OESIGN AND ANALYSIS OF ALGORITHMS

UNIT-1

Basic concept

Algorithm:

Finite set of instructions which are going to provide solution for a particular task

→ Algorithm must satisfy some characteristics

1. Input: zero or more quantities that are supplied externally.
2. Output: Result of an algorithm
3. Effectiveness: Instructions must be feasable
4. Fitness: Terminating point
5. Definiteness: Instructions must be clear and unambiguous

Study of an Algorithm

1. How to devise an algorithm
2. How to validate an algorithm
3. How to analyse an algorithm
4. How to test an algorithm or a program

Testing : 1. debugging
2. profilling → Specification of algorithms

Algorithm specifications:

Algorithm is divided into two sections

1. Algorithm Heading
→ Name of algorithm
→ problem defination
→ inputs
→ out put

Ex: factorial of a number (n) → Heading
calculating factorial value for 'n' → defination
(n) → inputs
factorial for n values → output

Algorithm Body
↳ variables, instructions

→ logical instructions (code)

→ read 'n' values declare fact

→ initialize 'i' value

```
for(i=0; i<n; i++)
  fact = fact * i
  print(fact)
```

Rules for writing an algorithm

1. compound statements are embedded within the flower braces

```
{ statement 1
  statement 2
  statement 3

}
```

2. single line comments

```
// . . . . . \\
```

multiple line comments

```
/* . . . . . . *\
```

Identifiers:
Must begin with a letter alpha numeric string.

→ Assign the values (or) expressions to a variable

```
Ex:  A = 10
     A : = 10
     A = a+b
     A ← a+b
     a+ = b
```

→ arrayName[ ]

conditional statements

```
if(condition){
    statement 1
}
if (condition) then
 { statement 1
   statement 2
 }
if(condition) then
statement
else
statement
```

Looping statements

```
for(i=0; i<n; i++)
i=0 // Initialize
while(i<n) // condition
{
i++; // increment or decrement
}
for variable ← val1 to value n
value = value + 2
value ++
value := initialization value
while (condition) do
{
value ++ /value --
}
```

Algorithm for finding even or odd

Algorithm to find whether number is even or odd

Problem defination : Finding number is even/odd

input : n

output : n is even or odd

```
Read n
if(n%2==0) then
{
write("n is even")
}
else
{
write("n is odd")
}
end if
```

Dowhile:

```
do
{
 i++/i--;
 statements
}while(condition)
break
return;
```

write an algorithm for sorting 'n' elements

```
Ex: 18,2,14,5

  18  2  14  5
   18>2 True
        swap
  2  18  14  5
     18>14 True
           swap
  2  14  18  5
     18>5 True
          swap

  2  14  5  18
```

```
2  14  5  18
   2>14 false
   2>5 false
   2>18 false
   14>2 false
   14>5 true
     swap

2  5  14  18
all the numbers are
   sorted
```

Algorithm

```
Read array[ ]n
for i←1 to n do
for j←1 ton-1 do
{
 if (a[i]>a[j]) then
 {
  temp←a[i]
  a[i]←a[j]
  a[j]←temp
 }
}
```

1. write an algorithm for multiplication of two matrices.

2. write an algorithm for finding even or odd

3. write an algorithm for finding fibonacci series

4. write an algorithm for finding prime or not.

5. algorithm for sum of n elements.

Recursive algorithm

An algorithm call by itself is called recursive algorithm. By using recursive algorithm we can reduce complexity of the algorithm.

Recursive algorithms are two types

1. Direct
2. Indirect

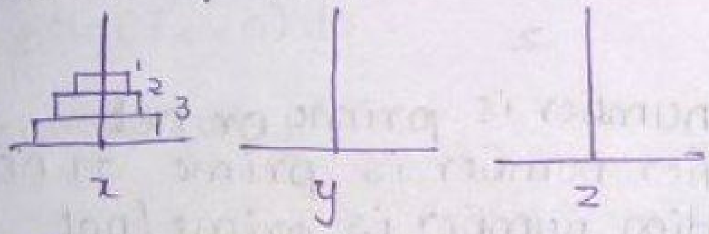
Syntax for Direct recursive algorithm

```
Direct
A( )
{
 A( )
}
```

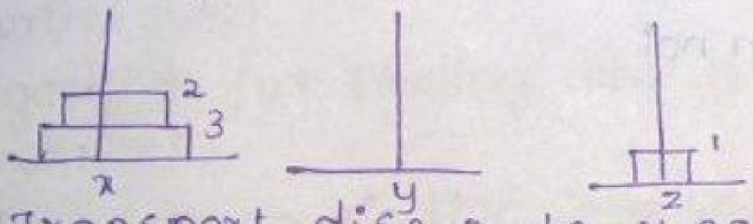
Syntax for Indirect recursive algorithm

```
indirect
A( )
{
 B( )
}
B( )
{
 A( )
}
```

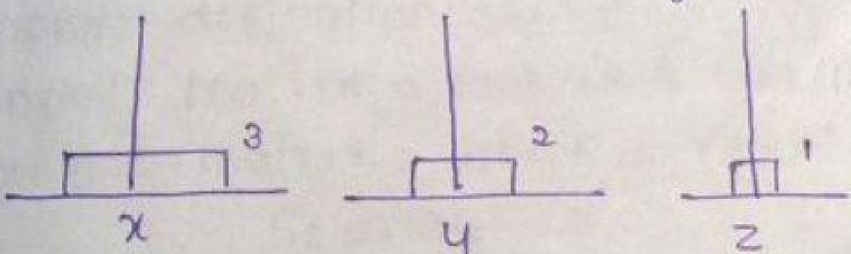
Towers of hanoi



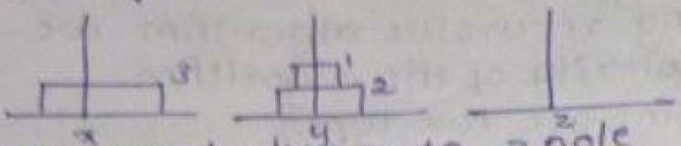
1. Transport disc 1 to z pole



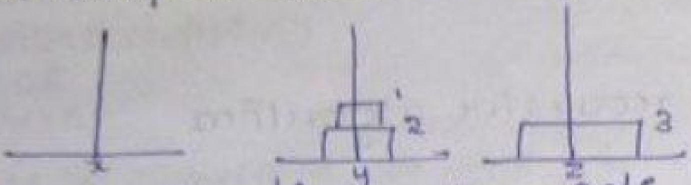
2. Transport disc 2 to y pole



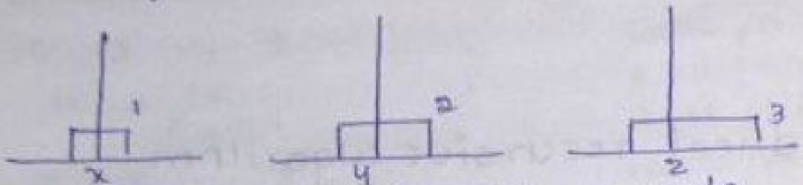
3. Transport disc 1 to y pole



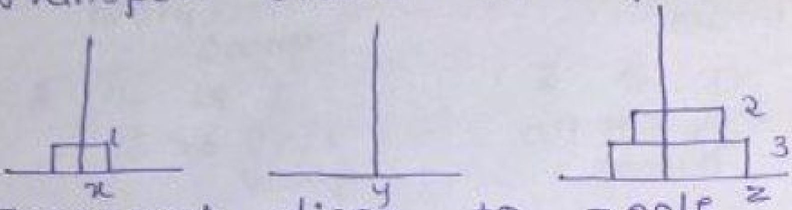
4. Transport disc 3 to z pole



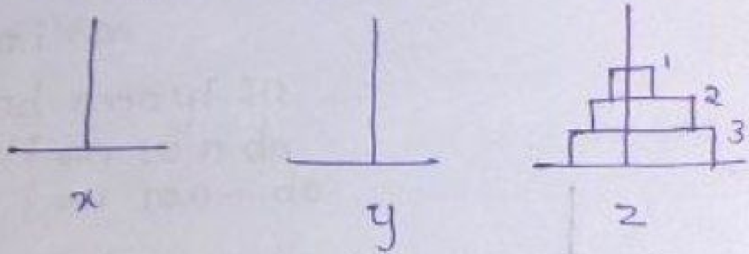
5. Transport disc 1 to x pole



6. Transport disc 2 to z pole



7. Transport disc 1 to z pole



Algorithm for finding number is prime or not

Algorithm to find whether number is prime or not

problem defination: Finding number is prime/not

input : n

output : n is prime or not

```
Read n
count=0, i=1
for i←1 to n do
{ if (n%i==0) then
 {
 count ++
 }
} if (count==2) then
{ write("n is prime")
}
```

```
else
{
write("not prime")
}
```

Algorithm for finding fibonacci series

Algorithm for finding fibonacci series
problem defination: finding fibonacci series
input : n
output : fibonacci series

```
Read n
f=0, s=1
write(f,s)
for i←3 to n do{
t←f+s
write(t)
f←s
s←t
}
```

Algorithm for sum of n elements

Algorithm for finding sum of n elements
problem defination: finding sum of n elements
input : n
output: sum of n elements

```
Read n, i=1
while(i<=n) do
{
sum ← sum + i
i++
}
end while
write(sum)
```

Algorithm for finding multiplication of two matrices

Algorithm for finding multiplication of two matrices
problem defination: Multiplication of two matrices
input: matrix a, matrix b (array[]a array[]b)
output: matrix c which is result of a,b
i.e array[]c

```
Read array[][]a, array[]]b
for i ← 2 to 3 do
{
for j ← 1 to 3 do
{
array[][]c ←
 sum = a[i][k] * b[k][j]
```

Performance analysis of an algorithm

The performance of an algorithm can be determined by space complexity and time complexity

Space complexity:

Must occupy less amount of memory space

Ex:

space complexity = c + s(p)

↓ fixed variable

↓ dependent variable

```
A()
{
int a,b
B()
}
```

→ 2+0
= 2bytes

Here a,b are fixed variables

Ex:

```
sum(x,y)
{
total: =0
for i ← 1 to n do
total : = total + x[i]
}
```

space complexity = c + s(P)
= 3 + n
↓(x,y,total)

Time complexity:

Total time that is required to execute the algorithm

Time complexity = compile time + execution time

Ex: Find sqrt of 'n' value

```
for(i=0; i<n; i++)
{
N=n*n
}
N
Return N*n
```

Time complexity = n-1

Asymptotic notations

Mathematical way to represent time complexity

Bigoh(O)
smalloh(o)
Big omega(Ω)
small omega(ω)
Theta(θ)

Linear search

k=2

1, 3, 5, 6, 7, 2 → worst case
2, 1, 4, 6, 7, 8 → best case
3, 4, 1, 2, 5, 6 → average case

Notations

O(Bigoh):

$f(n), g(n)$

no, c → any constant
↓
some input value of n

Bigoh specifies upper bound of the algorithm

$$\boxed{\begin{array}{l} f(n) = O g(n) \\ \text{if } f(n) \leq c * g(n) \end{array}}$$

Ex:

Let $f(n) = 2n+2$, $g(n) = 3n^2$

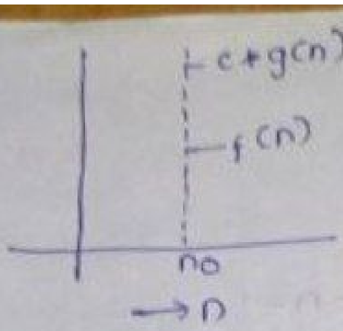
$n = 1$

$f(1) = 4 \quad g(1) = 3$

$f(1) > g(1)$ Not satisfied

$f(2) = 6 \quad g(2) = 12$

$f(2) < g(2)$ condition satisfied $(f(n) < g(n))$ $\forall n \geq 2$



Ω(omega):
omega represents lower bound of an algorithm

f(n), g(n)
no, c

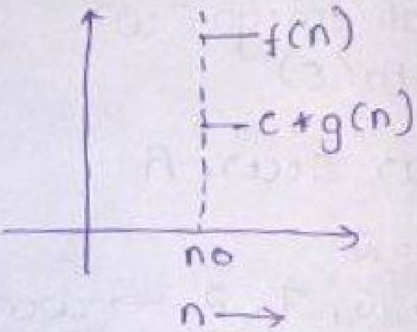
$\boxed{\text{if } f(n) \geq c * g(n)}$

$f(n) = 2n + 2$ $g(n) = 3n^2$

n = 1

$f(1) = 4$ $g(1) = 3$

$f(1) \geq g(1)$
condition satisfied
$\forall n \geq 1$



Theta(θ):
Theta notation specifies both upper bound and lower bound of the algorithm.

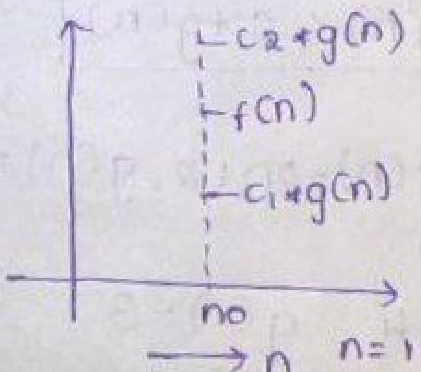
f(n), g(n)
c1, c2, no

$$\underbrace{c_1 * g(n) \leq}_{\text{lower bound}} f(n) \overbrace{\leq c_2 * g(n)}^{\text{upper bound}}$$

$5n^2 \leq 5n^2 + 2 \leq 5n^2 + 3$

n = 1
$5 \leq 7 \leq 8$

$f(n) = \Theta g(n) \ \forall n \geq no$



Smalloh(o): /Littleoh(o)

$$\underset{n \to \infty}{Lt} \frac{f(n)}{g(n)} = 0$$

Smallomega(ω):

when $f(n) > c * g(n)$

Order of growth:

$o(1) < o(\log n) < o(n) < o(n \log n) < o(n^2) < o(2^n) < o(n!)$

↓ minimum time complexity

↓ maximum time complexity

if $n = 8$

$o(1) < o(\log 8) < o(8) < o(8 \log 8) < o(64) < o(256)$
$< o(40320)$

Eg

$f(n) = n^3 + n^2 + 5 \quad o(n^3)$

$g(n) = n^4 + n^3 + n^2 + 1 \quad o(n^4)$

$\max[o(n^3), o(n^4)]$

$f(n) + g(n) = o(n^4)$

Analysis of an algorithm:

In analysis of an algorithm we have 4 techniques

1. Amortized analysis
2. Aggregate analysis
3. Accounting method
4. potential method

Amortized analysis:

Finding the average running time per operation over a sequence of operations

Aggregate analysis:

Amortized cost = $T(n)/N$

$T(n)$ = time required to run sequence of operations.

N = number of operations

when the cost of amortized analysis $T(n)/n$ then that is aggregate analysis.

Accounting method:

Amortized cost calculated per operation then that is called accounting method.

Assume

Actual cost is (c)

Amortized cost (c')

if $c' < c$ credits are used

if $c' > c$ credits are stored

Potential method:

calculate the potential energy stored in datastructures $D_0, D_1, D_2, \ldots D_n$

Actual cost $c_1, c_2, \ldots, c_n$

Amortized cost

= actual cost + potential charge

$= c_i + [\phi(d_i) - \phi(d_{i-1})]$

when $\phi(d_i) - \phi(d_{i-1}) > 0$ then that is effective algorithm

Divide and conquer Method

General Method:

problem 'p' divided into sub problems and each sub problem is solved independently, combined the solutions of all subproblems into a single solution.

Algorithm:

→ If the sub problem is large then divide and conquer method is reapplied.

→ In this method recursive algorithms are used.

Control abstraction / Algorithm:

```
Algorithm D and C(P)
{ if small(p) then
return s(p);
else
{
```

```
divide P into smaller instances P1, P2, P3 ---, Pk k≥1;
Apply D and C to each subproblems;
return combine (D and C(P1), D and C(P2), ----
                D and C(Pk));
}
}
```

→ By using recurrance relation we calculate the concluding time of divide and conquer method. computing

$$T(n) = \begin{cases} g(n) & \text{if } n \text{ is small} \\ T(n_1) + T(n_2) + \dots + T(n_r) + F(n) & \text{if } n \text{ is large} \end{cases}$$

wher $T(n_1)$ = total time required to compute n_1 sub problem.

$T(n_2)$ = total time required to compute n_2 sub problem.

$F(n)$ = total time required to the problem into subproblems and combine the solution of subproblems into a single solution.

→ Reccurence relation is a relation which defines some sequence of equation recursively

$T(n) = T(n-1) + r \longrightarrow$ ① General form

$T(0) = 0 \longrightarrow$ ② Initial form

→ If we want to divide a problem of size 'n' into a size of '$\frac{n}{b}$' taking $F(n)$ computing time to divide and combine the subproblems and solutions then the recurrence relation for obtaining the computing time of size 'n' is

$$T(n) = aT(n/b) + F(n)$$

a = no. of subproblems

Reccurance relation:

The reccurance relation is an equation thad defines a sequence recursively relation is The general form of reccurance

$$T(n) = T(n-1) + r \rightarrow ①$$

$$T(0) = 0 \rightarrow ②$$ Initial condition/form

Eq ① is Reccurance relation

→ The reccurance relation have infinite no.of sequences.

→ The reccurance relation can be solved by using two methods.

1. Substitution method
2. Masters method

1. Substitution method:

The substitution method is a method in which a guess is made for the solution. There are 2 types of substitution methods

(i) Forward substitution

(ii) Backward substitution

Forward substitution:

This method makes use of initial condition to generate the initial term and next term is generated based on the initial term. this process is continued until some formula is guessed.

Ex:

$T(n) = T(n-1) + n \rightarrow ①$

with initial condition $T(0) = 0 \rightarrow ②$

if $n = 1$

$T(1) = T(0) + n$

$T(1) = n = 1$

if $n = 2$

$T(2) = T(1) + 2$

$= 1 + 2$

$= 3$

if $n = 3$

$T(3) = T(2) + 3$

$= 3 + 3$

$= 6$

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

By observing above generated equations we can derive a formula

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

$$= \frac{n^2+n}{2}$$

we can also denote in terms of bigoh(o) notation as

$$T(n) = O(n^2)$$

Backward substitution:

In this method backward values are substituted recursively to derive formula.

Ex:

$T(n) = T(n-1)+n \longrightarrow ①$

$T(0)=0$ - initial condition

If $n=n-1$

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

$T(n-1) = T(n-2)+n-1 \longrightarrow ②$

sub ② in ①

$T(n) = T(n-2)+n-1+n$

$T(n) = T(n-2)+2n-1 \longrightarrow ③$

If $n=n-2$

$T(n-2) = T(n-3)+n-2 \rightarrow ④$

sub ④ in ③

$T(n) = T(n-3)+n-2+2n-1$

$T(n) = T(n-3)+3n-3 \rightarrow ⑤$

If $n=n-3$

$T(n-3) = T(n-4)+n-3 \rightarrow ⑥$

sub ⑥ in ⑤

$T(n) = T(n-4)+n-3+3n-3$

$T(n) = T(n-4)+4n-6$

$$T(n) = T(n-k)+kn-\frac{k(k-1)}{2}$$

$n=k$

$$T(k) = T(k-k)+kk-\frac{k(k-1)}{2}$$

$$= T(0)+k^2-\frac{k^2+k}{2} = 0+\frac{k^2+k}{2}$$

Time complexity = $O(k^2)$ / $O(n^2)$

Ex:

$T(n) = T(n-1) + 1 \rightarrow$ ①

$T(0) = 0$

Forward substitution

if $n=1$

$T(1) = T(0)+1$

$T(1) = 1 \rightarrow$ ②

if $n=2$

$T(2) = T(1)+1$

$T(2) = 2$

if $n=3$

$T(3) = T(2)+1$

$= 3$

if $n=4$

$T(4) = T(3)+1$

$= 3+1 = 4$

$T(n) = n$

Time complexity = $O(n)$

Backward substitution

if $n = n-1$

$T(n-1) = T(n-2)+1 \rightarrow$ ②

sub ② in ①

$T(n) = T(n-2)+2 \rightarrow$ ③

if $n = n-2$

$T(n-2) = T(n-3)+1 \rightarrow$ ④

sub ④ in ③

$T(n) = T(n-3)+3 \rightarrow$ ⑤

if $n = n-3$

$T(n-3) = T(n-4)+1 \rightarrow$ ⑥

sub ⑥ in ⑤

$T(n) = T(n-4)+4$

Now

$T(n) = T(n-k)+k$

Now $n=k$

$T(k) = T(k-k)+k$

$T(k) = T(0)+k$

$T(k) = k$

Time complexity = $O(k)$ / $O(n)$

2. Masters Method:

In this method the basic recurrance relation equation is

$$T(n) = a.T(n/b) + F(n)$$

where $a \geq 1$, $b > 1$ be constants

Let $F(n)$ be a function and $T(n)$ define non-negative integers

→ $T(n)$ can be boundes as follows

case 1:

if $F(n) = O(n^{\log_b^{a-\varepsilon}})$ (or) if $F(n) < n^{\log_b^a}$

for some constant $\varepsilon > 0$ then

$$T(n) = \Theta(n^{\log_b^a})$$

case 2:

if $F(n) = O(n^{\log_b^a})$ then or $F(n) = n^{\log_b^a}$

$$T(n) = \Theta(n^{\log_b^a} . \log n)$$

case 3:

if $F(n) = \Omega(n^{\log_b^{a+\varepsilon}})$ and if $a.F(n/b) \leq c * F(n)$ or $F(n) > n^{\log_b^a}$

then

$$T(n) = \Theta(F(n))$$

Ex:

$T(n) = 9T(n/3) + n$

$T(n) = aT(n/b) + F(n)$

$a = 9, b = 3, F(n) = n$

$n^{\log_b^a}$

$= n^{\log_3^9} = n^{2\log_3^3}$

$n^{\log_b^a} = n^2$

$F(n) = n$

$n < n^2$

$F(n) < n^{\log_b^a}$

case (i) is applied

$T(n) = \Theta(n^{\log_b^a})$

$T(n) = \Theta(n^2)$

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

$T(n) = a \cdot T(n/b) + F(n)$

$a = 2, b = 2, F(n) = n^3$

$n^{\log_b^a} = n^{\log_2^2}$

$= n^1$

$F(n) = n^3$

$n^3 > n$

$F(n) > n^{\log_b^a}$

case iii is applied

$T(n) = \Theta(F(n))$

$= \Theta(n^3)$

Binary search:

It is an efficient searching method while searching the elements using this method. The elements in the array should be sorted. an element which is to be search from the list of elements store in array and the searched element is called key element.

→ In this technique first find out the middle element A[m] then 3 conditions need to be tested with the key element.

i. If key = A[m]
then the searched element is in the list.

ii. If key < A[m]
then search the left sublist

iii. If key > A[m]
then search the right sublist

Algorithm:

Name of the algorithm:

Algorithm Binary Search (A[0 ... n], key)

problem discription:

This algorithm is for searching the element by using binary search method.

input: an array A where the key element is searched.

output: It returns the index of an array element if it is equal to key otherwise it returns -1.

```
low ← 0
high ← n-1
while(low<high) do
{
m←(low + high)/2;
if(key==A[m]) then
return m;
else if(key<A[m])then
high ←m-1;
else
low ← m+1;
}
return -1;
```

→

0	1	2	3	4	5	6	7	8	9
5	7	9	13	32	33	42	54	56	88

↓mid ↓m+1

key=33

$mid = \frac{0+9}{2} \simeq 4$

key==A[m] $\boxed{33==32}$ ×

key<A[m] $\boxed{33<32}$ ×

key>A[m] $\boxed{33>32}$ ✓

mi

low=m+1

5	6	7	8	9
33	42	54	56	88

low ↓m-1 ↓m high

$m = \frac{5+9}{2} = 7$

key==A[m] $\boxed{33==54}$ ×

key<A[m] $\boxed{33<54}$ ✓

high=m-1

5	6
33	42
low	high

mid = $\frac{5+6}{2} \cong 5$

key == A[m]

$\boxed{33 == 33}$ ✓

return 5

→

0	1	2	3	4	5	6	7	8	9
5	7	9	13	32	33	42	54	56	88
L				↓mid					H

mid = $\frac{0+9}{2} \simeq = 4$

key = 8

key == A[mid]
32 == 8
False

key < A[mid]
8 < 32
True
Now
high = mid - 1

0	1	2	3
5	7	9	13
L	↓mid		H

mid = $\frac{0+3}{2} \simeq 1$

key == A[mid]
8 == 7
False

key < A[mid]
8 < 7
False

key > A[mid]
8 > 7
True
Now
low = mid + 1

2	3
9	13
↙mid L	H

mid = $\frac{2+3}{2} \simeq 2$

key == A[mid]
8 == 9
False

key < A[mid]
8 < 9
True
high = mid - 1
searching is stopped
unsuccesful search
'8' is not present in given list

→ The basic operation in binary operation is comparision of search key with the array elements to analyse effeciency of binary search we must cound the number of times. The key gets compared the array elements

→ In the algorithm after one comparision the list of n elements are divided to n/2 sublist. The worst case efficiency is that the algorithm compares all the array elements for searching desired element. In this one comparision is made and based on this comparision array is divided each time into n/2 sublist. Hence the worst case time complexity is given by

$$C_{worst}(n) = C_{worst}\left(\frac{n}{2}\right) + 1 \text{ for } n > 1 \longrightarrow \textcircled{1}$$

$C_{worst}\left(\frac{n}{2}\right)$ = computing time required to compare left sublist or right sublist

1 → one comparision is made with middle element but as we consider the bounded value when array gets divided the above situation can be written as

$$C_{worst}(1) = 1 \longrightarrow \textcircled{2}$$

→ Assume $n = 2^k$

substitute in ①

$$C_{worst}(2^k) = C_{worst}\left(\frac{2^k}{2}\right) + 1$$

$$= C_{worst}(2^{k-1}) + 1 \longrightarrow \textcircled{3}$$

using backward substitution method we can substitute

$$C_{worst}(2^{k-1}) = C_{worst}(2^{k-2}) + 1 \longrightarrow \textcircled{4}$$

sub ④ in ③

$$C_{worst}(2^k) = C_{worst}(2^{k-2}) + 2$$

$$\vdots \qquad \vdots$$

$$C_{worst}(2^k) = C_{worst}(2^{k-k}) + k$$

$$C_{worst}(2^k) = C_{worst}(2^0)+k$$
$$= C_{worst}(1)+k$$
$$= 1+k$$

$$C_{worst}(n) = 1+\log_2^n$$
$$n=2^k$$
$$=1+\log_2^{2^k}$$
$$=1+k\log_2^2$$

$$C_{worst}(n) = 1+k$$

Time complexity $= O(\log_2^n)$

Advantages of Binary search.

→ This method results is an optimal searching algorithm which can search the desired element very efficiently.

Disadvantage

This algorithm requires the sorted list then only this method is applicable.

Applications:

→ This method is an efficient method to search the desired records from the database

→ For solving non-linear equations with one unknown value

Merge sort:

The merge sort is a sorting algorithm that uses divide and conquer stratagy in this method the division is done dynamically

→ Merge sort consists of 3 steps

1. Divide:
partition the array into two sublists

2. Conquer:
sort the each sublist

3. combine:
Merge solutions of sublists into single list

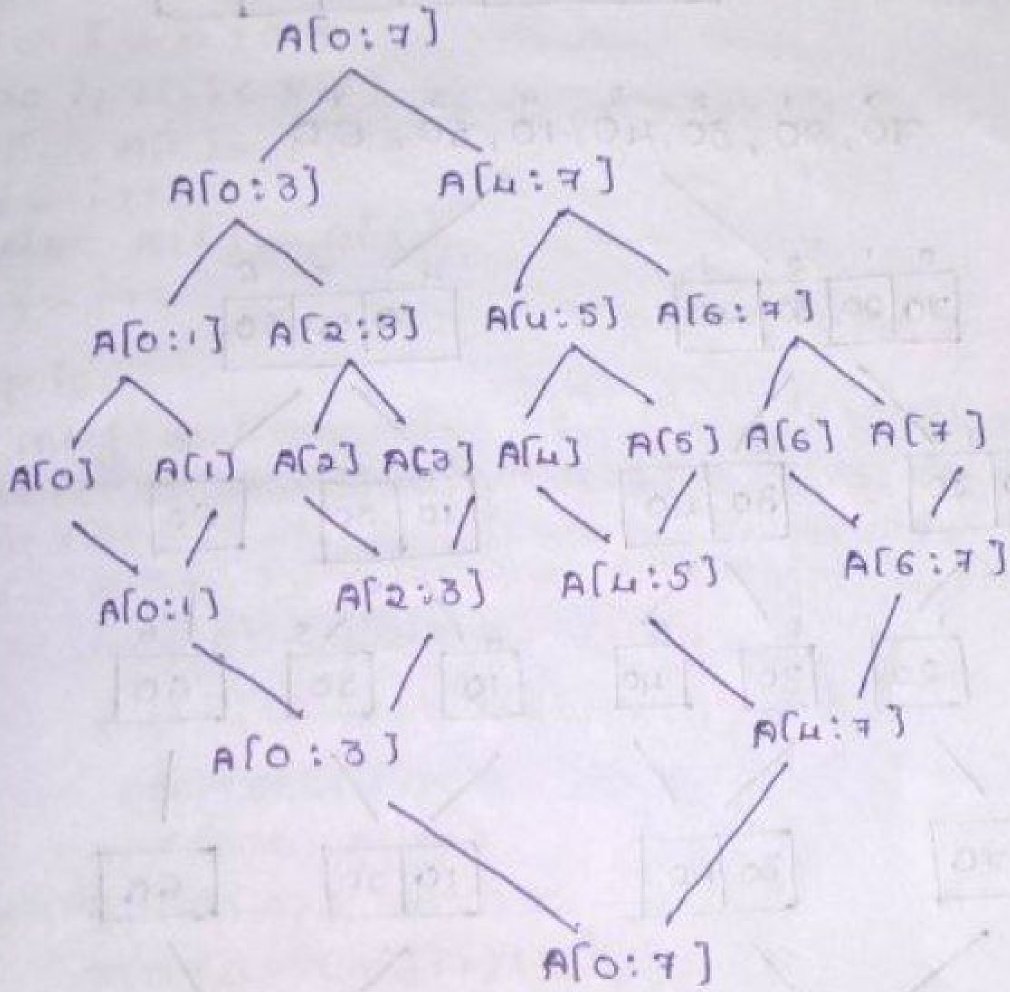
→ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

mid

mid = $\frac{0+7}{2} \simeq 3$

0	1	2	3

mid = $\frac{0+3}{2} = 1$

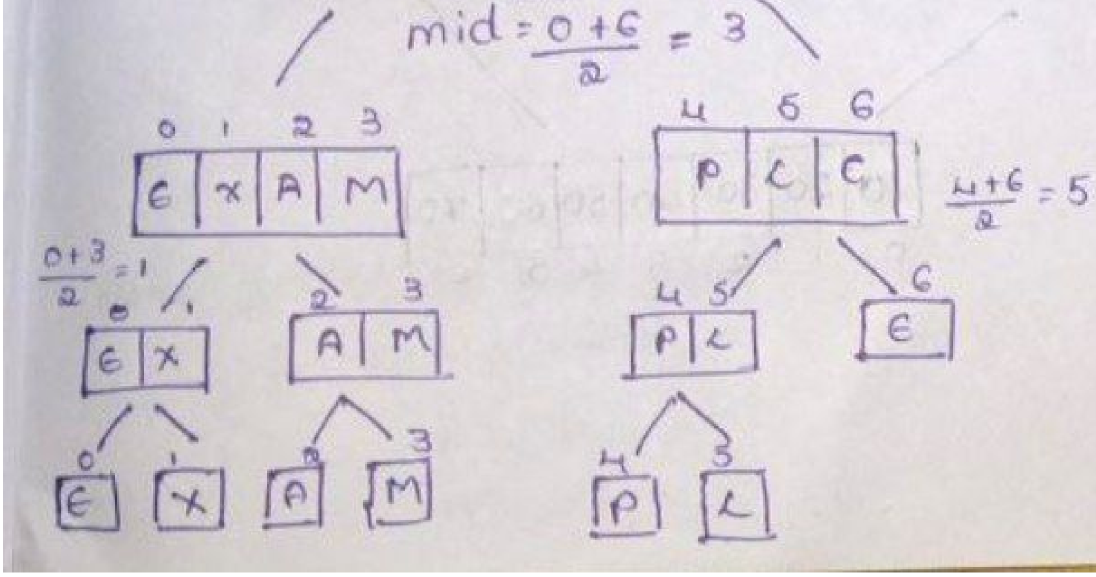


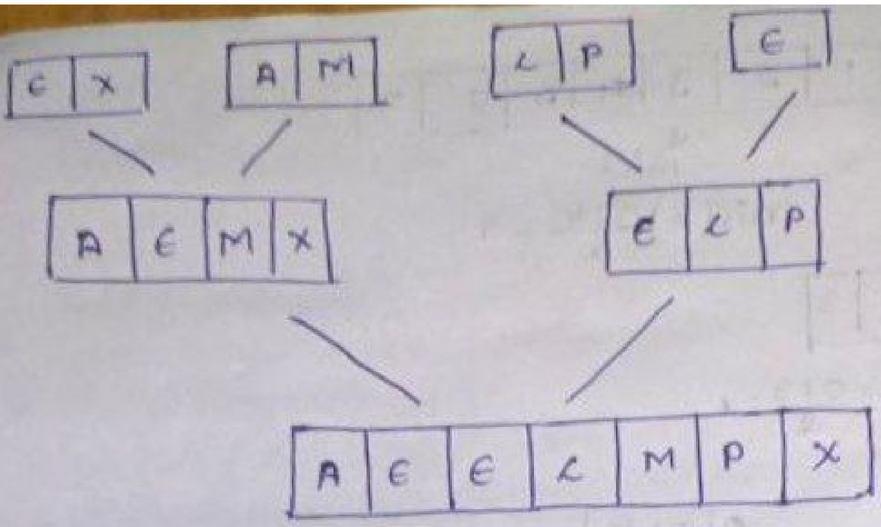
→ A = | E | X | A | M | P | L | E |

0 1 2 3 4 5 6

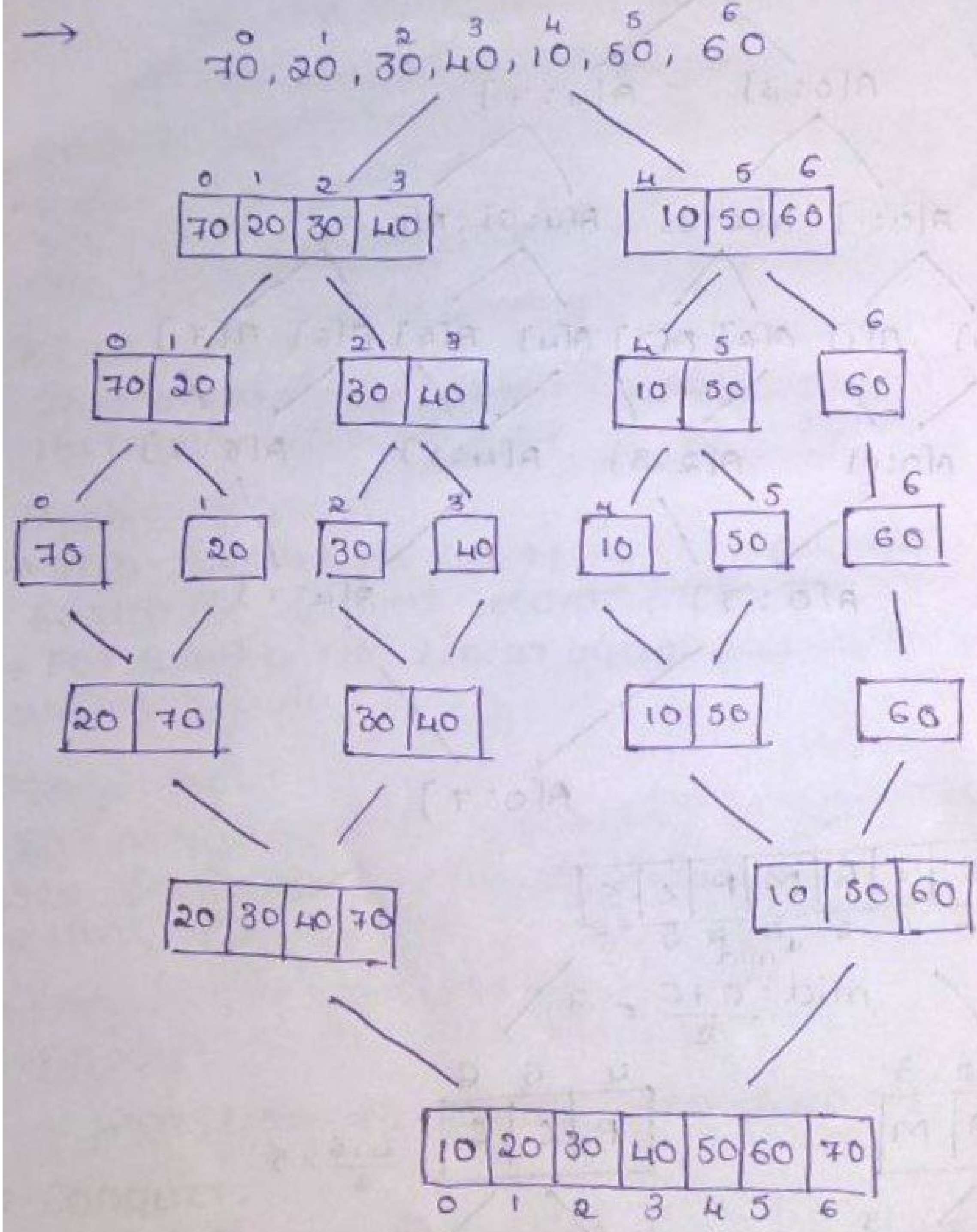
mid

mid = $\frac{0+6}{2} = 3$




E X
A M
L P
E
A E M X
E L P
A E E L M P X

→


0 1 2 3 4 5 6
70, 20, 30, 40, 10, 50, 60
0 1 2 3
70 20 30 40
4 5 6
10 50 60
0 1
70 20
2 3
30 40
4 5
10 50
6
60
0
70
1
20
2
30
3
40
4
10
5
50
6
60
20 70
30 40
10 50
60
20 30 40 70
10 50 60
10 20 30 40 50 60 70
0 1 2 3 4 5 6

Algorithm:

```
Algorithm Merge(A,P,q,r)
compute $n_1$ & $n_2$
compute the first $n_1$ elements into $L[1,\ldots,n_1+1]$ and the next $n_2$ elements
    into $R[1,\ldots n_2+1]$
$L[n_1+1] \leftarrow \infty$; $R[n_2+1] \leftarrow \infty$
$i \leftarrow 1$; $j \leftarrow 1$
for $k \leftarrow p$ to $r$
    do if $L[i] \leq R[j]$
        then $A[k] \leftarrow L[i]$
            $i \leftarrow i+1$
        else $A[k] \leftarrow R[j]$
            $j \leftarrow j+1$
```

Analysis:

In merge sort algorithm two recursive calls are made. Each recursive call focuses on n/2 elements of the list and one call is made to combine two sublists i.e., to merge all 'n' elements hence the reccurance relation is

$$c(n) = c(n/2) + c(n/2) + cn \quad , \text{for } n>1$$

$$c(n) = 2c(n/2) + cn$$

$$c(1) = 0, \text{ for } n=0$$

By using Master's theorm

$$T(n) = a \cdot T(n/b) + F(n)$$

$$a=2, n=n, b=2, F(n) = \underset{\substack{\downarrow \\ c \text{ is constant}}}{c} \cdot n = n$$

$$n^{\log_b a} = n^{\log_2 2} = n$$

$$F(n) = n$$

$$n = n$$

$$F(n) = n^{\log_b a}$$

Here case ii is satisfied

$$\text{if } F(n) = O(n^{\log_b a})$$

$$n = O(n^{\log_2 2})$$

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

$T(n) = O(n^{\log_2^2} \cdot \log n)$

$T(n) = O(n \cdot \log n)$

Time complexity of merge sort

Substitution Method:

$c(n) = c(n/2) + c(n/2) + c \cdot n \rightarrow ①$

$c(1) = 0 \longrightarrow ②$

$c(n) = 2c(n/2) + cn \longrightarrow ③$

apply backward substitution method

$n = 2^k$ in Eq ③

$c(2^k) = 2c(2^k/2) + c \cdot 2^k$

$c(2^k) = 2c(2^{k-1}) + c \cdot 2^k \longrightarrow ④$

put $k = k-1$

$c(2^{k-1}) = 2c(2^{k-2}) + c \cdot 2^{k-1} \rightarrow ⑤$

sub Eq ⑤ in ④

$$
\begin{aligned}
c(2^k) &= 2\cancel{c}(2c(2^{k-2}) + c \cdot 2^{k-1}) + c \cdot 2^k \\
&= 4c \cdot 2^{k-2} + 2c \cdot 2^{k-1} + c \cdot 2^k \\
&= c \cdot (2^k) + c \cdot 2^k + c \cdot 2^k \\
&= 2^3 c(2^{k-3}) + 3c2^k \\
&= 2^4 c(2^{k-4}) + 4c2^k \\
&\vdots \\
c(2^k) &= 2^k \cdot c(2^{k-k}) + kc2^k \\
&= 2^k \cdot c(1) + k \cdot c \cdot 2^k \\
&= 0 + k \cdot c \cdot 2^k
\end{aligned}
$$

$c(2^k) = \underset{\substack{\downarrow \\ \text{constants}}}{k \cdot c} \cdot 2^k$

$c(2^k) = 2^k = n = n \log_n^n$

Time complexity for merge sort

Quick sort:

This technique was invented by Hoare and he is considered that this method to be a fast method to sort the elements. Here the division into two sub-arrays is made so that the sorted sub-arrays donot need to be merge later. This is accomplished by rearranging the elements in an array.

→ In this method the list is divided into two based on the pivot element. Usually the first element is considered as pivot element. Now move the pivot into its correct position in the list. The elements to the left of pivot are less than the pivot and the elements to the right of pivot are greater than the pivot.

i<j : swap i^{th} element and j^{th} element
i>j : swap pivot element and j^{th} element

Example:

```
 1   2   3   4   5   6   7   8   9
65  70  75  80  85  60  55  50  45  +∞
     i                           j
↓ pivot
        i<j => 2<9

65  45  75  80  85  60  55  50  70
         i                   j
        i<j => 3<8

65  45  50  80  85  60  55  75  70
             i           j
        i<j => 4<7

65  45  50  55  85  60  80  75  70
                 i   j
        i<j => 5<6

65  45  50  55  60  85  80  75  70
                 j   i
        i>j => 6>5

60  45  50  55 |65| 85  80  75  70
          ← less      → greater
     1    2    3    4    5    6    7    8
→ 65   45   50   80   48   78   63   90   64
  ↓                                  j
pivot i
          i<j => 2<8

  65   64   50   80   48   78   63   90   45
            i                   j
            i<j => 3<7

  65   90   63   80   48   78   50   45
                 i         j
              i<j => 4<6

  65   90   63   78   48   80   50   45
                      i

 65   45   50   80   48   78   63   90   64
 ↓    i
pivot      i    i    i                  j
 65   64   50   80   48   78   63   90   45
           i                       j
 65   64   90   80   48   78   63   50   45
                i                  j
 65   64   90   63   48   78   80   50   45
                i    i    j
 65   64   90   63   78   48   80   50   45
                     j    i
 78   64   90   63   65   48   80   50   45
 ↓    i
piv                  i    i
                                        j
 78   45   90   63   65   48   80   50   64
           i
 75   45   50   63   65   48   80   90   64
                                   j
                i
 75   45   50   80   65   48   63   90   64
                          j
 75   45   50   80   48   65   63   90   64
                     i    j
                     j    i
 48   45   50   80   75   65   63   90   64
```

```
    1  2  3  4  5  6  7  8  9
→  65 95 70 80 98 58 63 40 25
   ↓   i                     j
 pivot
           i<j

   65 25 70 80 98 58 63 40 95
          i              j
           i<j

   65 25 40 80 98 58 63 70 95
             i        j

   65 25 40 63 98 58 80 70 95
                i  j
           i<j

   65 25 40 63 58 98 80 70 95
                j  i
           i>j
```

~~6~~

$58 \quad 25 \quad 40 \quad 63 \quad \boxed{65} \quad 98 \quad 80 \quad 70 \quad 95$

Analysis:

If the array is always partitioned at the middle then it brings the best case efficiency of an algorithm the reccurance relation of quick sort for obtaining best case is

$$T(n) = T(n/2) + T(n/2) + n \rightarrow T(n) = 2T(n/2) + n \rightarrow ①$$

$$T(1) = 0 \rightarrow ②$$

compare eq① with Master's formula

$$T(n) = a\,T(n/b) + F(n)$$

$$a = 2,\ b = 2,\ F(n) = n$$

$$n^{\log_b^a} = n^{\log_2^2} = n$$

$$n^{\log_b^a} = F(n)$$

It satisfies case (ii)

then

$$T(n) = \theta(n^{\log_b^a} . \log n)$$

$$T(n) = \theta(n \cdot \log n)$$

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

substitution method:

$$T(n) = 2T(n/2)+Tn \longrightarrow ①$$

$$T(1) = 0 \longrightarrow ⑤$$

$$n = 2^k$$

using backward substitution

$$T(2^k) = 2T(2^{k-1}) + T(2^k) \longrightarrow ③$$

if $k = k-1$

$$T(2^{k-1}) = 2T(2^{k-2}) + T(2^{k-1}) \longrightarrow ④$$

sub eq ④ in 3

$$T(2^k) = 2(2T(2^{k-2}) + T(2^{k-1})) + T(2^k)$$

$$= 4T(2^{k-2}) + 2T(2^{k-1}) + T(2^k)$$

$$= 4T(2^{k-2}) + 2T(2^k)$$

$$= 2^2T(2^{k-2}) + 2T(2^k)$$

$$T(2^k) = 2^nT(2^{k-n}) + n \cdot T(2^k)$$

Now $n = k$

$$= 2^kT(2^{k-k}) + k \cdot T(2^k)$$

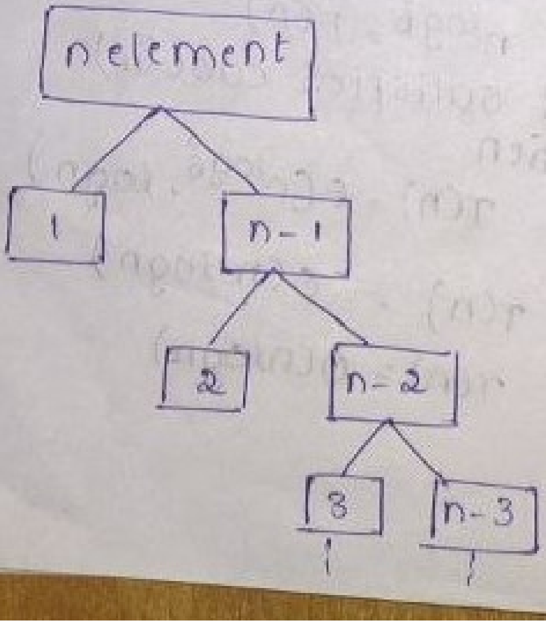
$$= 2^kT(1) + k \cdot T(2^k)$$

$$= 0 + k \cdot T(2^k)$$

$$T(2^k) = k \cdot T(2^k)$$

$$T(n) = \log_n^n \cdot n$$

→ The worstcase time complexity for quick sort occurs when the pivot element is minimum or maximum element of all the elements in the list. This can be graphically represented as



$T(n) = n+(n-1) + (n-2) + (n-3) + \dots + 1$

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

$= \frac{n^2+n}{2}$

time complexity $= n^2$