INTRODUCTION

Objective:

- Algorithms
- Techniques
- Analysis.

Algorithms:

Definition: An algorithm is a sequence of computational steps that take some value, or set of values, as input and produce some value, or set of values, as output.

Pseudocode: An easy way to express the idea of an algorithm (very much like C/C++, Java, Pascal, Ada, …)

Techniques

- Divide and Conquer
- The greedy method
- Dynamic programming
- Backtracking
- Branch and Bound

Analysis of Algorithms

🡻 Motivation:

- Estimation of required resources such as memory space, computational time, and communication bandwidth.
- Comparison of algorithms.

🡻 Model of implementation:

- One-processor RAM (random-access machine) model.
- Single operations, such arithmetic operations & comparison operation, take constant time.

🡻 Cost:

- Time complexity:
  - ✓ total # of operations as a function of input size, also called running time, computing time.
- Space complexity:
  - ✓ total # memory locations required by the algorithm.

Asymptotic Notation

↳ Objective:

- What is the rate of growth of a function?
- What is a good way to tell a user how quickly or slowly an algorithm runs?

↳ Definition:

- A theoretical measure of the comparison of the execution of an *algorithm*, given the problem size n, which is usually the number of inputs.

↳ To compare the rates of growth:

- Big-O notation: Upper bound
- Omega notation: lower bound
- Theta notation: Exact notation

1- **Big- O notation:**

- ✓ Definition: F(n)= **O(**f(n)**)** if there exist positive constants C & n0 such that F(n)≤c*f(n) when n≥n0
- ✓ F(n) is an ***upper bound*** of F(n).
- ✓ Examples:

F(n)= 3n+2

What is the big-O of F(n)?

F(n)=O(?)

For 2≤n 3n+2≤3n+n=4n

⇒ F(n)= 3n+2 ≤ 4n ⇒ F(n)=O(n)

Where C=4 and n0=2

F(n)=62^n+n2

What is the big-O of F(n)?

F(n)=O(?)

For $n^2 \leq 2^n$ is true only when n≥4

⇒ $62^n+n^2 \leq 62^n+2^n = 7*2^n$

⇒c=7 n0=4 F(n)≤ $7*2^n$

⇒F(n)= O(2^n)

✓ Theorem: If $F(n)= a_m n^m + a_{m-1}n^{m-1} +...+ a_1 n+a_0$

$$= \sum_{i=0}^{m} a_i n^i$$

Then $f(n)= O(n^m)$

Proof:

$$F(n) \le \sum_{i=0}^{m} |a_i| n^i \le n^m * \sum_{i=0}^{m} |a_i| n^{i-m}$$

Since $n^{i-m} \le 1 \Rightarrow |a_i|\, n^{i-m} \le |a_i|$

$$\Rightarrow \sum_{i=0}^{m} |a_i| n^{i-m} \le \sum_{i=0}^{m} |a_i|$$

$$\Rightarrow F(n) \le n^m * \sum_{i=0}^{m} |a_i| \quad \text{for } n \ge 1$$

$$\Rightarrow F(n) \le n^m * c \quad \text{where } c= \sum_{i=0}^{m} |a_i|$$

$$\Rightarrow F(n)= O(n^m)$$

$$O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3) < O(2^n)$$

2- **Omega notation:**

- ✓ Definition: $F(n)= \Omega (f(n))$ if there exist positive constant c and n_0 s.t.
  $F(n) \geq cf(n)$ For all n, $n \geq no$

- ✓ f(n) is a ***lower bound*** of F(n)

- ✓ Example:

  $F(n)=3n+2$
  Since $2 \geq 0 \Rightarrow 3n+2 \geq 3n$ for $n \geq 1$

  Remark that the inequality holds also for $n \geq 0$, however the definition of Ω requires $no > 0$

  $\Rightarrow C=3$, $no=1 \Rightarrow F(n) \geq 3n$
  $\Rightarrow F(n)= \Omega(n)$

- ✓ Theorem: If $F(n)= a_m n^m + a_{m-1}n^{m-1} +...+ a_1n+a_0 = \sum_{i=0}^{m} a_i n^i$ and $am > 0$
  Then $F(n)= \Omega (n^m)$

  Proof:

$$F(n)= a_m n^m \left[1 + \frac{\mathbf{a_{m-1}}}{\mathbf{a_m}} n^{-1} + ... + \frac{\mathbf{a_0}}{\mathbf{a_m}} n^{-m}\right]$$

$$= a_m n^m \alpha$$

For a very large n, let's say n_0, $\alpha \geq 1$
$\Rightarrow F(n) = a_m n^m \alpha \geq a_m n^m$

$\Rightarrow F(n)= \Omega(n^m)$

3- **Theta notation:**

- ✓ Definition: $F(n)= \theta (f(n))$ if there exist positive constants C1, C2, and n0 s.t. $C1\ f(n) \le F(n) \le C2\ f(n)$ for all $n \ge no$

- ✓ F(n) is also called an exact bound of F(n)

- ✓ Example1:

  $F(n)= 3n+2$

  We have shown that $F(n) \le 4n$ & $F(n) \ge 3n$

  $\Rightarrow 3n \le F(n) \le 4n$
  $\Rightarrow C1=3,\ C2=4,$ and $n0=2$
  $\Rightarrow F(n)= \theta(n)$

- ✓ Example2:

  $$F(n) = \sum_{i=0}^{n} i^k$$

  Show that $F(n)= \theta (n^{k+1})$

  Proof:

  $$F(n)= \sum_{i=0}^{n} i^k \le \sum_{i=0}^{n} n^k = n * n^k = n^{k+1}$$

  $\Rightarrow F(n)= O(n^{k+1})$

$$F(n) = \sum_{i=0}^{n} i^k$$

$$= 1 + 2^k + \ldots + (\frac{n}{2} - 1)^k + (\frac{n}{2})^k + (\frac{n}{2} + 1)^k + \ldots + n^k$$

$$= \alpha + \beta$$

where

$$\alpha = 1 + 2^k + \ldots + (\frac{n}{2} - 1)^k + (\frac{n}{2})^k$$

$$\beta = (\frac{n}{2} + 1)^k + \ldots + n^k$$

$$\Rightarrow \beta = (\frac{n}{2} + 1)^k + \ldots + n^k \geq (\frac{n}{2})^k + \ldots + (\frac{n}{2})^k = (\frac{n}{2})^k * \frac{n}{2}$$

$$\Rightarrow F(n) \geq (\frac{n}{2})^k * \frac{n}{2} = \frac{n^{k+1}}{2^{k+1}}$$

$$\Rightarrow F(n) \geq \Omega(n^{k+1})$$

$$\Rightarrow \Omega(n^{k+1}) \leq F(n) \leq O(n^{k+1})$$

$$\Rightarrow$$

$$\boxed{F(n) = \theta(n^{k+1})}$$

Summary:

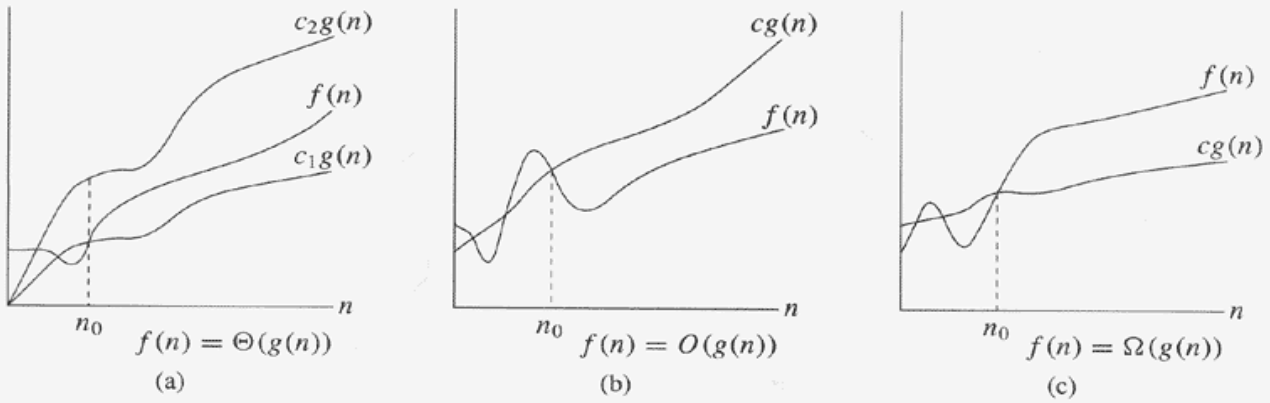

Figure 3.1 Graphic examples of the Θ, O, and Ω notations. In each part, the value of n_0 shown is the minimum possible value; any greater value would also work. **(a)** Θ-notation bounds a function to within constant factors. We write $f(n) = \Theta(g(n))$ if there exist positive constants n_0, c_1, and c_2 such that to the right of n_0, the value of $f(n)$ always lies between $c_1g(n)$ and $c_2g(n)$ inclusive. **(b)** O-notation gives an upper bound for a function to within a constant factor. We write $f(n) = O(g(n))$ if there are positive constants n_0 and c such that to the right of n_0, the value of $f(n)$ always lies on or below $cg(n)$. **(c)** Ω-notation gives a lower bound for a function to within a constant factor. We write $f(n) = \Omega(g(n))$ if there are positive constants n_0 and c such that to the right of n_0, the value of $f(n)$ always lies on or above $cg(n)$.

4. **Properties:**

Let $T1(n)= O(f(n))$
$T2(n)= O(g(n))$

1- The sum rule:

If $T(n)= T1(n)+ T2(n)$
Then $T(n)= O(\max (f(n),g(n))$

Example:
$T(n)= n^3+n^2 \Rightarrow T(n)= O(n^3)$

2- The product rule:

If $T(n)= T1(n) * T2(n)$
Then $T(n)= O(f(n)*g(n))$

Example:
$T(n)= n * n \Rightarrow T(n)= O(n^2)$

3- The scalar rule:

If $T(n)= T1(n) * k$ where k is a constant,
Then $T(n)= O(f(n))$

Example:
$T(n)= n^2 * \frac{1}{2} \Rightarrow T(n)= O(n^2)$

Be careful

- ✓ Which is better $F(n)=6n^3$ or $G(n) = 90n^2$

- ✓ $F(n)/G(n)= 6n^3/90n^2 =6n/90 =n/15$

Case1:

$$\frac{n}{15} <1 \Rightarrow n<15$$

$$\Rightarrow 6n^3<90n^2$$

$$\Rightarrow F(n) \text{ is better.}$$

Case2:

$$\frac{n}{15} >1 \Rightarrow n>15$$

$$\Rightarrow 6n^3>90n^2$$

$$\Rightarrow G(n) \text{ is better.}$$

Complexity of a Program

↳ Time Complexity:

- Comments: no time.
- Declaration: no time.
- Expressions & assignment statements: 1 time unit a O(1)
- Iteration statements:
  - For i= exp1 toexp2 do
    Begin
    Statements // For Loop takes exp2-exp1+1 iterations
    End;
  - While exp do is similar to For Loop.

↳ Space complexity

- The total # of memory locations used in the declaration part :
  - ✓ Single variable: O(1)
  - ✓ Arrays (n:m) : O(nxm)
- In addition to that, the memory needed for execution (Recursive programs).

Total of 5n + 5 ; therefore O(n) ;

```
PROCEDURE bubble (VAR a: array_type ) ;
VAR   i , j , temp  : INTEGER ;
 BEGIN
1              FOR  i  := 1   TO  n-1    DO
2                     FOR  j  := n    DOWN TO   i    DO
3                            IF    a[ j-1 ]  >    a [ j ]   THEN BEGIN
                              {swap}
4                                    temp   :=  a [j-1];
            5                        a [j-1]  := a [j] ;
            6                        a [j]      := temp
                             END
 END   ( * bubble * ) ;
```

° Line 4,5,6 O (max (1,1,1)) = O (1)
° move up line 3 to 6 still O(1)
° move up line 2 to 6 O((n-i) * 1) = O (n-i)
° move up line 1 to 6 $\sum$ (n-i) = (n-1)n/2 = $n^2/2 - n/2 \Rightarrow$ $O(n^2)$

Later we will see how change it to O(n log n)
Seven computing times are : O(1) ; O(log n) ;O(n); O(n log n); $O(n^2)$; $O(n^3)$; $O(2^n)$

```
° control  := 1 ;
  WHILE  control ≤ n LOOP
        …….
          something  O(1)
         control :=  2  *  control ;
  END LOOP ;
```

- control := n ;
  WHILE control ≠ 0 LOOP
  ……..
  something O(1)
  control := control /2 ; control integer
  END LOOP ;

  O(log n)

- FOR count IN 1. . n LOOP
  control := 1 ;
  WHILE control ≤ n LOOP
  …… ……..
  something O(1)
  control := 2 ∗ control ;
  END LOOP ;
  END LOOP ;

  O(n log n)

- FOR count IN 1. . n LOOP
  control := i ;
  WHILE control > n LOOP
  …… ……..
  something O(1)
  control := control div 2;
  END LOOP ;
  END LOOP ;

Amortized analysis:

Definition:

It provides an absolute guarantees of the total time taken by a sequence of operations. The bound on the total time does not refleth the time required for any individual operation, some single operations may be very expensive over a long sequence of operations, some may take more, some may take less.

Example:

Given a set of k operations. If it takes O(k f(n)) to perform the k operations then we say that the amortized running time is O (f(n)).

Pseudocode Conventions

Variables Declarations

```
Integer X,Y;
Real  Z;
Char C;
Boolean flag;
```

Assignment

```
X= expression;
X= y*x+3;
```

Control Statements

```
If condition:
 Then
        A sequence of statements.
 Else
        A sequence of statements.
 Endif
For loop:
   For I= 1 to n  do
         Sequence of statements.
   Endfor;
While statement:
    While condition do
         Sequence of statements.
    End while.
Loop statement:
    Loop
         Sequence of statements.
    Until condition.
Case statement:
     Case:
          Condition1:   statement1.
          Condition2:   statement2.
          Condition n:   statement n.
          Else       :     statements.
      End case;
```

Procedures:

Procedure name (parameters)
Declaration
Begin
Statements
End;

Functions:

Function name (parameters)
Declaration
Begin
Statements
End;

Recursive Solutions

Definition: A procedure or function, that calls itself, directly or indirectly, is said to be recursive.

General format:

```
Algorithm name(parameters)
Declarations;
Begin
        If (trivial case)
        Then
                Do trivial operations;
        Else
                One or more call name(smaller values of parameters);
                Do few more operations: process sub-solutions;
        End if;
End;
```

Example:

```
 Function  Max-set (S)
  Integer m1, m2;
  Begin
      If the number of elements s of  S=2
      Then
            Return (max(S(1), S(2)) );
       Else
            Begin
                  Split S into two subsets; S1,S2;
                   m1= Max-set (S1)
                   m2= Max-set (S2)
                   Return (max (m1,m2) );
                    End;
         Endif
       End;
```

Elimination of recursion

The standard method of conversion is to simulate the stack of all the previous activation records by a local stack. Thus, assume we have a recursive algorithm F (p1,p2,….,pn) where pi are parameters of F.

(1) Declare a local stack
(2) Each call F (p1,p2,…..,pn) is replaced by a sequence to:
  (a)Push pi, for $l \leq i \leq n$, onto the stack.
  (b) Set the new value of each pi.
  (c)Jump to the start of the algorithm.
(3) At the end of the algorithm (recursive), a sequence is added which:
  (a)Test whether the stack is empty, and ends if it is, otherwise,
  (b) Pop all the parameters from the stack.
  (c) Jump to the statement after the sequence replacing the call.

Example:

```
Procedure    C (X: xtype)
 Begin
        If  P(x) then  M(x)
        Else
          Begin
            S1 (x)
            C  (F(x) )
            S2 (x)
           End
 End
```

```
Non-procedure     C (X: xtype)
  Label     1,2 ;
   Var    s:  stack of x type
    Begin
        Clear  s;
    1: if P(x) then M(x)
        else
            Begin
               S1(x) ; push  x onto s ; x:= F(x);
               Goto 1;
               2: S2(x)
            end;
       if  S is not empty then
       Begin
            pop x from s ;
            goto 2
       End;
     End       {of procedure};
```

Graphs

Definition:

- A graph G consists of two sets, called:
  - Vertices: V
  - Edges : E finite set of pairs of vertices. G= (V,E)
- G is said to be directed graph if the pairs in E are ordered; otherwise, G is an undirected graph.
- If $(u,v) \in E$ then u is adjacent to v.

- Undirected graphs:
  - Degree: The degree of a vertex is the number of its adjacent vertices.
  - Theo: Let G=(V,E), the sum of the degrees of each vertex equals 2|E| where |E| is the #of edges of G.

- Directed graphs: (Digraphs)
  - In-degree: the indegree of a vertex v is the number of edges entring it.
  - Outdegree: A vertex u is the number of edges leaving it.
- Path:
  - A path from v0 to vk is a sequence of vertices v0,v1,…..,vk-1,vk s.t.

    $(v0,v1), (v1,v2),\ldots\ldots,(vk-1,vk) \in E$

  - The length of a path is the # of edges of the path.
  - A path is simple if all the vertices in the path, except possibly the first & the last are distinct.
  - A cycle is a simple path in which the first & the last vertices are the same.

- Connected graphs:

  An undirected graph is connected if every pair of vertices is connected by a path.

  ° Strongly connected graphs:

  A directed graph is strongly connected if for every two vertices (i,j) there exists a path

  From I to j & a path from j to i.

  ° Complete graph:

  Is an undirected graph in which every pair of vertices is adjacent.

  ° Graph representations:

  1) Sequential representation: - Adjacency matrix.
  2) Linked list representation: - Linked list.