A_j$ into

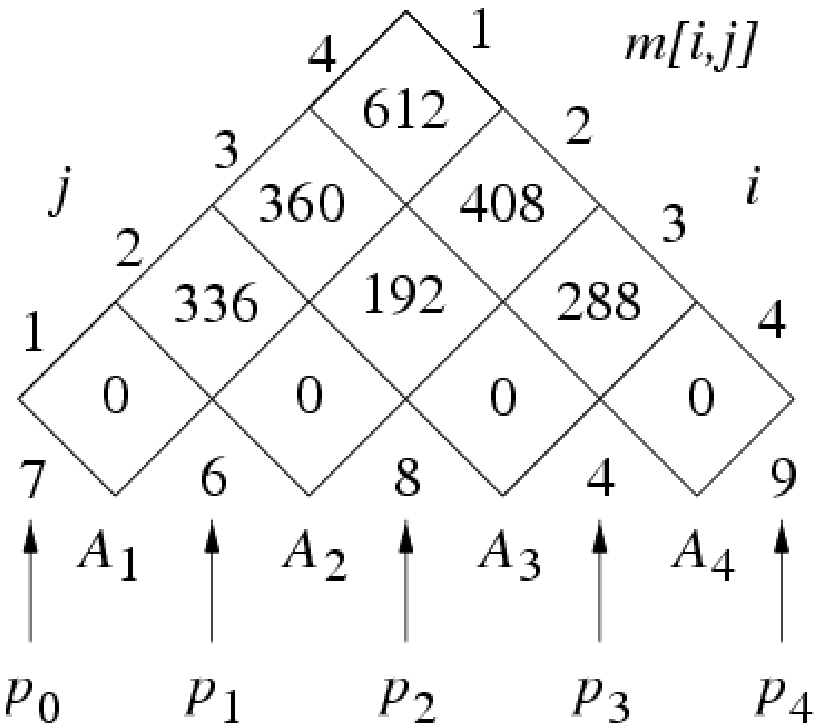
$$(A_i \times \ldots \times A_k) \times (A_{k+1} \times \ldots \times A_j).$$

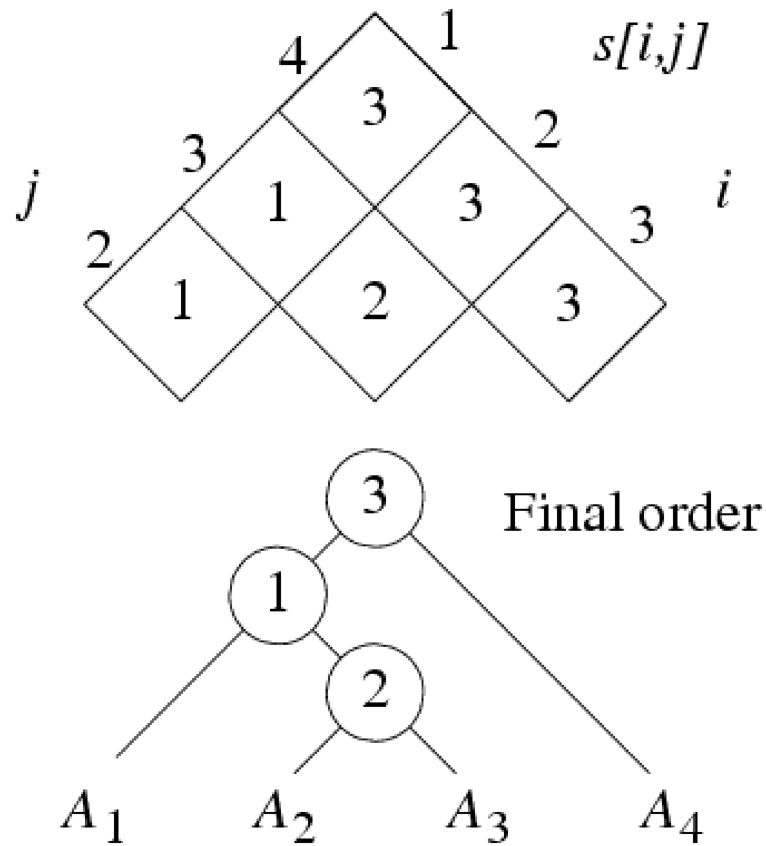
- Our solution it requires only $\Theta(n^2)$ space to store the optimal cost $m(i, j)$ for each of the subproblems: half of a 2d array $m[1..n, 1..n]$

Dynamic Programming Solution

```
MATRIXCHAINORDER(p)
n ← length[p] − 1
for i ← 1 to n
    do m[i, i] ← 0
for l ← 2 to n                      ▷ l is the chain length
    do for i ← 1 to n − l + 1
        do j ← i + l − 1
            m[i, j] ← ∞
            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
```

Matrix Multiplication Example





- http://www.cs.auckland.ac.nz/software/AlgAnim/mat_chain.html

Constructing the Optimal Solution

- Thus $s[1, n]$ is the index for the last multiplication, the earlier ones can be determined recursively.
- The initial call is MatrixChainMultiply(A, s, 1, n):

```
MATRIXCHAINMULTIPLY(A, s, i, j)
if i < j
    then X ← MATRIXCHAINMULTIPLY(A, i, s[i, j])
         Y ← MATRIXCHAINMULTIPLY(A, s[i, j] + 1, j)
         return MATRIXMULTIPLY(X, Y)
    else return A[i]
```

Elements of Dynamic Programming

- Optimal substructure:
  - The optimal solution for a problem consists of optimal solutions of subproblems.
  - For matrix chain multiplication, the optimal parenthesization of $A_1 \times \ldots \times A_n$ contains optimal parenthesization for the subproblems $A_1 \times \ldots \times A_k$ and $A_{k+1} \times \ldots \times A_n$.
- Overlapping subproblems:
  - Solving a subproblem leads to same subsubproblems over and over again.
  - For matrix chain multiplication, the subsequences occur in larger sequences in various forms, hence solutions for those can be re-used several times.
- This contrasts dynamic programming from divide-and-conquer, which solves all subproblems independently.

Memoization (1)

- Dynamic programming leads typically to a bottom-up construction of the solution, rather than to a top-down as divide-and-conquer.
- <u>*Memoization*</u> is a technique for *top-down* dynamic programming.
- Consider first a straightforward recursive top down solution:

```
RECURSIVEMATRIXCHAIN(p, i, j)

if i = j
    then return 0

m[i, j] ← ∞
for k ← i to j − 1
        do q ← RECURSIVEMATRIXCHAIN(p, i, k)+
           RECURSIVEMATRIXCHAIN(p, k + 1, j) + p_{i−1} p_k p_j
           if q < m[i, j]
              then m[i, j] ← q
return m[i, j]
```

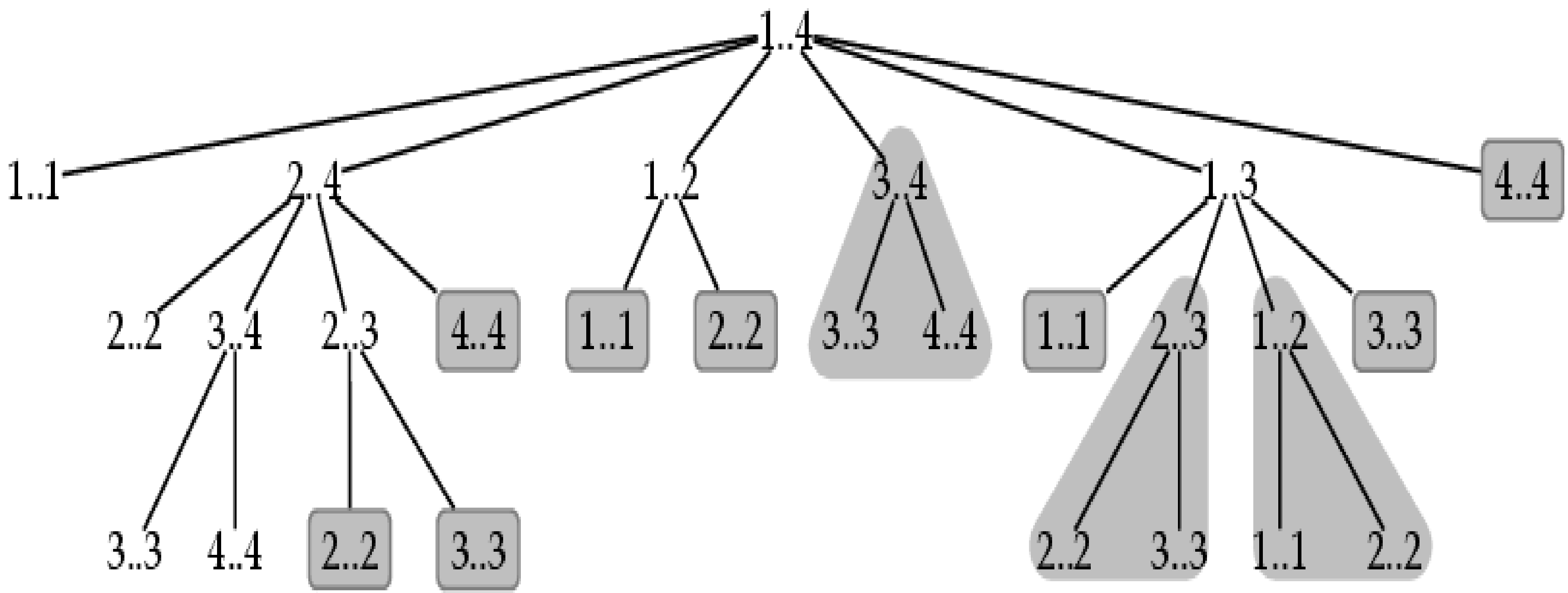
Memoization (2)

- The idea is, once subproblem has been solved, to *memo(r)ize* it in a table for future use. Initially the table contains special values indicating that the entry has not yet been computed.

```
MEMOIZEDMATRIXCHAIN(p)
  n ← length[p] − 1
  for i ← 1 to n
      do for j ← i to n
             do m[i, j] ← ∞
  return LOOKUPCHAIN(p, 1, n)
```

```
LOOKUPCHAIN(p, i, j)
  if m[i, j] ≠ ∞
     then return m[i, j]
  if i = j
     then m[i, j] ← 0                      ▷ Basis case
     else  for k ← i to j − 1              ▷ try all splits
               do q ← LOOKUPCHAIN(p, i, k)+
                      LOOKUPCHAIN(p, k + 1, j) + p_{i−1} p_k p_j
                  if q < m[i, j]
                     then m[i, j] ← q      ▷ update if better
  return m[i, j]                           ▷ final cost
```

Recursion Tree and the Effect of Memoization



Memoization (3)

- The asymptotic running time of MemoizedMatrix Chain is identical to that of MatrixChainOrder, both $O(n^3)$.
- The structure of a memoized algorithm is close to the recursive structure of a divide-and -conquer algorithm.
- The advantage of memoization is that possibly subproblems which are not needed are not solved.
- The disadvantage is that some more overhead due to the recursive calls and due to the checking of table entries is needed. This amounts to a constant factor.

Optimal Binary Search Trees

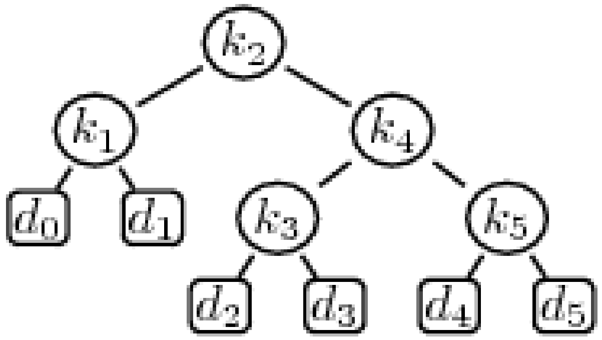
- Example problems:
  - Design a program to translate text from English to French
  - Design a part of a compiler which looks for language keywords
- Fact:
  - Words appear with different frequencies, thus some are more often looked for than others
- Problem: how to organize a binary search tree so as to minimize the number of nodes visited in all searches, given that we know how often each word occurs

Optimal Binary Search Trees. Example

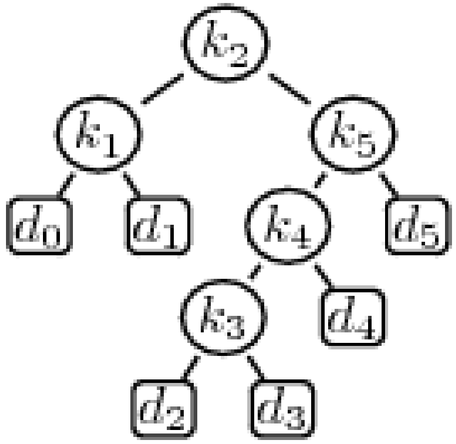
- Two BSTs each of 5 keys with probabilities:
  - p_i = probability of search for k_i
  - q_i = probability of search for value not in K

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

- Minimize: $$\sum_{i=1}^{n} p_i \times (depth_T(k_i)+1) + \sum_{i=0}^{n} q_i \times depth_T(d_i)$$



d_i =dummy keys



Expected search cost 2.80

Expected search cost 2.75 (optimal)

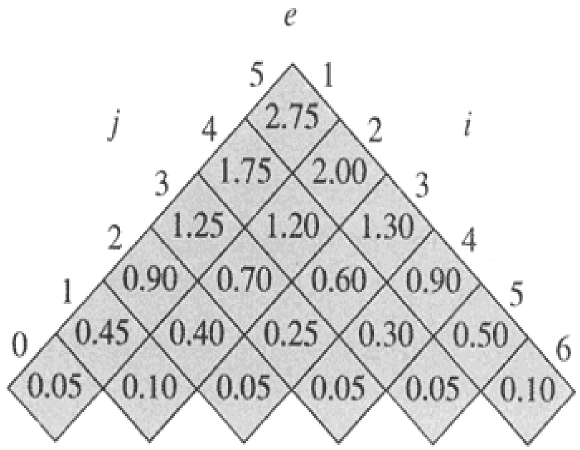
Optimal Binary Search Trees

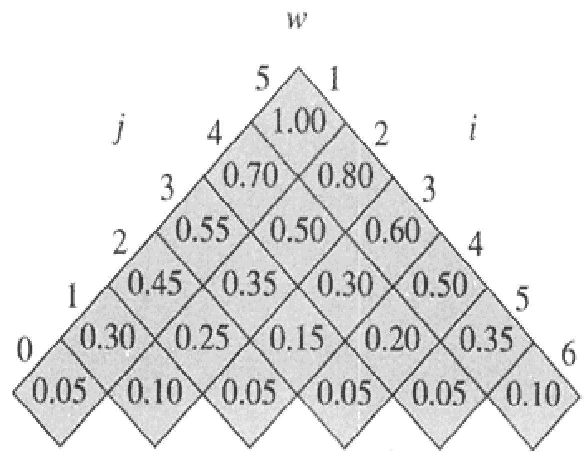
```
OPTIMAL-BST(p, q, n)
for i ← 1 to n + 1
    do e[i, i − 1] ← q_{i−1}
       w[i, i − 1] ← q_{i−1}
for l ← 1 to n
    do for i ← 1 to n − l + 1
           do j ← i + l − 1
              e[i, j] ← ∞
              w[i, j] ← w[i, j − 1] + p_j + q_j
              for r ← i to j
                  do t ← e[i, r − 1] + e[r + 1, j] + w[i, j]
                     if t < e[i, j]
                        then e[i, j] ← t
                             root[i, j] ← r
return e and root
```

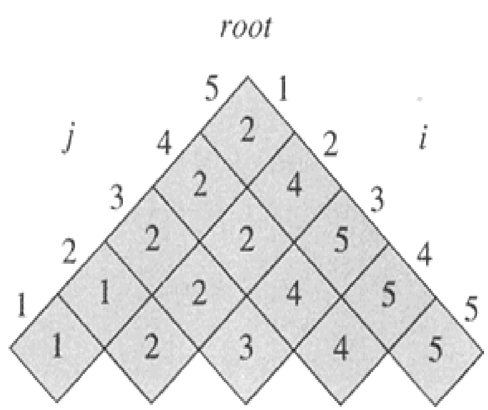
$e[i, j]$ as the expected cost of searching an optimal binary search tree containing the keys k_i, ..., k_j. Ultimately, we wish to compute $e[1, n]$.

- http://www.cse.yorku.ca/~aaw/Gubarenko/BSTAnimation.html

Optimal Binary Search Trees







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

Dynamic Programming

- In general, to apply dynamic programm-ing, we have to address a number of issues:

1. Show **optimal substructure** – an optimal solution to the problem contains within it optimal solutions to sub-problems
    - Solution to a problem:
        - Making a choice out of a number of possibilities (look what possible choices there can be)
        - Solving one or more sub-problems that are the result of a choice (characterize the space of sub-problems)
    - Show that solutions to sub-problems must themselves be optimal for the whole solution to be optimal (use “cut-and-paste” argument)

Dynamic Programming (2)

1. Write a recurrence for the value of an optimal solution
   - $M_{opt} = Min_{\text{over all choices } k}$ {(Sum of M_{opt} of all sub-problems, resulting from choice *k*) + (the cost associated with making the choice *k*)}
   - Show that the number of different instances of sub-problems is bounded by a polynomial
2. Compute the value of an optimal solution in a bottom-up fashion, so that you always have the necessary sub-results pre-computed (or use memoization)
   - See if it is possible to reduce the space requirements, by "forgetting" solutions to sub-problems that will not be used any more
3. Construct an optimal solution from computed information (which records a sequence of choices made that lead to an optimal solution)

Another animated example: http://optlab-server.sce.carleton.ca/POAnimations2007/Dynamic.html

Reading

- AHU, chapter 10, sections 1 and 2
- Preiss, chapter: Algorithmic Patterns and Problem Solvers, sections Top-Down Algorithms: Divide-and-Conquer and Bottom-Up Algorithms: Dynamic Programming
- CLR, chapter 16, CLRS chapter 2, section 2.3, chapter 15
- More visualizations (for other algs as well): http://alvie.algoritmica.org/alvie3/visualizations