CHAPTER 2

Algorithm

- Steps taken when solving a problem using a computer:
  - Problem definition and specification
  - Design a solution
  - Testing and documentation
  - Evaluation of the solution

- These steps could overlap
- Not all problems could be solved using a computer. Some difficult problems we could build a simple model and then test it and build on this model more and more sophisticated models.
- Design a solution id finding a suitable algorithm.

Algorithm:

Precise method used by the computer to solve a problem. The algorithm is composed of a finite set of steps each of which require one or more operations.

Characteristics of Algorithm

1. Definite → clear
2. Effective: It can be solved by the person using a pencil and paper within a limit time.

When studying Algorithm, we study:

1. How to design Algorithm
2. How to analysis Algorithm
3. Prove of correctness.
4. How to express Algorithm
5. Test and documentation

Writing structure programs:

1. Local and global variables are defined
2. Should specify input, output variables for function and procedure.
3. Should use indentation
4. Should be divided into well-defined procedures
5. Flow should be forward. Unless it is necessary to do otherwise or looping.
6. Documentation should be clear.

Min-Max application

Divide and conquer

```
Min-Max ( lower , upper ,minD, maxD)
    If ( lower == upper )
        If ( maxD < A[ Upper ] )
            maxD = A[ upper ];
        end if
        if  ( minD > A [ upper ] )
            minD = A[ upper ];
        end if
    else
        mid = ( lower + upper ) /2;
        Min-Max ( lower, mid, minD, maxD);
        Min-Max ( mid+1, upper, minD, maxD);
    end if
end.
```

Time:

$$T(n) = \begin{cases} d & n = 1 \\ \\ 2\ T(n/2) + c & n > 1 \end{cases}$$

$T(n) = 2\ T(n/2) + c$

$T(n/2) = 2T(n/4) + c$

$T(n) = 2\ [\ 2T(n/4) + c\ ] + c$

$T(n) = 2^2\ T(n/2^2) + 2c + c$

$T(n/4) = 2\ T(n/8) + c$

$T(n) = 2^2\ [\ 2\ T(n/8) + c) + 2c + c$

$T(n) = 2^3\ T(n/2^3) + 2^2c + 2c + c$

…

$T(n) = 2^k\ T(n/2^k) + 2^{k-1}\ c + 2^{k-2}\ c + \ldots + c$

$\quad\quad = 2^k\ T(n/2^k) + c\ (2^{k-1} + 2^{k-2} + \ldots + 1)$

$(x^{i-1} + x^{i-2} + x^{i-3} + .. + 1\ )\ \ *\ \ (\ x-1)/(x-1)$

$(\ x^i + \cancel{x^{i-1}} + \cancel{x^{i-2}} + .\ .\ . + \cancel{x} - \cancel{x^{i-1}} - \cancel{x^{i-2}} - .\ .\ .\ - \cancel{x} - 1)\ /\ (x\ -1)$

$(\ x^i - 1\ )\ /\ (\ x - 1\ )$

$T(n) = 2^k\ T(n/2^k) + c\ (\ 2^k\ -1\ )\ /\ (\ 2 - 1)$

Let $2^k = n$

$T(n) = n\ T(1) + c\ (\ n - 1)$

$T(n) = d\ n + c\ n - c$

$T(n) = O(n)$

Dynamic Programming

Combines solutions to subproblems to obtain a final solution.

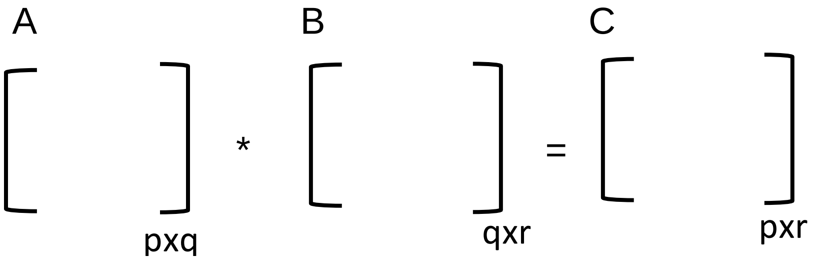
Approach taken:

1. Characterize the structure of optimal solution.
2. Recursively define the value of the optimal solution
3. Compute the value in the above fashion
4. Find the optimal solution

Multiplication of chain matrices

```
Multiply ( A, B)
    if ( columns( A ) ≠ rows( B ))
        error;
    else
        for ( i =1; i <= rows( A ); i++)
            for ( j = 1; j <= columns( B ); j++)_
                c [ i, j ] = 0;
                for ( k = 1; k <= columns( A ); k++)
                    c[ i, j ] = c[ i, j ] + A[ i, k ] * B[ k, j ]
                end for
            end for
        end for
    end if
end.
```

Find the optimal order (least cost = least number of multiplication) to multiply these n matrices.



Cost (A x B) = p x q x r

Example:

A = 10 x 100
B = 100 x 5
C = 5 x 50

A * B * C

- A * (B * C) =

  Cost (B * C) = 100 * 5 * 50 = 25000

  Cost (A * [BC]) = 10 * 100 * 50 = 50000

  Total cost = 25000 + 50000 = **75000**

- (A * B) * C =

  Cost (A * B) = 10 * 100* 5 = 5000

  Cost ([AB] * C) = 10 * 5 * 50 = 2500

  Total cost = 2500 + 5000 = **7500**

$(A_1 * A_2 * A_3 * \ldots * A_k) (A_{k+1} * A_{k+2} * \ldots * A_n)$

- Multiply $A_1 * A_2 * A_3 * \ldots * A_k$
- Multiply $A_{k+1} * A_{k+2} * \ldots * A_n$

Each solved optimally

What is k ?

Optimality part m[I, j] = least possible cost achievable for multiplying $A_i * A_{i+1} * \ldots * A_j$

Idea is

$$\begin{cases} \text{If } (i = j) & \Rightarrow m[i, j] = 0 \\ \text{If } (i < j) & \Rightarrow m[i, j] = \min\limits_{i \le k \le j} \{ m[i, k] + m[k+1, j] + p_{i-1} * p_k * p_j \} \\ \text{If } (i > j) & \Rightarrow x \end{cases}$$

$$(A_i * \underbrace{A_{i+1} * A_{i+2}}_{p_{i-1} \text{ x } p_k} * \ldots * A_k) (\underbrace{A_{k+1} * A_{k+2}}_{p_k \text{ x } p_j} * \ldots * A_j)$$

$$\underbrace{p_{i-1} \text{ x } p_k \qquad\qquad p_k \text{ x } p_j}_{p_{i-1} \text{ x } p_k \text{ x } p_j}$$

A * B * C * D * E * F

A = 4x2 B = 2x3 C = 3x1 D = 1x2 E = 2x2 F = 2x3

	A	B	C	D	E	F
A	0	24	B 14	D 22	D 26	D 36
B		0	6	D 10	D 14	D 22
C			0	6	D 10	D 19
D				0	4	F 10
E					0	12
F						0

A * B * C =

- (A * B) * C = 24 + 12 = 36
- A * (B * C) = 8 + 6 = 14

A * B * C * D

- (A .. C) * D = 14 + 0 + 8 = 22
- (A B) * (C D) = 24 + 6 + 24 = 54
- A * (B . . . D) = 0 + 10 + 16 = 26

(B C) * D = 6 + 0 + 4 = 10

B * (C D) = 0 + 6 + 12 = 18

[A (B C)] [(D E) F]

Algorithm:

```
for ( i = 1; i <= n; i++ )
    cost [ i ][ i ] = 0;
for ( i = 1; i <= n; i++)
    for ( j = i+1; j <= n; j++)
        cost [ i ][ j ] = maxInt;
for ( i = 1; i <= n -1; i++ )
    for ( j = 1; j <= n - i; j++ )
        for ( k = j +1; k <= i + j; k++ )
            t = cost[ j ][ k-1 ] + cost[ k ][ i+j ] + r[ j ] * r[ k ] * r[ i * j + 1]
            if ( t < cost[ j ] [ i + j ] )
                cost[ j ] [ i + j ] = t;
                best[ j ] [ i + j ] = k
            end if
        end for
    end for
end for
```

A$_{4x2}$ **B**$_{2x3}$ **C**$_{3x1}$ **D**$_{1x2}$ **E**$_{2x2}$ **F**$_{2x3}$

r =

4	2	3	1	2	2	3

Example: Longest Common Subsequence Problem (LCS)

Given a string x = < x_1, x_2, . . ., x_n >

z = < z_1, z_2, … z_n>

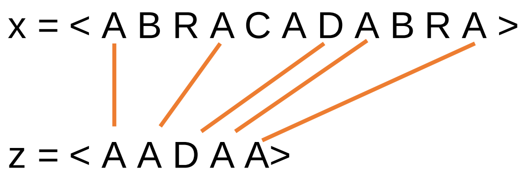
z is a subsequence of **x** if

there is a strictly increasing sequence of k indices < i_1, i_2, …, i_n>

1≤ i_1 < i_2 <… < i_k … ≤ n such that

Z = < x_{i1}, x_{i2}, …, x_{ik} >

Example:



x = < A B R A C A D A B R A >

z = < A A D A A>

Is **z** a subsequence of **x**?

Yes 5 indices are < 1, 4, 7, 8, 11 >

- Given two string **x** and **y**, the longest common subsequence of **x** and **y** a longest string **z** such that **z** is a subsequence of **x** and a subsequence of **y.**

Example:

Given two sequences x = < A B C >

y = < B A C >

z_1 = < A C >

z_2 = < B C >

idea:

let c [i, j] = length of longest common subsequence of x_i and y_j

c [i, 0] = 0

c [0, j] = 0

c [i, j] = ?

$x = < x_1, x_2, . . ., x_i >$, $y = < y_1, y_2, . . ., y_j >$

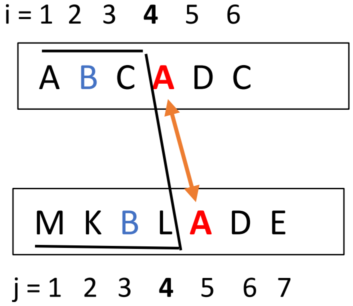
$$c[i, j] = \begin{cases} 0 & \text{if } (i = 0) \text{ OR } (j = 0) \\ c[i-1, j-1] + 1 & \text{if } x_i = y_j \\ \max(c[i-1, j], c[i, j-1]) & \text{if } x_i \neq y_j \end{cases}$$

Example:

x = < A B C A D C >

y = < M K B L A D >

if $x_i = y_j$

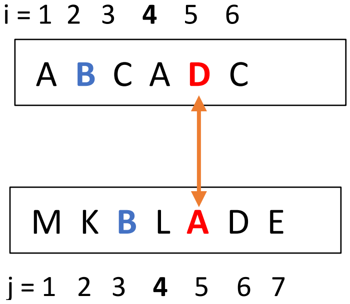


i = 4, j = 5

cost [3][4] = 1

cost [4][5] = cost[3][4] + 1= 2

if $x_i \neq y_j$



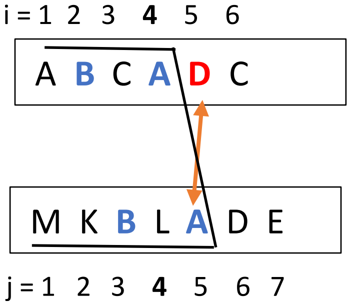
i = 5, j = 5

cost [5][5] =

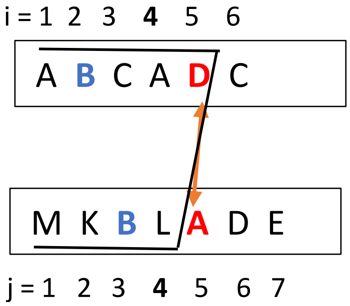
max (cost [4][5], cost[5][4])

= max(2, 1) = 2

Cost [4][5] = 2



Cost [5][4] = 1



Algorithm:

```
m = length (x);
n = length (y);
for ( i = 1; i <= m; i++)
    c [ i ] [ 0 ] = 0;
for ( j = 1; j <= n; j++)
    c [ 0 ] [ j ] = 0;

for ( i = 1; i <= m; i++)
    for ( j = 1; j <= n; j++)
        if ( x[ i ]  == y[ j ] )
            c[ i ] [ j ]  = c[ i-1 ][ j-1 ] +1;
            b[ i ][ j ] = '↖'
        else
            if ( c[ i ][ j -1 ]  >  c[ i-1 ][ j ] )
                c[ i ] [ j ]  = c[ i ][ j-1 ] ;
                b[ i ][ j ] = '←';
            else
                c[ i ] [ j ]  = c[ i-1 ][ j ] ;
                b[ i ][ j ] = '↑ '
            end if
        end if
    end for
end for
```

```
print_LCS ( b, x, i, j )
    if ( ( i == 0 ) or ( j == 0 ) )
        return
    else
        if ( b[ i ][ j ] = '↖' )
            ~~System.out.println( x[ i ] );~~     // Reverse order
            print_LCS ( b, x, i-1, j-1 );
            System.out.println( x[ i ] );     // Normal order
        else
            if ( b[ i ][ j ] = ' ↑ ' )
                print_LCS ( b, x, i-1, j );
            else
                print_LCS ( b, x, i, j-1 );
            end if
        end if
    end if
end.
```

Example:

x = < A B C B D A B >

y = < B D C A B A >

	0	**B**	**D**	**C**	**A**	**B**	**A**
0	0	0	0	0	0	0	0
A	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
B	0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
C	0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
B	0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
D	0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
A	0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4
B	0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4

z = < B C A B >

z = < B C B A >

Greedy Strategy

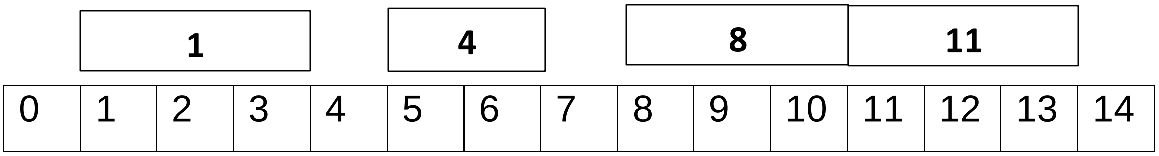
A **greedy algorithm** is any algorithm that follows the problem-solving heuristic of making the locally optimal choice at each stage. In many problems, a greedy strategy does not usually produce an optimal solution, but nonetheless a greedy heuristic may yield locally optimal solutions that approximate a globally optimal solution in a reasonable amount of time.

- Not always optimal solution
- It is a quick solution

Example:

Procedure Greedy selective activity

Job	Start	Finish
i	s_i	f_i
1	1	4
2	3	5
3	0	6
4	5	7
5	3	8
6	5	9
7	6	10
8	8	11
9	8	12
10	2	13
11	11	14



```
j = 1;
for ( i = 2; i <= n ; i++)
    if ( s[ i ] >= f[ j ] )
        A = A U { i };
        j = i;
    end if
end for
```

Knapsack Problem

- 0 – 1 Knapsack: take all of them or nothing
- Fractional knapsack: We can take a part from any item

Example:

3 Items ➔ 50 kg

Item	kg	Price
1	10	60 NIS
2	20	100 NIS
3	30	120 NIS

0 – 1 Knapsack

Weight = item 2 + item 3 = 20 + 30 = 50 kg

Profit = 100 + 120 = 220 NIS

Fractional Knapsack

Item	kg	Price	Price/kg
1	10	60 NIS	6 NIS
2	20	100 NIS	5 NIS
3	30	120 NIS	4 NIS

Weight = item 1 + item 2 + item 3 (10 kg) = 10 + 20 + 20 = 50 kg

Profit = 60 + 100 + 80 = 240 NIS

Example:

The knapsack problem is a problem in combinatorial optimization: Given a set of items, each with a weight and a value, determine the number of each item to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible. It derives its name from the problem faced by someone who is constrained by a fixed-size knapsack and must fill it with the most valuable items.

Weight: $w_1, w_2, \dots, w_n$

Profit: $p_1, p_2, \dots, p_n$

Capacity: M

Find x1, x2, …, xn

To maximizing $\sum_{i=1}^{n} x_i \cdot p_i$, $\sum_{i=1}^{n} x_i \cdot w_i <= M$

Dynamic Programming Approach

C [i][j] = Optimal profit using only $w_1, w_2, \dots, w_j$ and knapsack capacity is i.

$= \max \{ c[i][j-1], \ p_j + c[i-w_j, j] \}$

```
for ( i = 0; i <= m ; i++ )
        c[ i ][ 1 ] = p[ 1 ] * ( i / w[ i ] );
for ( j = 2; j <= n; j++ )
        for ( i = 1; i <= m; i++)
                if ( i – w[ j ] >= 0 )
                        if ( c[ i ][ j-1 ] < p[ j ] + c[ i- w[ j ], j ] )
                                c[ i ][ j ] = p[ j ] + c[ i- w[ j ], j ];
                        else
                                c[ i ][ j ] = c[ i ][ j-1 ];
                        end if
                end if
        end for
end for
```

w : 3 4 7 8 9

p: 4 5 10 11 13

m = 17

w	3	4	7	8	9
p	4	5	10	11	13
Item	1	2	3	4	5
0	0	0	0	0	0
1	0	0	0	0	0
2	0	0	0	0	0
3	**4**	**4**	**4**	4	4
4	4	5	5	5	5
5	4	5	5	5	5
6	8	8	8	8	8
7	8	9	10	10	10
8	8	10	10	11	11
9	12	12	12	12	13
10	12	13	**14**	14	14
11	12	14	15	15	15
12	16	16	16	16	17
13	16	17	18	18	18
14	16	18	20	20	20
15	20	20	20	21	21
16	20	21	22	22	23
17	20	22	**24**	**24**	**24**

Item 3 + Item 3 + Item 1

Weight = 7 + 7 + 3 = 17

Profit = 10 + 10 + 4 = 24

Item 5 + Item 4

Weight = 9 + 8 = 17

Profit = 13 + 11 = 24

Another Solution

One dimensional array

```
for( i = 0; i<= m; i++)
{
        c[i] = p[i] * ( i /w[i]);
        b[i] = 1;
}
for ( j=2; j<= n; j++)
        for( i = 1; i<= m; i++)
        {
                if ( i – s[i] >= 0)
                        if ( c[i] < p[i] + c[ i – w[i]] )
                        {
                                c[i] = p[i] + c[i-w[i]];
                                b[i] = j;
                        }
                }
```

Space = O(m)

It is a good algorithm or not ?

Depend on m, if m is not an integer it will be a big problem, in this case the problem is called NP compute problem.

To print the items

For example:

If i = 17 ➔ c[17] = 24 ➔ b[17] = 5, b[8] = 4

17 – w[5] = 17 – 9 = 8

8 – w[4] = 8 – 8 = 0