Scanned with CamScanner

NP What is an algorithm? Unit-I

Algorithm:- The word algorithm comes from the name of persian author Abu Jafar Mohammed ibn musa al khowarizmi

Imp Def: A finite set of step by step instructions to do (or) solve (or) to accomplish a particular task. text book on mathematics

→ Mainly an algorithm should satisfy five criteria's (or) properties

1) Input Def:-
2) Output
3) Definiteness
4) Finiteness
5) Effectiveness

1) Input: 0 (or) more quantities that are supplied externally

2) Output: result of algorithm
Output criteria must produce atleast one quantity as a result.

3) Definiteness: The instruction must be clear and unambiguous

4) Finiteness: The algorithm should execute some set of statements in each case and terminate at one point

5) Effectiveness: The instructions must be feasible. Instruction must be simple and easy. We must be able to write algorithm using pencil and paper.

The study of algorithms is used in many areas the four important (or) major areas to study the algorithm are:

Scanned with CamScanner

Scanned with CamScanner

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

1) How we write an algorithm, gather the requirements, produce the solution, implement the solution using appropriate technique

2) Validate - checking, by giving inputs to the algorithm to check the correctness - whether the algorithm is giving correct output or not

3) analysis - (based on operations) analyse the memory space, how much time it takes to execute

4) test a program (is done by 2 techniques)
- Debugging
- profilling

Debugging: traces the errors, corrects the error but will not check correctness of the result

Profilling: traces the errors, corrects the error and also checks the correctness of result

Algorithm: general english → no specifications
pseudo code: general code + algorithm
Present programmers usually prefer pseudo code technique - Algorithm specifications

→ "Algorithm" is derived from the Persian author "Abu Jafar Mohammed ibn musa al khowarizmi" who wrote books on Mathematics

Scanned with CamScanner

Algorithm Specifications:- (Steps to write algorithm using pseudo code)

Algorithm is divided into two sections

1) Algorithm heading: It consists of name of algorithm (with parameters), problem description, input, output

2) Algorithm body: It consists of logical instruction (Code)

The rules for writing an algorithm:

1) Algorithm is a procedure consisting of head and body. The head consists of name of the algorithm and parameter list

Syntax: Algorithm Name ($P_1, P_2, \ldots P_n$)

Eg: factorial(n, a)

2) Head section Consists the following things:

// problem description: finding the factorial of n numbers

// Input: (n, a)

// output: factorial of number

3) The body of an algorithm written in which various programming constructs like for loop, while condition or some assignment statements (read input, write output)

4) The Compound statements should be enclosed within { }

5) Single line of comments are written using "//" as beginning of comment

6) The identifier should begin by letter and not by digit. An identifier can be a combination of alpha numeric string

Scanned with CamScanner

7) Using assignment operator (←), an assignment statement be given
(or) (:=)

Eg: Variable ← expression

8) There are other types of operators such as boolean operators, arithmetic operators, logical operators and relational operators

9) The array indices are stored within [ ] square brackets. The index of array usually starts at '0'. The multi-dimensional arrays can also be used in algorithm

10) The Inputs and outputs can be done by using read and write statements.

The conditional statements such as if-then or if-else

```
if condition            if (condition) then
then                       statements
   statements           else
                           statements
```

While statement can be written as:

```
while (condition) do
{
 Statement 1
 statement 2
     :
 Statement n
}
```

The general form for writing for loop:

```
for Variable ← Val1 to Val n do   (or) step-1 do
{
statement1
```

(Val1: initialization cond; Val n: termination condition)

Scanned with CamScanner

statement 2

⋮

statement n

}

The general form for writing for loop:

for Variable ← Val1 to Val n do (or) step-1 do

{

statement1

Scanned with CamScanner

⋮

statement n

}

The repeat-until (do-while) statements can be written as

repeat

{

statement 1

⋮

statement n

}

until (Condition)

The break statement is used to exit from inner loop. The return statement is used to return control from one point to another

Examples:

1) Addition of two numbers

Name of the algorithm : add (a, b)

//problem description : Algorithm for addition of two numbers

//Input: a, b

//output : c

Read a, b

c = a+b

print c

Scanned with CamScanner

2) factorial of a number:

Name of the algorithm: Factorial(n)

// problem description: Algorithm for finding the factorial of a given number

// Input: n

// output: factorial of n

```
if (n=1) then
    return(1)
else
return n*factorial(n-1)
```

3) Algorithm to check whether the given number is even or odd

Name of the algorithm: Even or odd(n)

// problem description: Algorithm to check whether the given number is even or odd

// Input: n

// output: ~~even or odd~~ n is even or odd

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

4) Write an algorithm for sorting the 'n' elements

Name of the algorithm: sort(n[ ])

// problem description: algorithm for sorting the elements

// Input: n[ ]

// output:

Scanned with CamScanner

Scanned with CamScanner

Scanned with CamScanner


```
{
for i <- 1 to n do
for j <- 1 to n-1 do
 { if (a[i] > a[j]) then
   {
   temp <- a[i];
   a[i] <- a[j];
   a[j] <- temp;

   }
 }
Write("The sorted array")
}
```

5) Algorithm for finding the multiplication of two matrices

Name of the algorithm: multiplication (a, b)

// Problem description: Algorithm for finding the multiplication of two matrices

// Input: a/b a[ ][ ], b[ ][ ]

// output: c[ ][ ]

```
{
Read a[ ][ ], b[ ][ ]
for i <- 1 to n do
for j <- 1 to n do
   c[i][j] <- 0
for k := 1 to n do
  c[i, j] <- c[i, j] + A[i, k] + B[k, j]
Print c
```

Scanned with CamScanner

// Input : a, b a[ ][ ], b[ ][ ]
// output : c[ ][ ]

```
{
Read a[ ][ ], b[ ][ ]
for i ← 1 to n do
for j ← 1 to n do
    c[i][j] ← 0
for k := 1 to n do
  c[i,j] ← c[i,j] + A[i,k] * B[k,j]
print c
```

9/107

Scanned with CamScanner

2) Algorithm to count the sum of n elements

Name of the algorithm : sum(n)

// problem description : Algorithm to count the sum of n elements

// Input : n

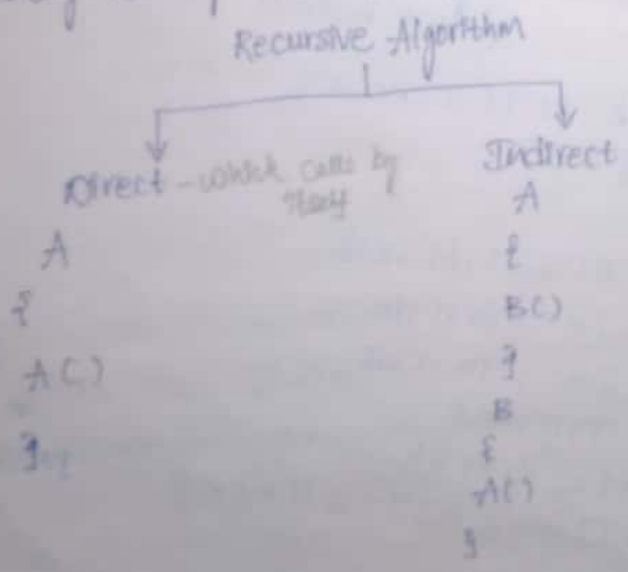
// output : sum of n

```
{
1. Read n
2. i=1, sum=0
3. if (i>n) goto 7
4. S=S+i
5. i=i+1
6. goto 3
7. print S
}
```

Recursive Algorithms:-

An algorithm call by itself is called recursive algorithm

→ Recursion reduces the no. of instruction in the program thereby reducing the complexity



Scanned with CamScanner

Towers of Hanoi (n, a, b, c)
// problem Description : An algorithm to solve towers of Hanoi
// Input : n no. of disks in peg A (Source)
// output : n no. of disks in peg C (destination)

```
for i := 1 to n do
{
  j := i;
  for k := 1+i to n do
  if (a[k] < a[j]) then
    j := k
    t := a[i];
    a[i] := a[j];
    a[j] := t;
  }
}
```

```
{
  if (n≥1) then
  {
    TowersofHanoi(n-1, x, y, z);
    write ("Move top disk from tower", x, "to top of tower", y
    TowersofHanoi(n-1, z, y, x);
  }
}
```

Performance analysis :-

To measure the performance of an algorithm, there are two techniques

a) space complexity
b) time complexity

1. Space Complexity : amount of storage i.e required to execute the algorithm.

-The amount of storage is calculated by

~~Space(s) = C + S(p)~~ $S(P) = C + S_p$

C = Constant
P = instant characteristics

For example:
Name of the Algorithm : add (a, b, c)
// problem description: addition of numbers

Scanned with CamScanner

Scanned with

// Input : a, b, c are of float type
// output: returns addition

return a+b+c

Space $S = C + S(P)$ → variables (a, b, c)

$= 3 + 0$

$= 3$

Algorithm Add (x, n)

{

sum ← 0.0

for i ← 1 to n do

sum ← sum + x[i]

return sum

}

space required : $S = C + S(P)$

$= 3 + n$ → array of n

↓ sum, i, x[ ]

Analysis of Space Complexity:

1. Instruction space
2. Data space
3. stack space

1. Instruction space : The compiler stores the machine code in instruction space

→ It is an fixed part

→ Compiler uses some optimisations techniques to convert

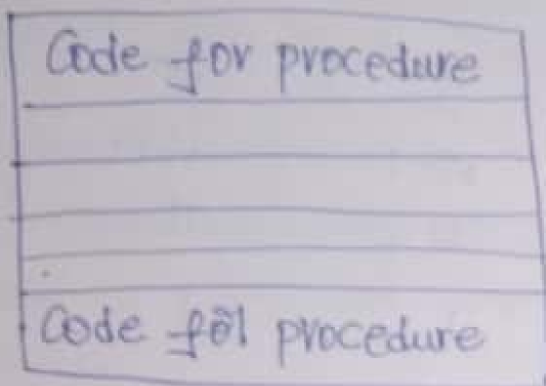
2. Data space : This is occupied by the variables in program

3. Stack Space : Depending on the execution of program are allocated dynamically

Scanned with CamScanner

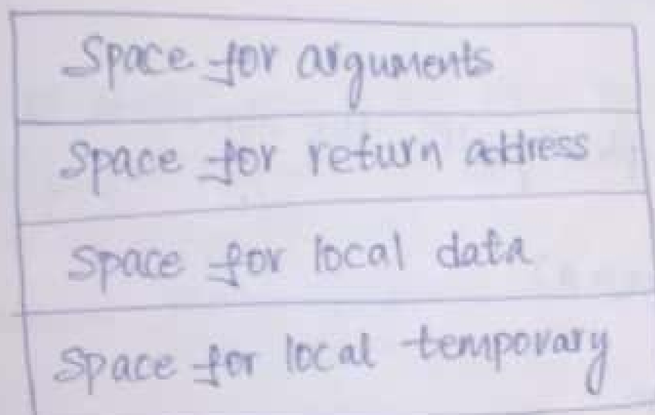
→ Mainly used in functions

Stack frame work



Core memory

Runtime storage



activation record

2) Time Complexity:
The total amount of time that is required to complete the execution of an algorithm is called the time complexity

$S(P) = C + S_P$

$T(P) =$ Compile time + run time

(fixed) (changes based on the no of i/p)

To measure the time complexity, scientists introduced "frequency count"

→ Frequency count is nothing but the number of times an instruction is executed

Scanned with CamScanner

The total amount of time that is required for the execution of an algorithm is called the time complexity

$S(P) = C + S_P$ (changes based on the no. of i/ps)

$T(P) =$ Compile time + run time (fixed)

To measure the time complexity, scientists introduced "frequency count"

→ Frequency count is nothing but the number of times an instruction is executed

13/107

Statement	s/e (steps per execution)	frequency	total steps
Algorithm sum(a,n)	0	—	—
{	0	—	—
s:=0.0;	1	1	1
for i:=1 to n do	1	n+1	n+1
{ s:=s+a[i]; }	1	n	n
return s;	1	1	1
}	0	—	—
total			2n+3

The time complexity for the algorithm is 2n+3

Statement	s/e	frequency (n=0)	frequency (n>0)	total steps (n=0)	total steps (n>0)
Algorithm Rsum(a,n)	0	—	—	0	0
{	0	—	—	0	0
if (n≤0) then	1	1	1	1	1+x
return 0.0;	1	1	0	1	0
else	—	—	—	—	—
return Rsum(a,n-1)+a[n];	1+x	0	1+x	0	1+x
}	—	—	—	—	—
				2	2+2x

Asymptotic Notations:

These are the short hand way representation to represent time complexity

→ We prefer time complexity when compared to space complexity because it is somewhat complex to calculate space complex, as there is a chance to lose data

Bigoh-O → represents upper bound - max time to execute algorithm

Omega - Ω

Scanned with CamScanner

Scanned with CamScanner

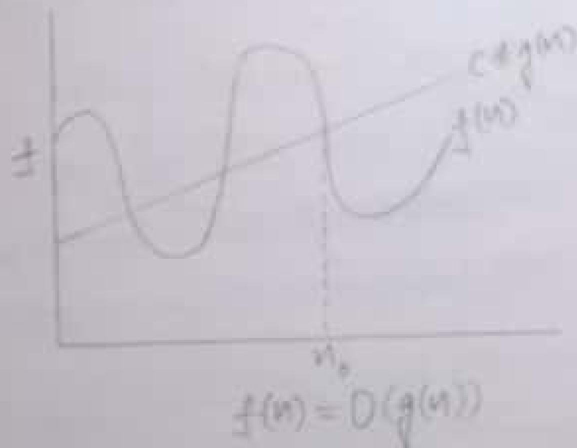
Theta - Θ
Small oh - o
Small omega - ω

→ We prefer Bigoh - O to represent time Complexity

Asymptotic Notation is a short hand way to represent the time Complexity. In asymptotic notations we have several Notations like Bigoh, omega, Theta, small omega and smalloh

i) Bigoh: Bigoh notation denoted by 'O' is a method for representing the upper bound of an algorithm running time

Def: Let f(n) and g(n) are two non-negative function and if there exists an integer n_0 and a constant 'c' such that $c > 0$ and for all integers $n > n_0$, $f(n) < c * g(n)$ then it is denoted as $f(n) = O(g(n))$



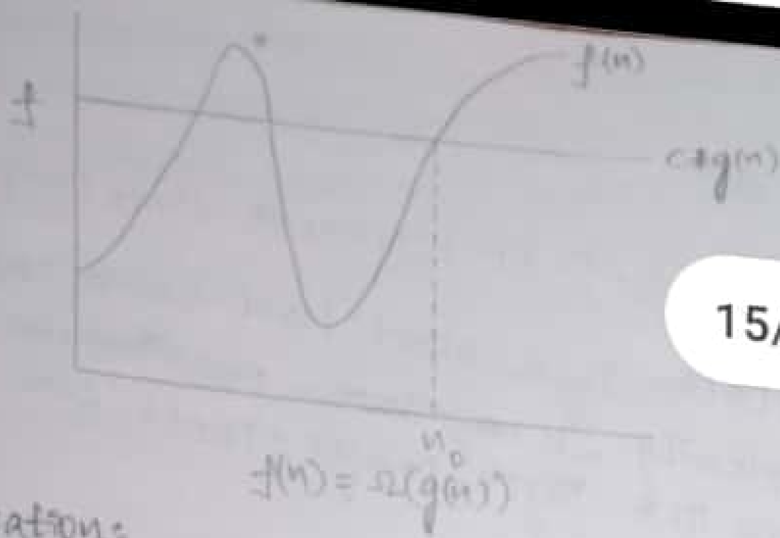
$f(n) = O(g(n))$

Omega Notation:

Omega notation is denoted as 'Ω' is a method of representing the lower bound of algorithm running time

Def: Let f(n) and g(n) are two non-negative functions and there exists a Constant 'c' and an integer 'n_0' such that $c > 0$ & $n > n_0$ then $f(n) > c * g(n)$ i.e $f(n) = \Omega(g(n))$

Scanned with CamScanner



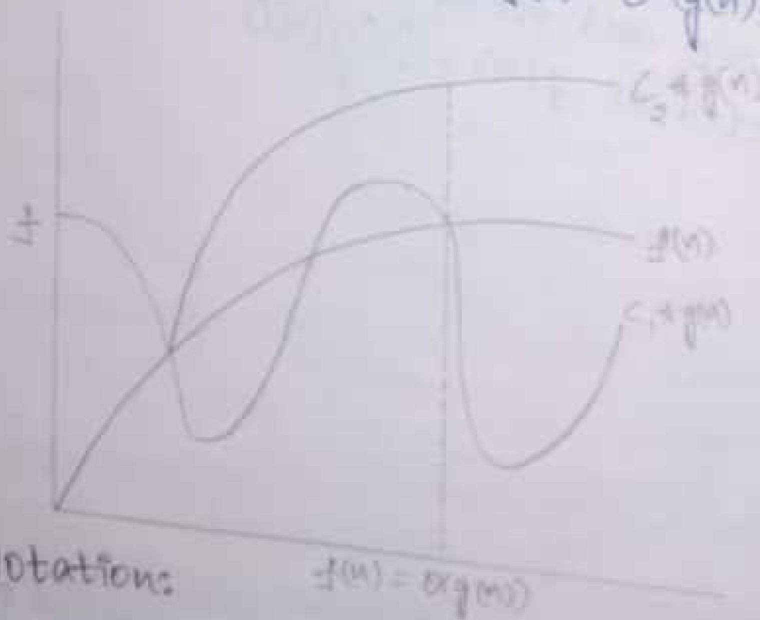
$f(n) = \Omega(g(n))$

15/107

Theta notation:

Theta notation denoted as 'θ' is a method of representing running time between upper bound and lower bound.

Def: Let $f(n)$ and $g(n)$ be two non-negative functions there exists two +ve constants c1 and c_2 such that $c_1 * g(n) \leq f(n) \leq c_2 * g(n)$ and it is denoted as $f(n) = \Theta(g(n))$.



$f(n) = \Theta(g(n))$

Littleoh Notation:

The littleoh is denoted as 'o'. It is defined as:

Let $f(n)$ and $g(n)$ be two non-negative functions such that $f(n) = o(g(n))$

$f(n) = o(g(n))$ iff $f(n) = O(g(n))$ & $f(n) \neq \Theta(g(n))$

Scanned with CamScanner

Littleoh Notation:

The littleoh is denoted as 'o'. It is defined as:

Let f(n) and g(n) be two non-negative functions such that $f(n) = o(g(n))$

$f(n) = o(g(n))$ iff $f(n) = O(g(n))$ & $f(n) \neq \Theta(g(n))$

Little omega Notation:

The little omega is denoted as ω. It is defined as:

Let f(n) and g(n) be two non-negative functions then

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

such that $f(n) = \omega(g(n))$

$f(n) = \omega(g(n))$ iff $f(n) = \Omega(g(n))$ & $f(n) \neq \Theta(g(n))$

Amortized Analysis:

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

Amortized cost is calculated by using three techniques

1) Aggregate Analysis: Here we calculate amortized cost for sequence of operations

n - no. of operations

T(n) - time required to run sequence of operations

Amortized cost/operation $= \frac{T(n)}{n}$

2) Accounting Method: we calculate amortized cost per operation

We compare actual cost with amortized cost

if amortised cost < actual cost - then credits are used

if amortised cost > actual cost - then credits are stored

amortized cost = actual cost + credits (may be used or stored)

3) ~~Potential Method~~: Conditions in calculating amortized cost:

1) $\sum_{i=1}^{n} c_i' \geq \sum_{i=1}^{n} c_i$ (amortized cost ≥ actual cost)

2) Total credits $= \sum_{i=1}^{n} c_i' - \sum_{i=1}^{n} c_i$

Total credits must non-negative then it is effective

if it is -ve, it is not effective

Scanned with CamScanner

Scanned with CamScanner

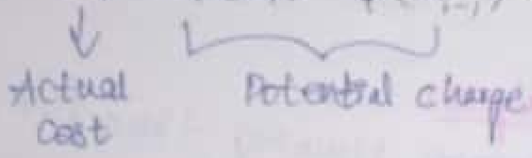
3) Potential Method:

We need to calculate potential energy / Potential. Potential is stored in data structure

If there are 'n' data structures $D_0, D_1, D_2, \ldots D_n$ then

actual cost $= n \rightarrow C_1, C_2, C_3, \ldots C_n$

Amortized cost $= C_i + \phi(D_i) - \phi(D_{i-1})$ (for i^{th} operation)



$$\sum_{i=1}^{n} c_i^{1} = \sum_{i=1}^{n} c_i + \phi(D_n) - \phi(D_0) > 0$$

Divide and Conquer Method:-

General Method:-

In divide and conquer method a given problem is divided into subproblems. These subproblems are solved independently. Combining the solution of all subproblems into a single solution.

If the subproblem is large then divide and conquer method is reapplied.

In this method recursive algorithms are used.

Control abstraction:

```
Algorithm DandC(P)
{
 if Small(P) then
   return S(P);
 else
  {
  divide P into smaller instances P1,P2,P3,----Pk; k>=1;
  Apply DandC to each of these subproblems;
  return Combine (DandC(P1), DandC(P2),-----DandC(Pk));
```

Scanned with CamScanner

Scanned with CamScanner

In this method recursive algorithms are used

Control abstraction:

```
Algorithm DandC(p)
{
if Small(P) then
  return S(P);
else
{
divide P into smaller instances P1,P2,P3,----Pk; k>=1;
Apply DandC to each of these subproblems;
return Combine (DandC(P1), DandC(P2),----DandC(Pk));
}
}
```

Scanned with CamScanner

→ By using recurren relation we calculate the computing time of Divide and Conquer method

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

$T(n_1)$ = total time required to compute 'n_1' subproblem

$T(n_2)$ = total time required to execute 'n_2' subproblem

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

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

a = no. of subproblems

n/b =

Recurrence relation is a relation which defines some sequence of equation recursively

Conditions:

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

$T(0) = 0 \rightarrow ②$ initial term

The computing time of divide and conquer method is given by the recurrencerelation

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

$T(n)$ = the total ~~time~~ computing time for divide and conquer method of size 'n'

$g(n)$ = the computing time required to solve small inputs

$F(n)$ = the computing time required in dividing problem P

Scanned with CamScanner

Scanned with CamScanner

into subproblems and combining the solutions into a single solution.

If we want to divide a problem of size 'n' into a size of 'n/b' taking F(n) computing time to divide and combine the subproblems and solutions, then the recurren relation for obtaining the computing time of size 'n' is

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

Recurrence Relation:

The Recurrence Relation is an equation that defines a sequence recursively. The general form of recurrence relation is $T(n) = T(n-1) + r \longrightarrow ①$

$T(0) = 0 \longrightarrow ②$

eq ① is called recurrance relation

eq ② is called initial condition

The recurrance relation have infinite no. of sequences.

→ The recurrance relation can be solved by using 2 metho

① Substitution method

② Master's 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

a) forward substitution

b) backward Substitution

a) Forward substitution:

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

Scanned with CamScanner

Scanned with CamScanner

methods

a) forward substitution

b) backward substitution

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

Scanned with CamScanner

Eg: $T(n) = T(n-1) + n$ with initial condition $T(0) = 0$

Sol: Let $T(n) = T(n-1) + n \rightarrow ①$

$T(0) = 0 \rightarrow ②$

if $n=1$ then $T(n) = T(1-1) + 1$
$= T(0) + 1$
$= 0 + 1$
$= 1$

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

if $n=3$ then $T(n) = T(3-1) + 3 = T(2) + 3 = 3 + 3 = 6$

By observing above generated equations we can derive a formula $T(n) = \frac{n(n+1)}{2}$

We can also denote $T(n)$ in terms of Bigoh notation as
$T(n) = O(n^2)$

b) Backward substitution:
In this method backward values are substituted recursively to derive formula

Ex: $T(n) = T(n-1) + n$ with initial condition $T(0) = 0$

Sol: Let $T(n) = T(n-1) + n \rightarrow ①$

Let $n = n-1$

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

Substitute eq ② in eq ①

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

Let $n = n-2$ in eq ①

$T(n-2) = T(n-2-1) + (n-2) \rightarrow ④$

Substitute ④ in ③ we get

$T(n) = T(n-3) + n-2 + n-1 + n \rightarrow ⑤$

$T(n) = T(n-k) + (n-k+1) + (n-k+2) + \ldots + n$

Scanned with CamScanner

Substitute ...

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

Let $n = n-2$ in eq (1)

$T(n-2) = T(n-2-1) + (n-2) \rightarrow (4)$

Substitute (4) in (3) we get

$T(n) = T(n-3) + n-2 + n-1 + n \rightarrow (5)$

$\vdots$

$T(n) = T(n-k) + (n-k+1) + (n-k+2) + \ldots + n$

Scanned with CamScanner

if $k = n$ then

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

$= T(0) + 1 + 2 + \ldots + n$

$= 0 + 1 + 2 + \ldots + n$

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

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

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

Eg 2) $T(n) = T(n-1) + 1$ with initial condition $T(0) = 0$

a) forward substitution:

Let $T(n) = T(n-1) + 1 \rightarrow (1)$

$T(0) = 0$

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

$= T(0) + 1$

$= 0 + 1$

$= 1$

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

$= T(1) + 1$

$= 1 + 1$

$= 2$

if $n = 3$ then $T(n) = T(3-1) + 1$

$= T(2) + 1$

$= 2 + 1$

$= 3$

b) backward substitution

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

Let $n = n-1$

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

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

Substitute eq (2) in eq (1)

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

$= T(n-2) + 2 \rightarrow (3)$

Let $n = n-2$ in eq (1)

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

$= T(n-3) + 1 \rightarrow (4)$

Substitute eq (4) in eq (3) we get

$T(n) = T(n-3) + 1 + 2 = T(n-3) + 4$

$T(n) = T(n-3)$ ...

Scanned with CamScanner

Scanned with CamScanner

Master's Method:-

In this method the basic recurrance relation equation is

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

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 bounded as follows :

Case i: if $F(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$ $\quad f(n) < n^{\log_b a}$

Case ii: if $F(n) = \Theta(n^{\log_b a})$ then $T(n) = \Theta(n^{\log_b a} \cdot \log n)$ $\quad f(n) = n^{\log_b a}$

Case iii: if $F(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$ and if $a \cdot F(n/b) \leq c\, F(n) \{c < 1\}$ $\quad f(n) > n^{\log_b a}$

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

1) Eg: $T(n) = 9T(n/3) + n$

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

Here $a = 9, b = 3, f(n) = n$

$n^{\log_b a} = n^{\log_3 9} = n^{\log_3 3^2} = n^{2 \log_3 3} = n^{2 \times 1} = n^2$

Scanned with CamScanner

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

$f(n) = n$

$f(n) < n^{\log_b^a}$ (case i is applied)

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

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

(2)

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

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

Here $a=2, b=2, f(n) = n^3$

$n^{\log_b^a} = n^{\log_2^2} = n^{\log 1} = n^1 = n$

$f(n) > n^{\log_b^a}$ (case iii is applied)

Then $f(n) = \Theta(F(n)) = \Theta(n^3)$

top down approach

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 searched from the list of elements stored 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 the searched element is in the list

1) if key = A[m] then the searched element
2) if key < A[m] then the search the left sublist
3) if key > A[m] then search the right sublist

Algorithm:

Name of the algorithm: Algorithm Binary Binsearch(A[0...n-1], key)

//problem description: This algorithm is for searching the element by using binary search method

Scanned with CamScanner

Scanned with CamScanner

//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;   location of the value
 Else if (key < A[m]) then
    high ← m-1;
     [ Binsearch (A [0 ... m-1], key) ];   optional
 else
   low ← m+1;
  [ Binsearch (A[m+1 ... n], key) ; ]   optional
 }
 return -1;
```

The basic operation in binary operation is comparison of search key with the array elements. To analyze efficiency of binary search we must count the number of times the key gets compared the array elements

→ In the algorithm after one comparison the list of 'n' elements are divided into $\frac{n}{2}$ sublists. The worst case efficiency is that the algorithm compares all the array elements for searching the desired element. In this, one comparison is made and based on this comparison array is divided each time into '$\frac{n}{2}$' sublists. Hence the worst case time complexity is given by

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

Scanned with CamScanner

Scanned with CamScanner

$C_{worst}(n/2)$ = Computing time required to compare left sublist or right sublist

1 = one comparison is made with middle element

But as we consider the rounded down value when array gets divided the above situation can be written as

$C_{worst}(1) = 1 \longrightarrow (2)$ (initial condition) middle value

Assume $n = 2^k$

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

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

using backward substitution method we can substitute

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

Then eq (3) becomes

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

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

Then $C_{worst}(2^k) = [C_{worst}(2^{k-3}) + 1] + 2$

$= C_{worst}(2^{k-3}) + 3$

|

|

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

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

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

$= 1 + k$

$\therefore C_{worst}(1) = 1$

Scanned with CamScanner

Scanned with CamScanner

$= C_{worst}(2^{k-3}) + 3$

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

Scanned with CamScanner

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

$$\log_2^n = \log_2^{2^k}$$
$$\log_2^n = k \log_2^2$$
$$\log_2^n = k.1$$
$$\therefore k = \log_2^n$$

~~the binary search~~

The binary search time complexity is $O(\log_2^n)$
average case

Advantages of Binary Search:
This method is an optimal searching algorithm which can search the desired element very efficiently

Disadvantages of Binary Search:
This algorithm requires the sorted list, then only this method is applicable

Applications:
1) This Method is an efficient method to search the desired records from the databases.
2) For solving non-linear equations with one unknown value

Merge sort: bottom up approach
The Merge sort is a sorting algorithm that uses divide and Conquer strategy. In this method, the division is done dynamically.

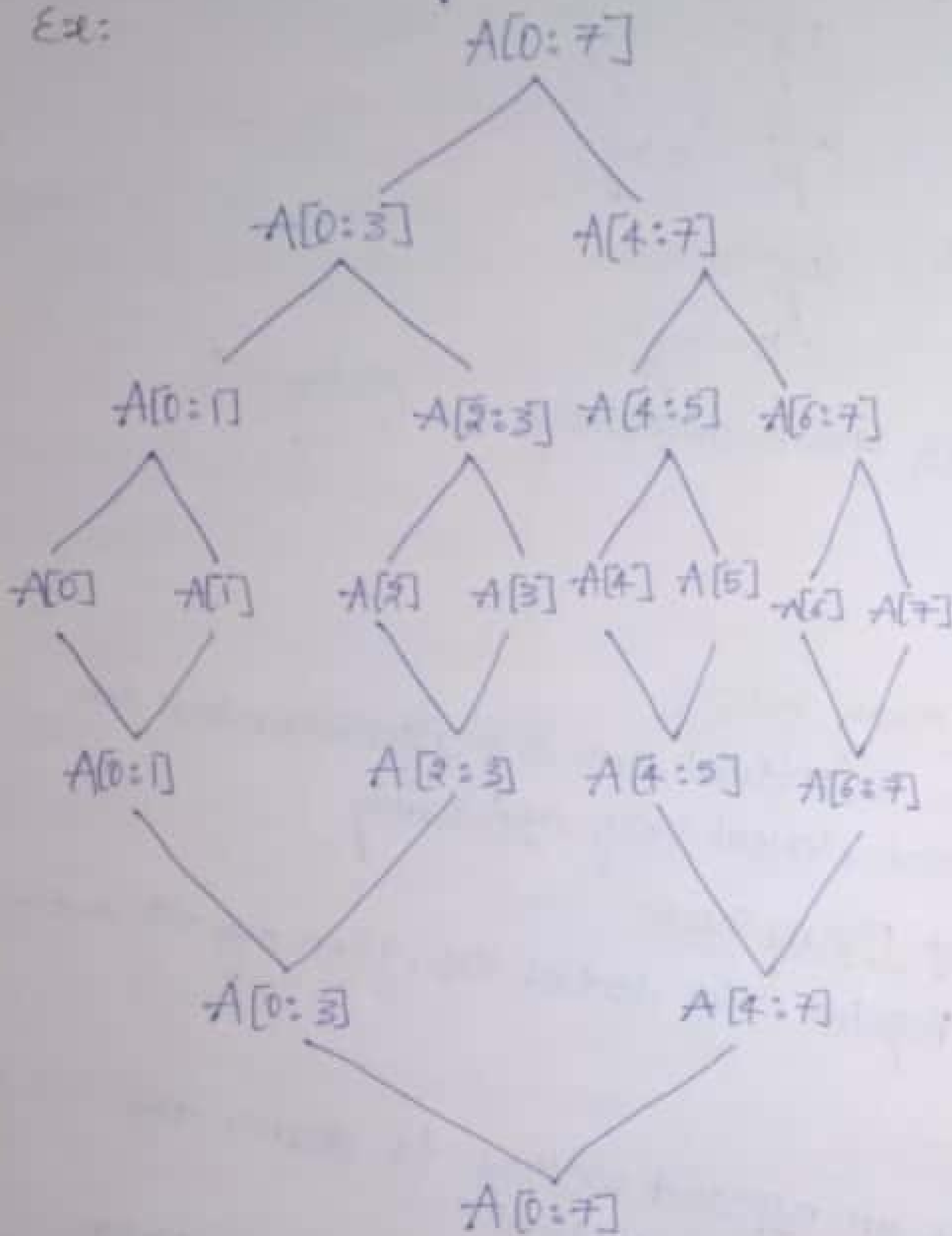
Merge sort consists of 3 steps:
1. Divide: partition the array into 2 sublists
2. Conquer: sort the each sublist

Scanned with CamScanner

Scanned with CamScanner

3. Combine: Merge solutions of sublist into a single list

Ex:



i) A = | E | X | A | M | P | L | E |

A[0:6]

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

a[1:10]: (310, 285, 179, 652, 351, 423, 861, 254, 450, 520)

27/107

Scanned with CamScanner

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

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

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

E X A M P L E

| E | X | | A | M | | E | L |

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

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

a) 70, 20, 30, 40, 10, 50, 60

A[0:6]

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

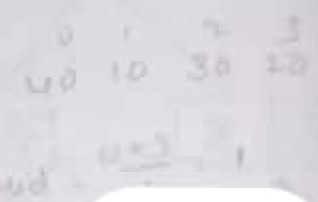
28/107

Scanned with CamScanner

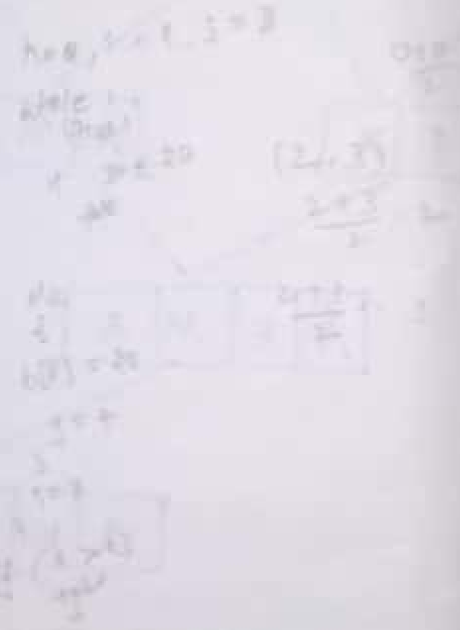
Scanned with CamScanner

Algorithm:

```
Algorithm mergesort (low, high)
{
if (low<high) then
  {
  mid = [(low+high)/2];
  mergesort (low, mid); mergesort(mid+1, high)
  merge (low, mid, high);
  }
}
```




```
Algorithm merge (low, mid, high)
{
h= low, i:= low, j:= mid+1;
while((h<=mid) and (j<= high))do
{
 if (a[h] <= a[j]) then
   {
   b[i] := a[h];
   h:= h+1;
   }
 else
  {
  b[i]:= a[j];
  j:= j+1;
  }
  i:=i+1;
}
if (h>mid) then
for k:=j to high do
  {
  b[i]:= a[k]
  i:= i+1;
  }
}
for k:= low to high do
   a[k]:= b[k];
}
```



```
else
{ for k:=h to mid do
  {  b[i]:= a[k];
     i:= i+1;
  }
}
```

Scanned with CamScanner

Analysis:-

In merge sort algorithm 2 recursive calls are made. Each recursive call focusses on 'n/2' elements of the list and 1 call is made to combine two sublists i.e to merge all 'n' elements. Hence the recurrance relation is

$C(n) = C(n/2) + C(n/2) + cn$ for $n>1$

$= 2C(n/2) + ~~C(n/2)~~ cn$

$C(1) = 0$ for $n=0$

By using Master's theorem

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

$a=2, b=2, f(n) = n$

$n^{\log_b^a} = n^{\log_2^2} = n^1 = n$

Here case (ii) condition is satisfied

if $f(n) = \theta(n^{\log_b^a})$

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

$n = n$ ✓

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

$= \theta(n^{\log_2^2} \log n)$

$= \theta(n \log n)$

Using Bigoh notation $\boxed{T(n) = O(n \log n)}$

Using substitution method, let the recurrance relation be

$C(n) = C(n/2) + C(n/2) + Cn$ for $n>1$ ⟶ (1)

$T(1) = 0$ for $n=0$ ⟶ (2)

$= 2C(n/2) + Cn$ ⟶ (3)

Apply Backward substitution method

$n = 2^k$

30/107

Scanned with CamScanner

Scanned with CamScanner

$$C(2^k) = 2C\left(\frac{2^k}{2}\right) + C2^k.$$

$$= C(2^{k-1}) + C2^k \longrightarrow (4)$$

Let $k = k-1$

$$C(2^{k-1}) = 2C(2^{k-1-1}) + C2^{k-1}$$

$$= 2C(2^{k-2}) + C2^{k-1} \longrightarrow (5)$$

Substitute eq (5) in eq (4)

$$C(2^k) = 2\left[2C(2^{k-2}) + C2^{k-1}\right] + C2^k$$

$$= 2^2 C(2^{k-2}) + 2C(2^{k-1}) + C2^k$$

$$= 2^2 C(2^{k-2}) + 2C\frac{2^k}{2} + C2^k$$

$$= 2^2 C(2^{k-2}) + C2^k + C2^k$$

$$= 2^2 C(2^{k-2}) + 2C2^k$$

We can write

$$C(2^k) = 2^3 C(2^{k-3}) + 3C2^k$$

$$= 2^4 C(2^{k-4}) + 4C2^k$$

$$\vdots \qquad \vdots$$

$$C(2^k) = 2^k C(2^{k-k}) + kC2^k$$

$$= 2^k C(2^0) + kC2^k$$

$$= 2^k C(1) + kC2^k$$

$$= 2^k \cdot 0 + kC2^k$$

$$= kC2^k$$

$$C(n) = \log n \cdot n$$

$$\therefore C(n) = n \log n$$

Scanned with CamScanner

$n = 2^k$

$\log_2^n = \log_2^{2^k}$

$\log_2^n = k \log_2^2$

$\log_2^n = k.1$

$\therefore \boxed{k = \log_2^n}$

Quick sort :-

This techuique was invented by Hoare and he is considered that this method to be a fast method to sort the elements. Here the division into 2 subarrays is made so that the sorted subarrays do not need to be merge later. This is accomplished by rearranging the elements in an array ~~A[A1 ---- An]~~ A [1-n]

In this method, the list is divided into 2 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 the pivot are greater than the pivot

$i < j$: swap i^{th} element & j^{th} element

$i > j$: swap pivot element & j^{th} element

Scanned with CamScanner

Eg:

pivot	2	3	4	5	6	7	8	9	10	i	j	
(65)	70 (i)	75	80	85	60	55	50	45 (j)	+∞	2	9	2<9
65	45	75 (i)	80	85	60	55	50 (j)	70	+∞	3	8	3<8
65	45	50	80 (i)	85	60	55 (j)	75	70	+∞	4	7	4<7
65	45	50	55	85 (i)	60 (j)	80	75	70	+∞	5	6	5<6
65	45	50	55	60 (j)	85 (i)	80	75	70	+∞	6	5	6>5

[60 45 50 55] [65] [85 80 75 70 +∞]

left array subset — right array subset

2) 65, 45, 50, ~~60~~, ~~45~~, 80, 48, 78, 63, 90

pivot	2	3	4	5	6	7	8	9	i	j	
(65)	45	50	80 (i)	48	78	63 (j)	90	+∞	4	7	4<7
65	45	50	63	48 (j)	78 (i)	80	90	+∞	6	5	6>5

48 45 50 63 [65] 78 80 90 +∞

pivot element

Algorithm Quick sort (left, right)

```
{
  (p = left, q = right + 1);
  if (p<q) then
  {
    j = partition (a, p, q);
    Quicksort (p, j-1);
    Quicksort (j+1, q);
  }
}
```

Scanned with CamScanner

```
Algorithm Partition(a, left, right);
{
 p=a[left], i:=left, j:=right;
 repeat
 {
  repeat
   i:=i+1;
  until (a[i]≥p);
  repeat
   j:=j-1;
  until (a[j]≤p);
  if (i<j) then
  Interchange (a,i,j);
 }until (i≥j)
  a[left]:= a[j]
   a[j]:=p;
   return j;
 }
Algorithm Interchange (a,i,j)
 {
 t:=a[i].
  a[i]:=a[j];
  a[j]:=t;
 }
```

35, 80, 75, 70, 45

p=a[1]
p=35, i=1, j=5
i=...=2
a[i]≥p
a[i]≥

34/107

```
repeat
   i:=i+1;
until (a[i]≥p);
repeat
   j:=j-1;
until (a[j]≤p);
```

Scanned with CamScanner

Scanned with CamScanner

1) 60, 45, 50, 55 (indices 1, 2, 3, 4)
↓ left ↓ right

$P = a[left] = a[1] = 60$

$i = left = 1$
$j = right+1 = 4+1$
$j = 5$

$i = i+1$
$i = 1+1$
$i = 2$

$a[i] \geq P$
$a[2] \geq 60$
$45 \geq 60$ ✗

$a[3] \geq P$
$50 \geq 60$ ✗

$a[4] \geq P$
$55 \geq 60$ ✗

$j = j-1$ → $a[j] \leq P$
$j = 5-1$
$j = 4$

$a[4] \leq 60$
$55 \leq 60$ ✓ → if $i < j$
$5 < 4$ ✗

$i \geq j$
$5 \geq 4$ ✓
$a[1] = a[4]$
$a[1] = 55$
$a[4] = 60$

Analysis:

If the array is always partitioned at the mid then it brings the best case efficiency of an algorithm. The recurrence relation of Quick sort for obtaining best case is

$$T(n) = T(n/2) + T(n/2) + n$$
$$T(1) = 0$$
$$T(n) = 2T(n/2) + n \longrightarrow (1)$$

Using Masters theorem

$$T(n) = 2T(n/2) + n \longrightarrow (1)$$
$$T(1) = 0 \longrightarrow (2)$$

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

$\Rightarrow a = 2, b = 2, f(n) = n$

$$n^{\log_b^a} = n^{\log_2^2} = n^1 = n$$

$$\therefore \boxed{n^{\log_b^a} = n}$$

$$f(n) = n^{\log_b^a} \Rightarrow \boxed{n = n}$$

Scanned with CamScanner

Case II is applied

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

$= \Theta(n \log n)$

By using Big oh Notation $T(n) = O(n \log n)$

Substitution method:-

We can obtain the best case time complexity of Quicksort using backward substitution method

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

$T(1) = 0$ for $n = 0 \longrightarrow (2)$

$T(n) = 2T(n/2) + Tn \longrightarrow (3)$

Apply backward substitution method

Assume $n = 2^k$

$T(2^k) = 2T\left(\frac{2^k}{2}\right) + T2^k$

$= T(2^{k-1}) + T2^k \longrightarrow (4)$

Let $k = k-1$

$T(2^{k-1}) = 2T(2^{k-1-1}) + Tc2^{k-1}$

$= 2T\left(\frac{k-2}{2}\right) + Tc^{2k-1} \longrightarrow (5)$

substitute eq (5) in eq (4)

$T(2^k) = 2[2T(2^{k-2}) + T2^{k-1}] + T2^k$

$= 2^2 T(2^{k-2}) + 2T(2^{k-1}) + T2^k$

$= 2^2 T(2^{k-2}) + 2T\frac{2^k}{2} + T2^k$

$= 2^2 T(2^{k-2}) + T2^k + T2^k$

$= 2^2 T(2^{k-2}) + 2T2^k$

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

We can write

$T(2^k) = 2^3 T(2^{k-3}) + 3T2^k$

$= 2^4 T(2^{k-3}) + 4T2^k$

⋮

Scanned with CamScanner

Scanned with CamScanner

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

$= 2^k T(2^0) + kT 2^k$

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

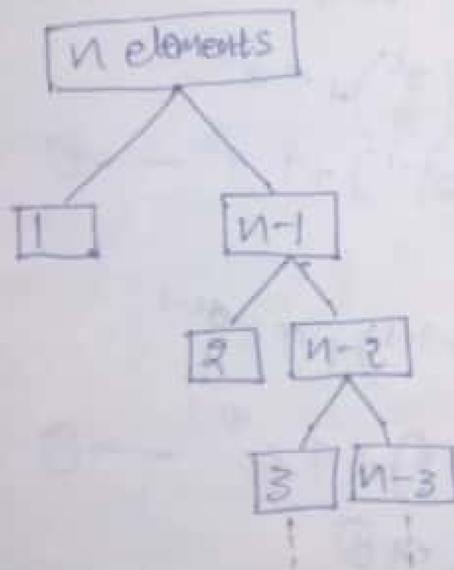
$= 0 + kT 2^k$

$= kT 2^k$

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

$\therefore \boxed{T(n) = \log n . n}$

The worst case time complexity for quicksort occurs when the pivot element is minimum or maximum element of all the elements in the list. This can be graphically represented as



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

We know, that $1 + 2 + 3 + \dots + (n-1) + n = \frac{n(n+1)}{2}$

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

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

$T(n) = n^2$

By using Bigoh notation $T(n) = O(n^2)$

Scanned with CamScanner

~~Selection sort~~

Applications of Divide and Conquer:
Binary search, Merge sort, Quick sort, selection sort, Stressen's matrix multiplication, finding Maximum and minimum elements

Selection sort:

In this problem, we have 'n' elements A[1-n] and are required to determine the kth smallest element if the partitioning element 'v' is positioned at A[j] then 'j-1' elements are less than or equal to A[j] and 'j+1' elements are greater than the A[j]. If k<j then the kth smallest element is in A[1:j-1]. If k=j then A[j] is the kth smallest element if k>j then k^{th} smallest element in A[j+1:n]. The select function places the kth smallest element into positions A[k] and partitions the remaining elements so that A[i] ≤ A[k], 1 ≤ i < k and A[i] ≥ A[k], k < i ≤ n

Algorithm:

```
Algorithm Select1(a,n,k)
{
  low:= 1, up:=n+1;          low=1 up=6+1
    a[n+1]:= ∞                up=7
  repeat
  {
    j:= partition (a,low,up);
     if (k=j) then
     return;                  1<18
  else if (k<j) then          up:=18
      up:= j
  else
```

```
  low:=j+1;
  } until (false);
}
                  low 18 up
Algorithm partition (a,left,right);
     { P=a[low]; i:=low, j:= high;
repeat    P=13,   i=1, j=7
{
 repeat
   i:=i+1;      i=2
```

Scanned with CamScanner

```
  low := j+1;
 } until (false);
}
Algorithm partition (a, left, right);
  { P = a [low] ; i := low , j := high ;
repeat
 {
  repeat
     i := i+1;
 until (a[i] >= P);
 repeat
     j := j-1;
 until (a [j] <= P);
 if (i<j) then
Interchange (a, i, j);
 } until (i >= j)
   a[left] := a [j];
   a [j] := P;
   return j;
}
Algorithm Interchange (a, i, j)
   {
  t := a[i];
  a[i] := a [j];
  a [j] := t;
  }
repeat
    i := i+1;
until (a[i] >= P);
 repeat
    j := j-1;
until (a[j] <= P);
```

39/107

① 18, 7, 15, 5, 3, 1

low := 1, up := 6+1

up := 7

a[7] = ∞

Analysis:

The worst case time complexity of selection sort is $O(n^2)$ when the input $a[1:n]$ is such that the partitioning element on the ith call to partition is the ith smallest element and $k = n$. In this case 'm' increases by 1 following each call to partition and 'j' remains unchanged.

Therefore the recurrance relation is

$$n + (n-1) + (n-2) + \cdots + 3 + 2 + 1 = \frac{n(n-1)}{2}$$

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

Average time complexity:

~~The average time complexity~~

On the first call to partition, the partition element 'v' is the ith smallest element with probability $1/n$, $1 \le i \le n$.

The time required by partition and 'if' statement is $O(n)$. Hence there is a constant 'c', $c > 0$ such that the ~~rec~~

$$T(n) \le c_n + \frac{1}{n} \max \left[\sum_{1 \le i < k} T^{k-1}(n-i) + \sum_{k < i \le n} T^k(i-1) \right], \quad n \ge 2$$

So $R(n) \le c_n + \frac{1}{n} \max \left\{ \sum_{1 \le i < k} R(n-i) + \sum_{k < i \le n} R(i-1) \right\}$

$$R(n) \le c_n + \frac{1}{n} \max \left\{ \sum_{1 \le i \le k} R(i) + \sum_{k < i \le n} R(i) \right\}, \quad n \ge 2 \rightarrow ①$$

Scanned with CamScanner

Scanned with CamScanner

The time required by [illegible] is a constant c, [illegible] is $O(n)$. Hence there is a constant [illegible] that the rec

$T(n) \leq c_n + \frac{1}{n} \max \left[\sum_{1 \leq i < k} T^{k-1}(n-1) + \sum_{k < i \leq n} T^k(i-1) \right], n \geq 2$

So $R(n) \leq c_n + \frac{1}{n} \max \left\{ \sum_{1 \leq i \leq k} R(n-1) + \sum_{k < i \leq n} R(i-1) \right\}$

$R(n) \leq c_n + \frac{1}{n} \max \left\{ \sum_{1 \leq i \leq k} R(i) + \sum_{k < i \leq n} R(i) \right\}, n \geq 2 \rightarrow (1)$

Scanned with CamScanner

$R(1) \leq c$

$R(n) \leq 4c_n$ by Induction on n

for $n = 2, i = 1$

$R(n) \leq 2c + \frac{1}{2} \max R(1), R(1)$

$\leq ~~2.5c~~$ $2.5c$

$\therefore R(n) \leq 2c \leq 4c_n$

Assume $R(n) \leq 4c_n$ for all n,

$2 \leq n < m$

Similarly for $n = m$

$R(m) \leq c_m + \frac{1}{m} \max \left\{ \sum_{m-k+1}^{m-1} R(i) + \sum_{k}^{m-1} R(i) \right\}$

Since we know that $R(n)$ is a non-decreasing function of 'n', it follows that $\sum_{m-k+1}^{m-1} R(i) + \sum_{k}^{m-1} R(i)$ is maximised if $k = \frac{m}{2}$ when 'm' is even and $k = \frac{m+1}{2}$ when 'm' is odd

If m is even $R(m) \leq c_m + \frac{2}{m} \sum_{m/2}^{m-1} R(i)$

$\leq c_m + \frac{8c}{m} \sum_{m/2}^{m-1} R(i)$

$< 4c_m$

If m is odd

$R(m) \leq c_m + \frac{2}{m} \sum_{\left(\frac{m+1}{2}\right)}^{m-1} R(i)$

$\leq c_m + \frac{8c}{m} \sum_{\left(\frac{m+1}{2}\right)}^{m-1} R(i)$

$\leq 4c_m$

Scanned with CamScanner

Since $T(n) \leq R(n)$

It follows that $T(n) \leq 4^{cn}$

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

Stressen's Matrix Multiplication:-

To multiply '2' matrices A and B each of size 'n'

$C = A * B$

suppose the matrix size is 2X2 then

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

$C_{11} = A_{11} * B_{11} + A_{12} * B_{21}$

$C_{12} = A_{11} * B_{12} + A_{12} * B_{22}$

$C_{21} = A_{21} * B_{11} + A_{22} * B_{21}$

$C_{22} = A_{21} * B_{12} + A_{22} * B_{22}$

In this procedure, to accomplish 2X2 matrix multiplication we require 8 multiplications and 4 additions. We can write the following algorithm

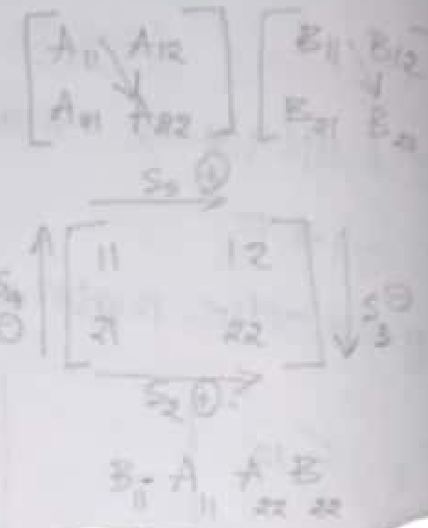
```
Algorithm mul(A, B, C, n)
{
for i:=1 to n do
for j:=1 to n do
   C[i,j]:=0;
 for k:= 1 to n do
  C[i,j] = C[i,j] + A[i,k] * B[j,k]
  }
```

42/107

Scanned with CamScanner

Scanned with CamScanner

The time complexity of the above algorithm is $O(n^3)$
Stressen showed that 2x2 Matrix multiplication accomplished by using 7 multiplications and 18 subractions (or) additions



$S_1 = (A_{11} + A_{22}) * (B_{11} + B_{22})$

$S_2 = (A_{21} + A_{22}) * B_{11}$

$S_3 = A_{11} * (B_{12} - B_{22})$

$S_4 = A_{22} * (B_{21} - B_{11})$

$S_5 = (A_{11} + A_{12}) * B_{22}$

$S_6 = (A_{21} - A_{11}) * (B_{11} + B_{12})$

$S_7 = (A_{12} - A_{22}) * (B_{21} + B_{22})$

$C_{11} = S_1 + S_4 - S_5 + S_7$

$C_{12} = S_3 + S_5$

$C_{21} = S_2 + S_4$

$C_{22} = S_1 + S_3 - S_2 + S_6$

The divide & Conquer approach can be used for implementing

i) divide:
Divide the matrices into submatrices
ii) Conquer
Use a group of matrix multiplication equations
iii) Combine
Recursively multiply submatrices and get the final result of multiplication

Algorithm:

Algorithm stmul(A, B, C, n)
{
if (n==1) then

Scanned with CamScanner

```
{
C = C + (A*B)
}
else
{
stmul (A, B, C, n/4);
stmul (A, B+n/4, C+n/4);
stmul (A+ 2*n/4, B, C+ 2*n/4, n/4);
stmul (A+2*n/4, B+n/4, C+3* n/4, n/4);
stmul (A+ n/4, B+ 2*n/4, C, n/4);
stmul (A+ n/4, B+ 3*(n/4), C+ n/4, n/4);
stmul (A+3*(n/4), B+ 2*(n/4), C+ 2*(n/4), n/4);
stmul (A+ 3*(n/4), B+ 3*(n/4), C+3*(n/4), n/4);
  }
}
```


Analysis:-

$T(n) = 7T(n/2)$

$T(1) = 1$

Assume $n = 2^k$

$T(2^k) = 7T\left(\frac{2^k}{2}\right)$

$= 7T(2^{k-1}) \longrightarrow (1)$

Let $k = k-1$ in (1)

$T(2^{k-1}) = 7.T(2^{k-1-1})$

$= 7.T(2^{k-2}) \longrightarrow (2)$

Substitute eq(2) in eq(1)

$T(2^k) = 7[7T(2^{k-2})]$

$= 7^2 T(2^{k-2})$

$k-2$

Scanned with CamScanner

Scanned with CamScanner

$T(2^k) = 7^3\, T(2^{k-3})$

$\vdots$

$T(2^k) = 7^k\, T(2^{k-k})$

$= 7^k \cdot T(2^0)$

$= 7^k\, T(1)$

$= 7^k (1)$

$= 7^k$

$= \log_2^n$

$= n^{\log_2^7}$

$= n^{2.81}$

$T(n) = n^{2.81}$

$T(n) = O(n^{2.81})$

45/107

~~Substitution~~ Masters Method:

$T(n) = 7\, T(n/2)$

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

$a = 7,\ b = 2,\ F(n) = 0$

$n^{\log_b^a} = n^{\log_2^7} = n^{2.81}$

$F(n) = 0$

$n^{\log_b^a} = n^{2.81}$

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

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

$= \Theta(n^{2.81})$

By using Big oh notation $T(n) = O(n^{2.81})$

Scanned with CamScanner

Scanned with CamScanner

Finding Maximum and Minimum:-

Here the problem is to find the maximum and minimum items in a set of 'n' elements. This can be solved by using the divide and Conquer Method (or) straight forward algorithm

Straight Forward Algorithm:

```
Algorithm straightMinMax(a,n,max,min)
{
  max = min = a[1];
for i:=2 to n do
{
 if (a[i] > max) then
    max:= a[i];
 if (a[i] < min) then
    min:= a[i];
  }
}
```

Modified Straight Forward Algorithm:

```
Algorithm straightminmax(a,n,max,min)
 { for i=2 to n do {
  max:= min= a[1];
  if (a[i] > max ) then
     max:= a[i];
  else if (a[i] < min) then
     min:= a[i];     }}
```

Here the no. of the comparisons are 2(n-1) and the time complexity is O(2(n-1)) so the above algorithm is modified Now the comparisons required for modification method is (n-1) and the time complexity is O(n-1)

When the elements are in increasing order then the best case time complexity and the no of comparisons required are n-1

Scanned with CamScanner

Scanned with CamScanner

When the elements are in decreasing order then we get the worst case time complexity and no. of comparisons required are $2(n-1)$.

Then the algorithm is modified, then we require 'n-1' element comparisons and the time complexity is $O(n-1)$ in best, average and worst cases.

Divide and Conquer method:

The number of comparisons for finding maximum minimum elements can be reduced by using divide and conquer strategy. Here the list is divided into 2 sublists A_1, A_2 where $A_1 = (A[1] ----- A[n/2])$
$A_2 = (A[n/2+1] ---- A[n])$

The two sublists are solved separately by calling divide and conquer method recursively. Finally $Max[A] = (Max(A_1), max(A_2))$
$Min(A) = min(min(A_1), min(A_2))$

Algorithm:

```
Algorithm MaxMin (first, last, Max, Min)
{
if (first = = last)
{
Max:= A[first];
Min: = A[first];
}
else if (first+1 = = last)
{
if (A [first] < A[last]) then
{
Max:= A[last];
Min: = A[first];
```

Scanned with CamScanner

Scanned with CamScanner

```
}
else
{
Max: = A[first];
Min: = A[last];
}
}
else
{
mid = (first + last) / 2 ;
MaxMin (first, mid, Max, Min);  -> left sublist
MaxMin (mid+1, last, tMax, tMin);  -> right sublist
if (Max < tMax)  -> to combine the elements
{
Max: = tMax;
}
if (Min > tMin)
{
Min: = tMin;
}
}
}
```

① 53, 43, -3, -8, 48, 92, 68, 25, 75

mid = (1+9)/2 = 5

53, 43, -3, -8, 48

mid = (1+5)/2 = 3

92, 68, 25, 75

mid = (1+4)/2 = 2.5 = 3

53, 43, -3

mid = (1+3)/2 = 2

-8, 48

92, 68, 25

75

53 43 -3 -8 48

Max = 53 Min = 43

Min = -8
Max = 48

92 68

Max = 92

Max = 92, Min = 25

Scanned with CamScanner

Scanned with CamScanner

Analysis: write the method if asked even ...

$$T(n) = \begin{cases} T(n/2) + T(n/2) + 2 & \text{if } n>2 \\ 1 & \text{if } n=2 \\ 0 & \text{if } n=1 \end{cases}$$

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

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

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

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

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

$\vdots$

$= 2^{k-1} T\left(\frac{n}{2^{k-1}}\right) + 2^{k-1} + 2^{k-2} + \cdots + 2^2 + 2$

$= 2^{k-1} T\left(\frac{n}{2^{k-1}}\right) + \sum_{i=1}^{k-1} 2^i$

Put $n = 2^k$ we get

$= 2^{k-1} T\left(\frac{2^k}{2^{k-1}}\right) + \sum_{i=1}^{k-1} 2^i$

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

$= 2^{k-1}(1) + 2^k - 2 \qquad [\because T(2) = 1]$

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

$T(n) = \frac{3n}{2} - 2$

$\therefore \boxed{T(n) = \frac{3n}{2} - 2}$

$\frac{a(r^{n}-1)}{r-1}$ ... $\frac{2(2^{k-2}-1)}{2-1}$

optimal - best or most favourable optimum.

optimum - most conductive to a favourable outcome, best

feasible - Possible to do easily or conveniently (practical)