Algorithms and Lab (CSE130)

Dynamic Programming

Muhammad Tariq Mahmood

tariq@koreatech.ac.kr
School of Computer Science and Engineering

Note: These notes are prepared from the following resources.

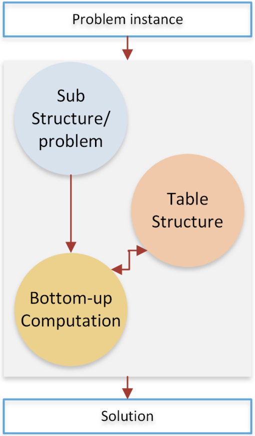
- (main text)Foundations of Algorithms, by Richard Neapolitan and Kumarss Naimipour
- Python Algorithm (파이썬 알고리즘) by Y.K. Choi (2021) (Korean)
- Introduction to the Design and Analysis of Algorithms by Anany Levitin
- Introduction to Algorithms, by By Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein
- https://www.geeksforgeeks.org

Contents

Dynamic Programming Approach

- Dynamic programming is a bottom-up approach for solving problems with overlapping subproblems.
- There are basically three elements that characterize a dynamic programming algorithm:

  1. Substructure: Decompose the given problem into smaller subproblems. Express the solution of the original problem in terms of the solution for smaller problems. (Establish a recursive property)
  2. Table Structure: After solving the sub-problems, store the results to the sub problems in a table.
  3. Bottom-up Computation: Using table, combine the solution of smaller subproblems to solve larger subproblems and eventually arrives at a solution to complete problem.



- The word "programming" in the name of this technique stands for "planning" and does not refer to computer programming.

Example: Computing Binomial Coefficients

- Binomial Theorem $(a+b)^n = \sum_{k=0}^{n} \frac{n!}{k!(n-k)} a^k b^{n-k}$
- Binomial coefficients: $\binom{n}{k} = \frac{n!}{k!(n-k)}$
- Another Representation:

$$\binom{n}{k} = \begin{cases} \binom{n-1}{k-1} + \binom{n-1}{k}, & 0 < k < n \\ 1 & k = 0 \text{ or } k = n \end{cases}$$

- Establish a recursive property.

$$B[i][j] = \begin{cases} B[i-1][j-1] + B[i-1][j], & 0 < j < i \\ 1, & j = 0 \text{ or } j = i \end{cases}$$

$$\begin{aligned}
\binom{n}{k} &= \frac{n!}{k! * (n-k)!} \\
&= \frac{(n-1)! * n}{k! * (n-k)!} \\
&= \frac{(n-1)! * n}{((k-1)! * k * (n-k-2)! * (n-k-1) * (n-k))} \\
&= \frac{(n-1)!}{((k-1)! * (n-k-1)!)} * \frac{n}{k * (n-k)} \\
&= \left[\frac{(n-1)!}{((k-1)! * (n-k-1)!)}\right] * \left[\frac{1}{(n-k)} + \frac{1}{k}\right] \\
&= \frac{(n-1)!}{((k-1)! * (n-k)!)} + \frac{(n-1)!}{(k! * (n-k-1)!)} \\
&= \binom{n-1}{k-1} + \binom{n-1}{k}
\end{aligned}$$

Binomial Coefficient Using Decrease/Divide-and-Conquer

- **Pseudo-code**
  - **Problem:** Compute the binomial coefficient.
  - **Inputs:** nonnegative integers n and k, where $0 \leq k \leq n$.
  - **Outputs:** bin, the binomial coefficient $\binom{n}{k}$

```
procedure BIN1(integer n, integer k)                ▷ compute binomial coefficient
    if (k = 0 || n = k) then
        return 1                                      ▷ Base Cases
    else                                              ▷
        return bin(n − 1, k − 1) + bin(n − 1, k) Recursive Cases
    end if
end procedure
```

- **Complexity Analysis**
  - The algorithm is easy to design, but not efficient.
  - reason-1 :The divide-and-conquer approach is always inefficient when an instance is divided into two smaller instances that are almost as large as the original instance.
  - reason-2: The same instances are solved in each recursive call.
  - To determine $\binom{n}{k}$, $2\binom{n}{k} - 1$ terms are computed.

Proof through mathematical induction

- **induction base:** Show that for $n = 1$, $2\begin{pmatrix} n \\ k \end{pmatrix} - 1$ is true

$$2\begin{pmatrix} n \\ k \end{pmatrix} - 1 = 2\begin{pmatrix} 1 \\ 1 \end{pmatrix} - 1 = 2 - 1 = 1$$

- **induction hypothesis :** Assume that the number of terms needed to compute $\begin{pmatrix} n \\ k \end{pmatrix}$ are $2\begin{pmatrix} n \\ k \end{pmatrix} - 1$
- **induction step:** Prove that the number of terms needed to compute $\begin{pmatrix} n+1 \\ k \end{pmatrix}$ are $2\begin{pmatrix} n+1 \\ k \end{pmatrix} - 1$
- By the property of binomial coefficient

$$\begin{pmatrix} n+1 \\ k \end{pmatrix} = \begin{pmatrix} n \\ k-1 \end{pmatrix} + \begin{pmatrix} n \\ k \end{pmatrix} + 1$$

So, by putting

$\begin{pmatrix} n \\ k-1 \end{pmatrix} = 2\begin{pmatrix} n \\ k-1 \end{pmatrix} - 1$, $\begin{pmatrix} n \\ k \end{pmatrix} = 2\begin{pmatrix} n \\ k \end{pmatrix} - 1$ in above equation.

$$\begin{aligned}
\begin{pmatrix} n+1 \\ k \end{pmatrix} &= 2\begin{pmatrix} n \\ k-1 \end{pmatrix} - 1 + 2\begin{pmatrix} n \\ k \end{pmatrix} - 1 + 1 \\
&= 2\left[\frac{n!}{(k-1)!\,(n-k-1)!} + \frac{n!}{(k)!\,(n-k)!}\right] - 1 \\
&= 2\left[\frac{n!\,(k+n-k+1)}{(k)!\,(n+1-k)!}\right] - 1 \\
&= 2\left[\frac{n!\,(n+1)}{(k)!\,(n+1-k)!}\right] - 1 \\
&= 2\left[\frac{(n+1)!}{(k)!\,(n+1-k)!}\right] - 1 \\
&= 2\begin{pmatrix} n+1 \\ k \end{pmatrix} - 1
\end{aligned}$$

Binomial Coefficient Using Dynamic Programming

- Algorithm

```
procedure BC( n, k)
    integer i, j
    integer B[0..n][0..k]
    for (i = 0; i <= n; i++) do
        for (j = 0; j <= min(i, k); j++) do
            if (j == 0 || j == i) then
                B[i][j] = 1
            else
                B[i[j] = B[i − 1][j − 1] + B[i − 1][j]
            end if
        end for
    end for
end procedure
```

	0	1	2	3	4	j	k
0	1						
1	1	1					
2	1	2	1				
3	1	3	3	1			
4	1	4	6	4	1		
i							
n							

$$B[i][j] = \begin{cases} B[i-1][j-1] + B[i-1][j], & 0 < j < i \\ 1 & , \quad j = 0 \ or \ j = i \end{cases}$$

- Time complexity function : $T(n,k) = T_1(n,k) + T_2(n,k) \in \Theta(nk)$

$T_1(n,k) = \sum_{i=1}^{k} \sum_{j=1}^{i} 1 = \sum_{i=1}^{k} i = \frac{k(k+1)}{2}, (i \leq k)$ $\quad T_2(n,k) = \sum_{i=k+1}^{n+1} \sum_{j=1}^{k+1} 1 = (n-k+1)(k+1), (i > k)$

Dynamic Programming Approach (cont...)

- Solve the problem in bottom up fashion. It means that first compute the lowest/base value
- To compute $B[4][2] = \binom{4}{2}$ row wise computing entries of matrix B.
  - Row 0:

$$B[0][0] = 1$$

  - Row 1:

$$B[1][0] = 1$$
$$B[1][0] = 1$$

  - Row 2:

$$B[2][0] = 1$$
$$B[2][1] = B[1][0] + B[1][1] = 1 + 1 = 2$$
$$B[2][2] = 1$$

- Row 3:

$$B[3][0] = 1$$
$$B[3][1] = B[2][0] + B[2][1] = 1 + 2 = 3$$
$$B[3][2] = B[2][1] + B[2][2] = 2 + 1 = 3$$
$$B[3][3] = 1$$

- Row 4:

$$B[4][0] = 1$$
$$B[4][1] = B[3][0] + B[3][1] = 1 + 3 = 4$$
$$B[4][2] = B[3][1] + B[3][2] = 3 + 3 = 6$$
$$B[4][3] = B[3][2] + B[3][3] = 3 + 1 = 4$$
$$B[4][4] = 1$$

Path Counting Problem

Path Counting Problem:

- A chess rook can move horizontally or vertically to any square in the same row or in the same column of a chessboard.
- Find the number of shortest paths by which a rook can move from one corner of a chessboard to the diagonally opposite corner.
- The length of a path is measured by the number of squares it passes through, including the first and the last squares.

- Observations
- Let $T(i, j)$ be the number of the rook's shortest paths from square $(1, 1)$ to square (i, j) in the ith row and the jth column, where $1 \leq i, j \leq 8$
- base case: $T(i, 1)= P(1, j)$ =1 for any $1 \leq i, j \leq 8$.

Path Counting Problem (cont...)

- recursive case : Any shortest path $T(i,j)$ to square (i,j) is reached either from its left neighbor$(i-1,j)$ or from its upper neighbors $(i,j-1)$.

- Recursive Property

$$T[n][m] = \begin{cases} T[i][0] = 1, & j = 0 \\ T[0][j] = 1, & i = 0 \\ T[i][j] = T[i-1][j] + T[i][j-1] & 1 < i \leq n, 1 < j \leq m \end{cases}$$

- Using this recurrence, we can compute the values of $T(i,j)$ for each square (i,j) of the board.

- This can be done either row by row, or column by column, or diagonal by diagonal.

Table 1: Number of Paths

1	1	1	1	1	1	1	1
1	2	3	4	5	6	7	8
1	3	6	10	15	21	28	36
1	4	10	20	35	56	84	120
1	5	15	35	70	126	210	330
1	6	21	56	126	252	462	792
1	7	28	84	210	462	924	1716
1	8	36	120	330	792	1716	3432

Path Counting Problem (cont...)

- Divide/decrease and conquer based solution

```
procedure countPathDC(n,m)
    if (n == 1 || m == 1) then
        return 1
    else
        return (countPathDC(n-1,m) + countPathDC(n,m-1))
    end if
end procedure
```

Complexity

$$T(n,m) = \begin{cases} 1 & n = 0, m = 0 \\ T(n-1,m) + T(n,m-1) & n > 0, m > 0 \end{cases}$$

$$\in O(2^{max\{m,n\}})$$

- Dynamic programming based algorithm

```
procedure countPathDP(n,m)
    T[n][m]
    for (int i = 0; i < n; i++) do
        T[i][0] = 1
    end for
    for (int j = 0; j < m; i++) do
        T[0][j] = 1
    end for
    for (int i = 1; i < n; i++) do
        for (int j = 1; j < m; i++) do
            T[i][j] = T[i-1][j] + T[i][j-1]
        end for
    end for
end procedure
```

Complexity

$$T(n,m) = n + m + nm \in \Theta(nm)$$

Path Counting Problem (cont...)

Permutations and Combinations

- Combinatorics:, Permutations $\rightarrow$ all possible ways of doing something, (lists).
  - Number of permutations of an n-element set: $P(n) = n!$
  - having n-elements and want to find the number of ways k items can be ordered: $P(n,k) = \frac{n!}{(n-k)!}$
- Combinations (groups)
  - Number of k-combinations of an n-element set: $\binom{n}{k} = \frac{P(n,k)}{k!} = \frac{n!}{k!(n-k)!}$
  - Number of subsets of an n-element set: 2^n
- Combinatorics formulae can be used to calculate the number of unique paths to reach destination cells starting from the cell(1,1). If there is lattice of size $n \times m$ then paths from (1,1) to (n,m) are given as

$$paths = \frac{n!}{m!(n-m)!}$$

```
procedure countPathCmn(n,m) paths=1
    for (i=n; i< m+n-1; i++) do
        paths = paths × i
        paths = paths / i
    end for
    return paths
end procedure
```

- Complexity

$$T(m,n) = \sum_{i=n}^{m+n-1} 1 = \sum_{i=1}^{m} 1 \in O(m)$$

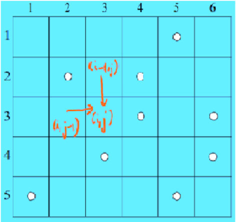
$$\binom{14}{7} = \frac{14!}{7!(14-7)!} = 3432$$

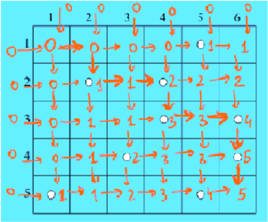
- For example, the shortest path composed of the vertical move from $(1,1)$ to $(8,1)$ followed by the horizontal move from $(8,1)$ to $(8,8)$ corresponds to the following sequence of 14 one-square moves: $d,d,d,d,d,d,d,r,r,r,r,r,r,r$

Coin-collecting problem

- Several coins are placed in cells of an $n \times m$ board, no more than one coin per cell
- A robot, located in the upper left cell of the board, needs to collect as many of the coins as possible and bring them to the bottom right cell.
- On each step, the robot can move either one cell to the right or one cell down from its current location.
- Solution
- Let $F(i,j)$ be the largest number of coins the robot can collect and bring to the cell (i, j) in the ith row and jth column of the board.
- When the robot visits a cell with a coin, it always picks up that coin.
- It can reach this cell either from the adjacent cell (i-1, j) above it or from the adjacent cell (i, j-1) to the left of it.
- The largest number of coins the robot can bring to cell (i, j) is the maximum of the two numbers F(i-1, j) and F(i, j-1), plus the one possible coin at cell (i, j) itself c_{ij}.
- The recursive property for computing $F(i,j)$:

$$\begin{cases} F(0,j) = 0, \ \textit{for}\ 1 \le j \le m \\ F(i,0) = 0, \ \textit{for}\ 1 \le i \le n \\ F(i,j) = \max\{F(i-1,j) + c_{ij}, F(i,j-1) + c_{ij}\} \\ \quad \textit{for}\ 1 \le i \le n \quad \textit{and}\ 1 \le j \le m \end{cases}$$





Algorithm

- **Problem:** Apply dynamic programming to compute the largest number of coins a robot can collect on an $n \times m$ board by starting at (1, 1) and moving right and down from upper left to down right corner
- **Input:** Matrix $C[n, m]$ whose elements are equal to 1 and 0 for cells with and without a coin, respectively
- **Output:** Largest number of coins the robot can bring to cell (n, m)

Algorithm (Complexity Analysis)

$$
\begin{aligned}
T(n,m) &= \sum_{j=2}^{m} 1 + \sum_{i=2}^{n} \sum_{j=2}^{m} 1 \\
&= m - 1 + \sum_{i=2}^{n} (m-1) \\
&= m - 1 + (m-1)(n-1) \\
&= m - 1 + mn - m - n + 1 \\
&= mn - n + 2
\end{aligned}
$$

- Pseudo-code

```
procedure ROBOTCOINCOLLECTION(C[1...n, 1...m])
    F[1,1] = C[1,1]
    for (j = 2; j <= m; j++) do
        F[1,j] = F[1,j-1] + C[1,j]
    end for
    for (i = 2; <= n; i++) do
        F[i,1] = F[i,1] + C[i,1]
        for (j = 2; j <= m; j++) do
            F[i,j] =
max{F[i-1,j] + C[i,j], F[i,j-1] + C[i,j]}
        end for
    end for
end procedure
```

$$T(n,m) \in \Theta(nm)$$

- Optimal Path

- It is possible to trace the computations backwards to get an optimal path.

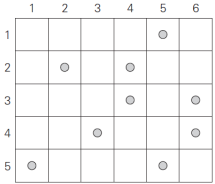
- If F(i-1, j) > F(i, j-1), an optimal path to cell (i, j) must come down from the adjacent cell above it;

- If F(i-1, j) < F(i, j-1), an optimal path to cell (i, j) must come from the adjacent cell on the left;

- If F(i-1, j) = F(i, j-1), it can reach cell (i, j) from either direction.

-

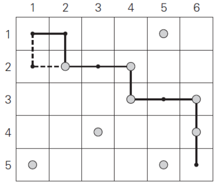
- Figures: (a) Coins to collect. (b) Dynamic programming algorithm results. (c) Two paths to collect 5 coins, the maximum number of coins possible.



(a)

	1	2	3	4	5	6
1	0	0	0	0	1	1
2	0	1	1	2	2	2
3	0	1	1	3	3	4
4	0	1	2	3	3	5
5	1	1	2	3	4	**5**

(b)



(c)

Problem definition

- Suppose we want to multiply a 2×3 matrix with a 3×4 matrix

$$\underbrace{\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}}_{2\times 3} \times \underbrace{\begin{bmatrix} 7 & 8 & 9 & 1 \\ 2 & 3 & 4 & 5 \\ 6 & 7 & 8 & 9 \end{bmatrix}}_{3\times 4} = \underbrace{\begin{bmatrix} 29 & 35 & 41 & 38 \\ 74 & 89 & 104 & 83 \end{bmatrix}}_{2\times 4}$$

- Total entries in the resultant matrix are $2 \times 4 = 8$.

- The number of multiplication operation in one entry are $= \underbrace{1 \times 7 + 2 \times 2 + 3 \times 6}_{3 \ \textit{multiplications}} = 29$

- The number of multiplication in $2 \times 4 = 8$ entries are $= 2 \times 4 \times 3 = 24$.

- In general, to multiply $A_{i\times j}$ matrix with $B_{j\times k}$ matrix using the standard method, the required number of multiplications are .

$$i \times j \times k$$

Chained Matrix Multiplication (cont...)

- Example: Consider the multiplication of the following four matrices:

$$A_{20\times 2} \times \; B_{2\times 30} \times \; C_{30\times 12} \; \times D_{12\times 8}$$

- For different order of matrices multiplications, the number of elementary multiplications are changed.

$$\begin{aligned}
A\,(B\,(CD)) &= 30\times 12\times 8 + 2\times 30\times 8 + 20\times 2\times 8 && = 3,680\\
(AB)\,(CD) &= 20\times 2\times 30 + 30\times 12\times 8 + 20\times 30\times 8 && = 8,880\\
A\,((BC)\,D) &= 2\times 30\times 12 + 2\times 12\times 8 + 20\times 2\times 8 && = \mathbf{1{,}232}\\
((AB)\,C)\,D &= 20\times 2\times 30 + 20\times 30\times 12 + 20\times 12\times 8 && = 10,320\\
(A\,(BC))\,D &= 2\times 30\times 12 + 20\times 2\times 12 + 20\times 12\times 8 && = 3,120
\end{aligned}$$

- Our goal is to develop an algorithm that determines the optimal order for multiplying n matrices.
- The optimal order depends only on the **dimensions** of the matrices.
- Therefore, besides n (number of matrices), these **dimensions** would be the only input to the algorithm

Chained Matrix Multiplication (cont...)

Recursive Solution

- Algorithm

```
procedure MCMRec( dims[], i, j)
    cost = 0, minmul = inf
    if j <= i + 1 then
        return 0
    end if
    for k = i + 1; k < j; k + + do
        cost = cost + MCMRec(dims, i, k)
        cost = cost + MCMRec(dims, k, j)
        cost = cost + dims[i] * dims[k] * dims[j]
        if cost < minmul then
            minmul = cost
        end if
    end for
    return minmul
end procedure
```

- The brute-force algorithm is to consider all possible orders and take the minimum
- If we have just 1 item, then there is only one way to parenthesize.
- If we have n items, then there are $n-1$ places where you could break the list with the outermost pair of parentheses
- Complexity
  - The number of different ways of parenthesizing n items is

$$\begin{cases} P(n) = 1, & n = 1 \\ P(n) = \sum_{k=1}^{n-1} P(k)P(n-k), & n > 1 \end{cases}$$

  - Solution

$$P(n) \in \Omega\left(4^n / n^{2/3}\right)$$

  - This is related to a famous function in combinatorics called the **Catalan numbers**.
  - **Catalan numbers** are related to the number of different binary trees on n nodes.
  - **Catalan numbers** are given by the formula:

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

Dynamic Programming Approach

- let n matrices: $\{A_1, A_2, \cdots, A_k, \cdots, A_n\}$ are given for multiplication
- principle of optimality applies in this problem. That is, the optimal order for multiplying n matrices includes the optimal order for multiplying any subset of the n matrices.
- For example, if the optimal order for multiplying six particular matrices is

$$A_1\left(\left(\left(\left(A_2 A_3\right) A_4\right) A_5\right) A_6\right)$$

  Then any subset $(A_2A_3)A_4$ *or* $((A_2A_3)A_4)A_5$ must be the optimal order for multiplying matrices
- If A_{k-1} and A_k matrices are multiplied then the number of columns in A_{k-1} must equal the number of rows in A_k.
- If let d_{k-1} be the number of columns in A_{k-1} and d_k be the number of rows in A_k for $1 \le k \le n$, the dimension of A_k is $d_{k-1} \times d_k$, as shown in the Figure 1.

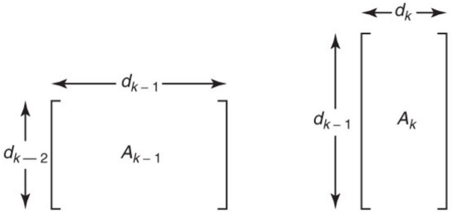


Figure 1: The number of columns in A_{k-1} is the same as the number of rows in A_k

- Based on this observation, the following recursive property can be established when multiplying n matrices. for $1 < i < j < n$

$$\begin{cases} M[i][j] = \underset{i \le k \le j-1}{\text{minimum}} \left(M[i][k] + M[k+1][j] + d_{i-1} d_k d_j \right) \ , \quad if \ i < j \\ M[i][i] = 0, \quad otherwise \end{cases}$$

Algorithm: Minimum Multiplications

- **Problem:** Determining the minimum number of elementary multiplications needed to multiply n matrices and an order that produces that minimum number.

- **Inputs:** the number of matrices n, and an array of integers d, indexed from 0 to n, where $d[i-1] \times d[i]$ is the dimension of the i^{th} matrix.

- **Output:** minmult, the minimum number of elementary multiplications needed to multiply the n matrices; a two-dimensional array P from which the optimal order can be obtained. P has its rows indexed from 1 to $n-1$ and its columns indexed from 1 to n. $P[i][j]$ is the point where matrices i through j are split in an optimal order for multiplying the matrices.

Pseudo-code

```
procedure MINMULT(integer n, integer d[], integer P[][])
    integer i, j, k, diagonal                    ▷ variables i, j, k, diagonal of type integer
    integer M[1..n][1..n]                        ▷ an array M[1..n][1..n] of type integer
    for (i = 1; i <= n; i++) do
        M[i][i] = 0                              ▷ Base-case
    end for
    for (diagonal = 1; diagonal <= n − 1; diagonal++) do   ▷ diagonal is just above the main diagonal
        for (i = 1; i <= n − diagonal; i++) do
            j = i + diagonal
            M[i][j] = minimum_{i≤k≤j−1} (M[i][k] + M[k+1][j] + d[i−1] ∗ d[k] ∗ d[j])
            P[i][j] = k                          ▷ a value of k that gave the minimum
        end for
    end for
    return M[1][n]
end procedure
```

$$M[i][j] = \underset{i \le k \le j-1}{minimum}\left(M[i][k] + M[k+1][j] + d[i-1] * d[k] * d[j]\right)$$

- Note that, matrices themselves are not inputs because the values in the matrices are irrelevant to the problem

- **Example:** Consider the Problem Instance:

$$\underset{\substack{5\times 2\\ d_0\times d_1}}{A_1} \times \underset{\substack{2\times 3\\ d_1\times d_2}}{A_2} \times \underset{\substack{3\times 4\\ d_2\times d_3}}{A_3} \times \underset{\substack{4\times 6\\ d_3\times d_4}}{A_4} \times \underset{\substack{6\times 7\\ d_4\times d_5}}{A_5} \times \underset{\substack{7\times 8\\ d_5\times d_6}}{A_6}$$

	1	2	3	4	5	6
1		1	1	1	1	1
2			2	3	4	5
3				3	4	5
4					4	5
5						5

Figure 2: The matrix P

Diagonal 1 Diagonal 2 Diagonal 3 Diagonal 4 Diagonal 5

	1	2	3	4	5	6	
1	0	30	64	132	226	348	← Final answer
2		0	24	72	156	268	
3			0	72	198	366	
4				0	168	392	
5					0	336	
6						0	

Figure 3: The matrix M

- The matrices M and P obtained by using the above algorithm are shown. Upper right corner provide the results. The matrix P produced by the algorithm can be used to print the optimal order

Chained Matrix Multiplication (cont...)

- The steps in the dynamic programming algorithm follow
  - Compute diagonal 0

$$M\left[i\right]\left[i\right] = 0 \;\; \textit{for} \;\; 1 \leq i \leq 6$$

  - Compute diagonal 1

$$\begin{aligned}
M\left[1\right]\left[2\right] &= \underbrace{\textit{minimum}}_{1 \leq k \leq 1}\left(M\left[1\right]\left[k\right] + M\left[k+1\right]\left[2\right] + d_{i-1}d_k d_j\right) \\
&= M\left[1\right]\left[1\right] + M\left[2\right]\left[2\right] + d_0 d_1 d_2 \\
&= 0 + 0 + (5 \times 2 \times 3) = 30
\end{aligned}$$

  - Compute first element of diagonal 2 M[1][2]

$$\begin{aligned}
M\left[1\right]\left[3\right] &= \underbrace{\textit{minimum}}_{1 \leq k \leq 2}\left(M\left[1\right]\left[k\right] + M\left[k+1\right]\left[2\right] + d_{i-1}d_k d_j\right) \\
&= \textit{minimum}\begin{bmatrix} M\left[1\right]\left[1\right] + M\left[2\right]\left[3\right] + d_0 d_1 d_3, \\ M\left[1\right]\left[2\right] + M\left[3\right]\left[3\right] + d_0 d_2 d_3 \end{bmatrix} \\
&= \textit{minimum}\begin{bmatrix} 0 + 24 + (5 \times 2 \times 4), \\ 30 + 0 + (5 \times 3 \times 4) \end{bmatrix} \\
&= 64
\end{aligned}$$

Chained Matrix Multiplication (cont...)

- Compute first element of diagonal 3 M[1][3]

$$
\begin{aligned}
M[1][4] &= \underbrace{minimum}_{1 \leq k \leq 3} \left(M[1][k] + M[k+1][2] + d_{i-1} d_k d_j \right) \\
&= minimum \begin{bmatrix} M[1][1] + M[2][4] + + d_0 d_1 d_4, \\ M[1][2] + M[3][4] + d_0 d_2 d_4, \\ M[1][3] + M[4][4] + d_0 d_3 d_4 \end{bmatrix} \\
&= minimum \begin{bmatrix} 0 + 72 + (5 \times 2 \times 6), \\ 30 + 72 + (5 \times 3 \times 6), \\ 64 + 0 + (5 \times 4 \times 6) \end{bmatrix} \\
&= 132
\end{aligned}
$$

- Compute first element of diagonal 4 $\rightarrow$ M[1][4]
- Compute first element of diagonal 5 $\rightarrow$ M[1][5]
- Compute first element of diagonal 6 $\rightarrow$ M[1][6]
- Similarly, compute other entries of the resultant matrix

$$
\begin{aligned}
&M[2][3], \quad M[2][4], \quad M[2][5], \quad M[2][6] \\
&M[3][4], \quad M[3][5], \quad M[3][6] \\
&M[4][5], \quad M[4][6] \\
&M[5][6]
\end{aligned}
$$

- Complexity Function

$$T(n) = \underbrace{\sum_{d}^{n-1} \underbrace{\sum_{i=1}^{n-d} \underbrace{\sum_{k=i}^{j-1}}_{k-loop} 1}_{i-loop}}_{diagonal-loop} \quad \text{where } j = i + d$$

$$= \sum_{d=1}^{n-1} \sum_{i=1}^{n-d} (j - 1 - i + 1)$$

$$= \sum_{d=1}^{n-1} \sum_{i=1}^{n-d} (i + d - 1 - i + 1)$$

$$= \sum_{d=1}^{n-1} \sum_{i=1}^{n-d} d$$

$$= \sum_{d=1}^{n-1} (n - d) \times d$$

$$= \sum_{d=1}^{n-1} \left(nd - d^2\right)$$

$$= \sum_{d=1}^{n-1} nd - \sum_{d=1}^{n-1} d^2$$

$$= n\frac{(n-1)(n-1+1)}{2} - \frac{(n-1)(n-1+1)(2n-2+1)}{6}$$

$$= \frac{n^3 - n^2}{2} - \frac{2n^3 - 3n^2 + n}{6}$$

$$= \frac{3n^3 - 3n^2 - 2n^3 + 3n^2 - n}{6}$$

$$= \frac{n^3 - n}{6}$$

$$= \frac{n(n-1)(n+1)}{6}$$

Hence

$$T(n) \in \Theta\left(n^3\right)$$

Algorithm: Print Optimal Order

- **Problem:** Print the optimal order for multiplying n matrices.
- **Inputs:** Positive integer n, and the array P produced by Algorithm 3.6. $P[i][j]$ is the point where matrices i through j are split in an optimal order for multiplying those matrices.
- **Outputs:** the optimal order for multiplying the matrices.

Pseudo-code

```
procedure ORDER(integer i, integer j)
    if (i == j) then
        print(A, i)
    else
        k = P[i][j]
        print(()
        order(i, k)
        order(k + 1, j)
        print())
    end if
end procedure
```

- Complexity in asymptotic notations

$$T(n) \in \Theta(n)$$

- Remarks
  - The presented $\Theta\left(n^3\right)$ algorithm for chained matrix multiplication is from Godbole (1973).
  - Yao (1982) developed methods for speeding up certain dynamic programming solutions. Using those methods, it is possible to create a $\Theta\left(n^2\right)$ algorithm for chained matrix multiplication.
  - Hu and Shing (1982, 1984) describe a $\Theta(n \lg n)$ algorithm for chained matrix multiplication.

Binary Search Tree

- A **binary search tree** is a **binary tree** of items (called keys), that come from an ordered set, such that
  - Each node contains one key.
  - The keys in the **left subtree** of a given node are less than or equal to the key in that node.
  - The keys in the **right subtree** of a given node are greater than or equal to the key in that node.
- The **depth/height** of a node in a tree is the number of edges in the unique path from the root to the node. It is also called the **level** of the node in the tree

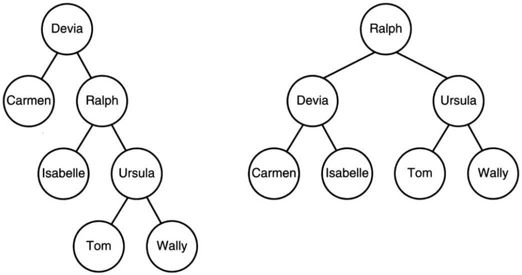


Figure 4: Two binary search trees

- A binary tree is called **balanced tree** if the depth of the two subtrees of every node never differ by more than 1
- The tree on the left in Figure 4 is not balanced, whereas the tree on the right in Figure 4 is balanced.
- An algorithm that searches for a key in a binary search tree is provided here

Optimal Binary Search Tree (cont...)

- **Algorithm: Searching Binary Tree**
  - **Problem:** Determine the node containing a key in a binary search tree. It is assumed that the key is in the tree.
  - **Inputs:** A pointer **tree** to a binary search tree and a key *keyin*.
  - **Outputs:** a pointer p to the node containing the key.

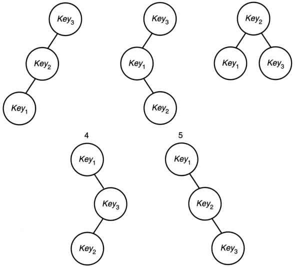
- Pseudo-code

```
procedure SEARCH(Tree, keyin)
    bool found = false                          ▷ Boolean variable found
    while (!found) do
        if (p−>key == keyin) then
            found = true
        else
            if (keyin < p−>key) then
                p = p−>left                     ▷ advance to the left child
            else
                p = p−>right                    ▷ advance to the right child
            end if
        end if
    end while
    return p
end procedure
```

- **Optimal Binary Search Tree**
- Our goal is to organize the keys in a binary search tree so that the average time it takes to locate a key is minimized.
- Let $k_1, k_2, ..., k_n$ be keys and their probabilities be $p_1, p_2, ..., p_n$
- Search time (number of comparisons) for ith key $time(k_i) = depth(k_i) + 1$

$$\begin{aligned} Average\ \ Time &= \sum_{i=1}^{n} time(k_i)p_i \\ &= \sum_{i=1}^{n} (depth(k_i) + 1)p_i \\ &= \sum_{i=1}^{n} (depth(k_i))p_i + \sum_{i=1}^{n} p_i \\ &= \sum_{i=1}^{n} (depth(k_i))p_i + 1 \end{aligned}$$

- **Example:** five different trees are shown when $n = 3$ and probability for each item $p_1 = 0.7,\ p_2 = 0.2,\ p_3 = 0.1$



- The average search times for the trees in Figure are:

$$\begin{aligned} 3\,(0.7)\ &+ 2\,(0.2)\ + 1\,(0.1)\ = 2.6 \\ 2\,(0.7)\ &+ 3\,(0.2)\ + 1\,(0.1)\ = 2.1 \\ 2\,(0.7)\ &+ 1\,(0.2)\ + 2\,(0.1)\ = 1.8 \\ 1\,(0.7)\ &+ 3\,(0.2)\ + 2\,(0.1)\ = 1.5 \\ 1\,(0.7)\ &+ 2\,(0.2)\ + 3\,(0.1)\ = 1.4 \end{aligned}$$

Optimal Binary Search Tree (cont...)

- Let $key_1, key_2, key_3, \cdots, key_n$ be the n keys in order, and let p_i be the probability that key_i is the search key.
- The number of binary search trees with n keys are given by $\frac{1}{(n+1)}\begin{pmatrix} 2n \\ n \end{pmatrix}$
- We will call a tree optimal for those keys with minim average time(AST) for searching and denote the ASt values by $A[i][j]$.
- It takes one comparison to locate a key in a tree containing one key, $A[i][i] = p_i$.
- let tree 1 be an optimal tree given the restriction that key_1 is at the root, tree 2 be an optimal tree given the restriction that key_2 is at the root, $\cdots$, tree n be an optimal tree given the restriction that key_n is at the root.

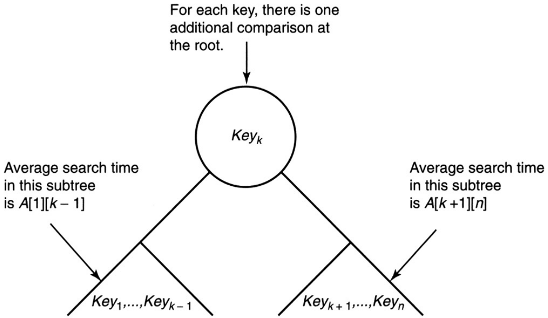


Figure 5: Optimal binary search tree given that key_k is at the root.

- For $1 \le k \le n$, the subtrees of tree k must be optimal. The average search times in these subtrees are as depicted in Figure .

- **Average search time**

$$\underbrace{A[1][k-1]}_{\textit{Average time in left subtree}} + \underbrace{p_1 + \cdots + p_{k-1}}_{\textit{Additional time comparaing at root}} + \underbrace{p_k}_{\textit{Average time serching for root}} + \underbrace{A[k+1][n]}_{\textit{Average time in right subtree}} + \underbrace{p_{k+1} + \cdots + p_n}_{\textit{Additional time comparaing at root}}$$

$$\Rightarrow A[1][k-1] + A[k+1][n] + \sum_{m=1}^{n} p_m$$

- **The recursive property**

$$\begin{cases} \left.\begin{array}{ll} A[i][j] = 0, & i > j \\ A[i][i] = 0, & i = j \end{array}\right\} \rightarrow (\text{Base Cases}) \\ A[i][j] = \underset{i \le k \le j}{\text{minimum}} \left(A[i][k-1] + A[k+1][j]\right) + \sum_{m=i}^{j} p_m \rightarrow (\text{Recursive Cases}) \end{cases}$$

- **Algorithm Optimal Binary Search Tree** Dynamic programming will be used to develop a more efficient algorithm.
  - **Problem:** Determine an optimal binary search tree for a set of keys, each with a given probability of being the search key.
  - **Inputs:** n, the number of keys, and an array of real numbers p indexed from 1 to n, where p [i] is the probability of searching for the ith key.
  - **Outputs:** A variable minavg, whose value is the average search time for an optimal binary search tree; and a two-dimensional array R from which an optimal tree can be constructed. R has its rows indexed from 1 to n + 1 and its columns indexed from 0 to n. R [i] [j] is the index of the key in the root of an optimal tree containing the ith through the jth keys.

Optimal Binary Search Tree (cont...)

- **Pseudo-code**

procedure OPTSEARCHTREE($P[]$)
    for $(i = 1; i <= n; i++)$ **do**
        $A[i][i-1] = 0,\ A[i][i] = 0$
        $R[i][i-1] = 0,\ R[i][i] = 0$
    end for
    $A[n+1][n] = 0,\ R[n+1][n] = 0$
    for $(diagonal = 1; diagonal <= n-1; diagonal++)$ **do**
        for $(i = 1; i <= n - diagonal; i++)$ **do**
            $j = i + diagonal$
            $A[i][j] = \underset{i \le k \le j}{\text{minimum}} \left(A[i][k-1] + A[k+1][j]\right) + \sum_{m=i}^{j} p_m$
            $R[i][j] = k$
        end for
    end for
    return *minavg* $A[1][n]$
end procedure

- **Complexity** $T(n) = \frac{n(n-1)(n+4)}{6} \in \Theta\left(n^3\right)$

Optimal Binary Search Tree (cont...)

- **Algorithm: Build Optimal Binary Search Tree**
  - **Problem:** Build an optimal binary search tree.
  - **Inputs:** n, the number of keys, an array Key containing the n keys in order, and the array R produced by Algorithm 3.9. $R[i][j]$ is the index of the key in the root of an optimal tree containing the i_{th} through the j_{th} keys.
  - **Outputs:** a pointer tree to an optimal binary search tree containing the n keys.
- Complexity $T(n) \in \Theta(n)$

- **Pseudo-code**

```
procedure TREE( R[][], i, j)
    Integer k = R[i][j]
    node-pointer p
    if (k == 0) then
        return null
    else
        p = new  nodetype
        p-> key = Key[k]
        p-> left = tree(R[][], i, k - 1)
        p-> right = tree(R[][], k + 1, j)
        return p
    end if
end procedure
```

Optimal Binary Search Tree (cont...)

- Example: Supposed we have the following values of the array Key:

Don	Isabelle	Ralph	Wally
Key[1]	Key[2]	Key[3]	Key[4]

$$p = \begin{bmatrix} 0.375 & 0.375 & 0.125 & 0.125 \end{bmatrix}$$

- The matrices A and R produced by Algorithm 3.9 are shown in Figure. The minimal average search time is $7/4$.

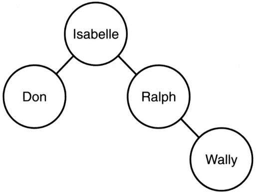
	0	1	2	3	4
1	0	$\frac{3}{8}$	$\frac{9}{8}$	$\frac{11}{8}$	$\frac{7}{4}$
2		0	$\frac{3}{8}$	$\frac{5}{8}$	1
3			0	$\frac{1}{8}$	$\frac{3}{8}$
4				0	$\frac{1}{8}$
5					0

A

	0	1	2	3	4
1	0	1	1	2	2
2		0	2	2	2
3			0	3	3
4				0	4
5					0

R

- The tree created by Algorithm 3.10 are shown in Figure.



More Problems....

More Problems....

- Rod-cutting problem : Design a dynamic programming algorithm for the following problem.
  Find the maximum total sale price that can be obtained by cutting a rod of n units long into integer-length pieces if the sale price of a piece i units long is pi for $i = 1, 2, ..., n$.

- Longest path in a DAG : Design an efficient algorithm for finding the length of the longest path in a dag.
  This problem is important both as a prototype of many other dynamic programming applications and in its own right because it determines the minimal time needed for completing a project comprising precedence constrained tasks.

- Maximum square submatrix Given an $m \times n$ boolean matrix B, find its largest square submatrix whose elements are all zeros.
  The algorithm may be useful for, say, finding the largest free square area on a computer screen or for selecting a construction site.

- 0-1 Knapsack : Given objects $x_1, \ldots, x_n$, where object x_i has weight w_i and profit p_i (if its placed in the knapsack), determine the subset of objects to place in the knapsack in order to maximize profit, assuming that the sack has capacity M.

More Problems.... (cont...)

- Longest Common Subsequence: Given an alphabet Σ, and two words X and Y whose letters belong to Σ, find the longest word Z which is a (non-contiguous) subsequence of both X and Y.

- blue All-Pairs Minimum Distance : Given a directed graph $G = (V, E)$, find the distance between all pairs of vertices in V.

- Polygon Triangulation: Given a convex polygon $P =< v_0, v_1, \ldots, v_{n-1} >$ and a weight function defined on both the chords and sides of P, find a triangulation of P that minimizes the sum of the weights of which forms the triangulation.

- Traveling Salesperson : given n cities $c_1, \ldots, c_n$, where c_i has grid coordinates (x_i, y_i), and a cost matrix C, where entry C_{ij} denotes the cost of traveling from city i to city j, determine a left-to-right followed by right-to-left Hamilton-cycle tour of all the cities which minimizes the total traveling cost. In other words, the tour starts at the leftmost city, proceeds from left to right visiting a subset of the cities (including the rightmost city), and then concludes from right to left visiting the remaining cities.

- Viterbi's algorithm for context-dependent classification: Given a set of observations $\vec{x}_1, \ldots, \vec{x}_n$ find the sequence of classes $\omega_1, \ldots, \omega_n$ that are most likely to have produced the observation sequence.

More Problems.... (cont...)

- Edit Distance : Given two words u and v over some alphabet, determine the least number of edits (letter deletions, additions, and changes) that are needed to transform u into v.

Summary

1. Computing Binomial Coefficients
2. Path Counting Problem
3. Coin-collecting problem
4. Chained Matrix Multiplication
5. Optimal Binary Search Tree