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: $f(n)= \mathbf{O(}g(n)\mathbf{)}$ if there exist positive constants c & n_0 such that $f(n) \le c*g(n)$ when $n \ge n_0$
- ✓ g(n) is an ***upper bound*** of f(n).
- ✓ <u>Examples:</u>

$f(n)= 3n+2$

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

$$f(n)=O(?)$$

For $2 \le n$ $\quad 3n+2 \le 3n+n=4n$

$\Rightarrow \quad f(n)= 3n+2 \le 4n \Rightarrow f(n)=O(n)$

Where $c=4$ and $n_0=2$

$f(n)=62^n+n^2$

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

$$f(n)=O(?)$$

For $n^2 \le 2^n$ is true only when $n \ge 4$

$\Rightarrow 62^n+n^2 \le 62^n+2^n =7*2^n$

$\Rightarrow c=7 \qquad n_0=4 \qquad f(n) \le 7*2^n$

$\Rightarrow f(n)= O(2^n)$

- ✓ Theorem: If $f(n)= a_m n^m+ a_{m-1}n^{m-1} +...+ a_1n+a_0$

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

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: $f(n) = \Omega\,(g(n))$ if there exist positive constant c and n_0 s.t.
$f(n) \ge cg(n)$ For all n, $n \ge n_0$

- ✓ g(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

  $\Rightarrow c=3$, $n_0=1 \Rightarrow f(n) \geq 3n$
  $\Rightarrow f(n)=\Omega(n)$

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

  $$= \sum_{i=0}^{m} a_i n^i \text{ and am} > 0$$

  Then $f(n)=\Omega(n^m)$

Proof:

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

$$= a_m n^m \alpha$$

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- <u>Theta notation:</u>

- ✓ Definition: $f(n)=\theta(g(n))$ if there exist positive constants c1, c2, and n_0 s.t. $c1g(n) \leq f(n) \leq c2g(n)$ for all $n \geq n_0$

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

✓ <u>Example1:</u>

$f(n) = 3n+2$

We have shown that $f(n) \leq 4n$ & $f(n) \geq 3n$

$\Rightarrow 3n \leq f(n) \leq 4n$

$\Rightarrow c1=3,\ c2=4$, and $n_0=2$

$\Rightarrow f(n) = \theta(n)$

✓ <u>Example2:</u>

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

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

Proof:

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

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

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

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

$$= \alpha + \beta$$

where

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

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

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

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

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

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

$$\Rightarrow$$

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

Summary:

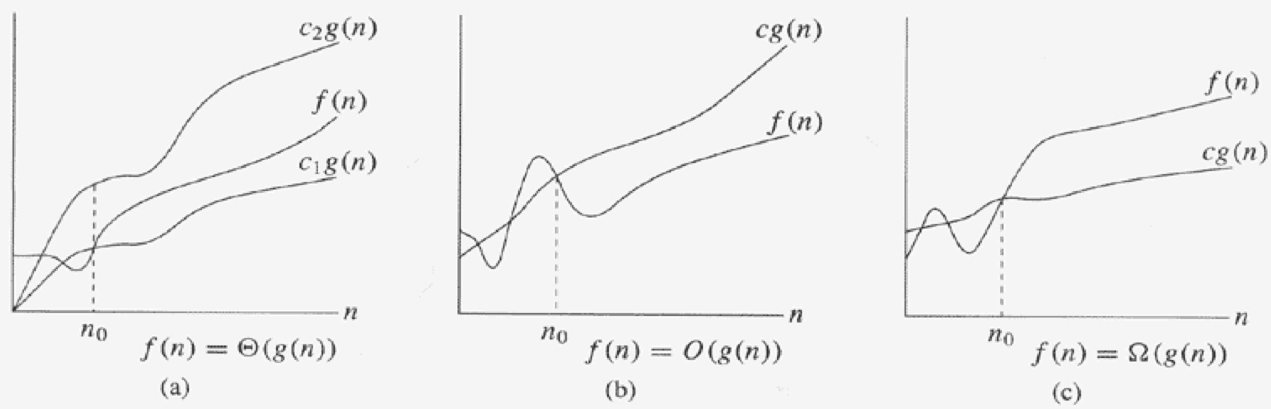


Figure 3.1 Graphic examples of the Θ, O, and Ω notations. In each part, the value of n_0 shown is the minimum possible value; any greater value would also work. **(a)** Θ-notation bounds a function to within constant factors. We write $f(n) = \Theta(g(n))$ if there exist positive constants n_0, c_1, 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) is better.

Case2:

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

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

$\Rightarrow$ G(n) 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;

  $\sum (n-i) = \sum n - \sum i = n^2 - (n-1)n/2 = n^{2-} n^2/2 - n/2 \Rightarrow O(n^2)$

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 >= 1  LOOP
        ……..                                   O(log n)
      something  O(1)
    control :=  control /2  ;  control integer
END LOOP  ;
```

```
◦ FOR  count = 1 to n   LOOP
        control :=  1 ;
      WHILE   control  ≤ n    LOOP
           ……      ……..                              O(n log n)
                  something  O(1)
                control  := 2 ∗ control  ;
      END LOOP  ;
  END LOOP  ;
```

```
◦ FOR  count = 1 to n      LOOP
        control :=  count;
      WHILE   control  >= 1    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.

- **Why recursion**?
  - For many problems, the recursion solution is more natural than the alternative non-recursive or iterative solution
  - It is often relatively easy to prove the correction of recursive algorithms (often prove by induction).
  - Easy to analyze the performance of recursive algorithms. The analysis produces recurrence relation, many of which can be easily solved.

- **Format of a recursive algorithm**:

  - Algorithm name(parameters)

```
Declarations;
Begin
        if (trivial case)
        then do trivial operations
        else  begin
                - one or more call name(smaller values of
                  parameters)
                - do few more operations: process the sub-solution(s).
        end;
end;
```

- **Taxonomy:**

  - **Direct Recursion:**
    - It is when a function refers to itself directly
  - **Indirect Recursion:**
    - It is when a function calls another function which refer to it.
  - **Linear Recursion:**
    - It is when one a function calls itself only once.
  - **Binary Recursion:**
    - A binary-recursive routine (potentially) calls itself twice.

- **Examples**
  - **Find Max**

```
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;
```

- **Printing a stack:**
  - **Recursive method to print the content of a stack**

Z
C
B
A

➔

Printing the stack

Z	**C**	**B**	**A**

```
public void printStackRecursive() {
 if (isEmpty())
        System.out.println("Empty Stack");
 else   {
                System.out.println(top());
                pop();
                if (!isEmpty())
                printStackRecursive();
        }
}
```

- **Palindrome:**

```
int function Palindrome(string X)
Begin
        If Equal(S,StringReverse(S))
        then return TRUE;
        else return False;
end;
```

- **Reversing a String:**
  - **Pseudo-Code:**

```
string function StringReverse(string S)
/* Head(S): returns the first character of S */
/* Tail(S): returns S without the first character */
begin
        If (Length(S) <=1)
        then return S;
        else
                return (concat(StringReverse(Tail(S)) & Head(S));
        endif;
end;
```

- **Performance**

  - **Definition**: A recurrence relation of a sequence of values is defined as follows:
    - (B) Some finite set of values, usually the first one or first few, are specified.
    - (R) The remaining values of the sequence are defined in terms of previous values of the sequence.

  - **Example**:

    - The familiar sequence of factorial is defined as:
      - (B) FACT(0) = 1
      - (R) FACT(n+1) = (n+1)*FACT(n)

  - **Time Complexity:**

    - The analysis of a recursive algorithm is done using recurrence relation of the algorithm.

- **Example 1:**

  - The time complexity of StringReverse function:
    - Let $T(n)$ be the time complexity of the function where n is the length of the string.
    - The recurrence relation is:

      (B) $n=1$ let $T(n)=1$
      (R) $n>1$ let $T(n)=T(n-1)+1$

  - Solution:

$$\begin{aligned}
T(n) &= \cancel{T(n-1)}+1 \\
\cancel{T(n-1)} &= \cancel{T(n-2)}+1 \\
\cancel{T(n-2)} &= \cancel{T(n-3)}+1 \\
&\dots \\
\cancel{T(3)} &= \cancel{T(2)}+1 \\
\cancel{T(2)} &= T(1)+1
\end{aligned}$$

$T(n) = T(1)+1+1+1+...+1 = T(1)+(n-1) = n$

$===> T(n) = O(n$

- **Example 2:**

$$T(n) = \begin{cases} 2T(\frac{n}{2}) + C & if\ n > 1 \\ C & if\ n = 1 \end{cases}$$

Assume that n = 2^k

$T(n) = \cancel{2T(n/2)}+C$

$\cancel{2T(n/2)} = \cancel{2^2T(n/4)}+2C$

$2^2T(n/4) = 2^3T(n/8)+ 2^2C$ ...

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

$2^{k-1}\cancel{T(n/2^{k-1})} = 2^kT(n/2^k)+ 2^{k-1}C$

$T(n) = 2^kT(1)+C+2C+2^2C +...+2^{k-2}C+2^{k-1}C =$

$T(n) = 2^kC +C(1+2+2^2 +...+2^{k-2}+2^{k-1})$

$T(n) = nC +C\sum_{i=0}^{k-1} 2^i = \frac{2^{(k-1)+1}-1}{2-1} = \frac{2^k}{1} = 2^k$

$T(n) = nC +C\frac{2^{(k-1)+1}-1}{2-1}$

$T(n) = nC +C\frac{2^k}{1}$

$T(n) = nC +C2^k$

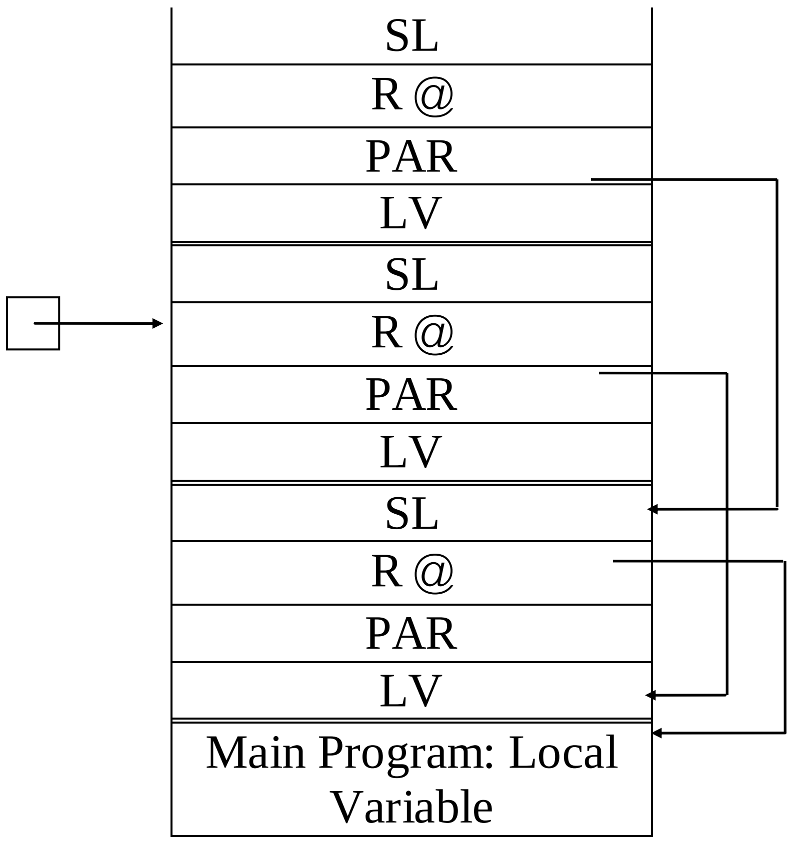
$T(n) = nC +nC=2nC$

$\Rightarrow$ $T(n) =O(n)$

- **Space Complexity:**

    - Each recursive call requires the creation of an activation record
    - Each activation record contains the following:
        - Parameters of the algorithm (PAR)
        - Local variable (LC)
        - Return address (R @)
        - Stack link (SL)

    - **Example**:



- Complexity:
    - Let

        P: parameters

L: local variables
2: SL and R @
n: is the maximum recursive depth

➔ space= n*(P+L+2)

- **Disadvantages**:
  - Recursive algorithms require more time:
    - At each call we have to save the activation record of the current call and Branch to the code of the called procedure
    - At the exit we have the recover the activation record and return to the calling procedure.
    - If the depth of recursion is large the required space may be significant.

- **Exercises:**
  - What is the time complexity of the following function:

$$T(n) = \begin{cases} T(\frac{n}{2}) & n > 1 \\ C & n = 1 \end{cases}$$

  - What is the time complexity of the following function:

$$T(n) = \begin{cases} T\left(\frac{n}{2}\right) + n & n > 1 \\ C & n = 1 \end{cases}$$

  - Write a recursive function that returns the total number of nodes in a singly linked list.

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 ≤i ≤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.

<u>Example:</u>

```
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};
```