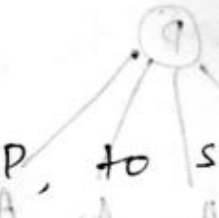
Design & Analysis of Algorithms

* Design of Algorithm
* Analysis of Algorithm



If we have a problem P, to solve it, we write programs.

P. ⟶ programs
↳ step by step procedure
↳ Algorithm.

Good s/w engineering practice

Before writing programs, we write the sequence of operations.

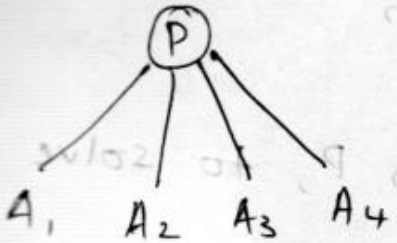
Eg: Add two numbers

Algorithm:

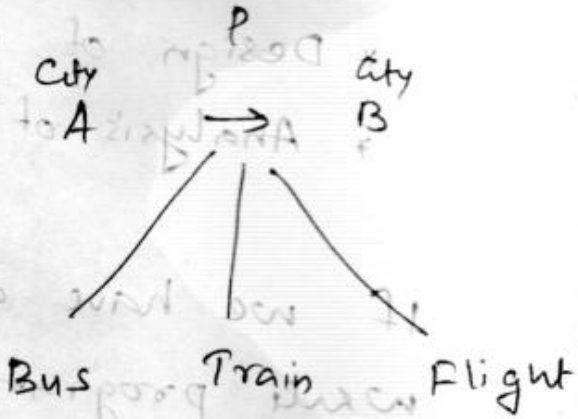
1. Read first number a
2. Read second number b
3. Calculate sum = a+b
4. Print sum

P. ⟶ Design Algorithm ⟶ Program

Consider one problem, there are multiple algorithms to solve the problem



Eg: City A → City B



A problem has more than one strategies available. Then here comes the importance of analysis of algorithms.

Analysis

Select best from different possible algorithms.

Then we implement that best one as program.

To choose a best one, we have to analyse and compare the algorithms.

There are two criteria

* Time
* Space (Memory)

Evaluate each algo. that takes how much time and space to solve a problem.

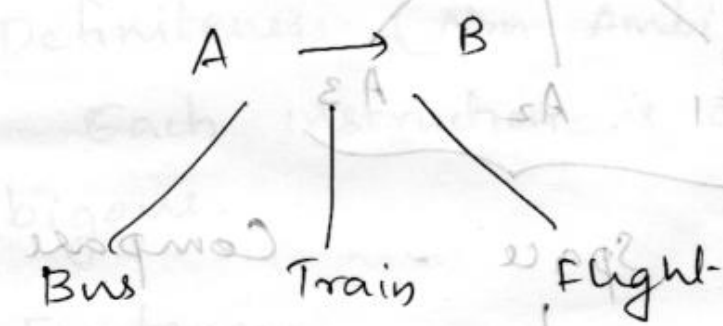
Based on that, compare the algorithms. We select a best one and implement as program.

Generally best means, it takes less time and less space.

But in most cases,

If we want to develop an algorithm, which takes less time, then we have to compromise with space. Also if we want to develop an algo. which takes less space, then we compromise with time.

Eg:



Criteria.
Time -
cash -

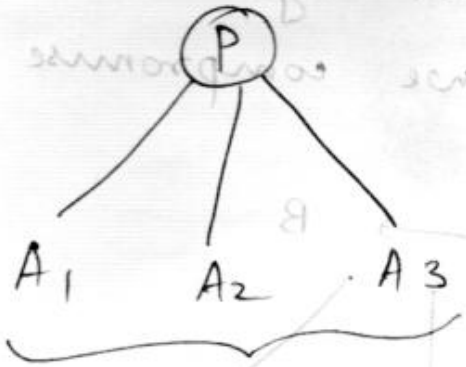
Best is less time & less cash (It is not possible)

Less time means we have to compromise with cash

Less cash means we have to compromise with time

Consider an algo. A1, it takes more time for larger inputs but less time for smaller inputs. It is not generally best.

Suppose we have a s/w: Always we are giving smaller inputs to the s/w. According to this s/w. A1 algo. is best.
That means it depends on application



Time, Space - Compare.

↓ best algo.

Program (Implement)

Algorithm

An algorithm is a finite set of instruction that if followed accomplishes a particular task.

Criteria for algorithms

1. Input (which type required)

   One or more quantities are externally supplied

2. Output (Specified)

   Atleast one quantity of o/p is produced.

3. Definiteness (Non Ambiguity)

   Each instruction is clear and unambigous.

4. Finiteness

   After a finite number of steps, algo. should terminate.

5. Effectiveness

   Every instruction must be basic so that it can be carried out in principle by person by using only a pencil and a paper.

Properties of Algorithm

1. Input
2. Output
3. Finiteness
4. Definiteness
5. Effectiveness
6. Generality
7. Correctness

The analysis of an algorithm focuses on time and space complexity.

Time Complexity

is the amount of CPU time it needs to run to completion.

Space Complexity

is the amount of memory it needs to run to completion.

Space Complexity

1 word = 4 byte.

Memory space S(P) needed by a program P, consists of two components

* Fixed part: (independent/const.)
  needed for instruction space (byte code), simple variable space, constants space etc.
* Variable part: (dependent char.)
  dependent on a particular instance of input and output data → S_p (instance)
  (reference var., stack space, array size)

$$S(P) = C + S_p(\text{instance})$$

Eg: Algorithm abc (a, b, c)

{

return a+b+b*c+(a+b-c)/(a+b)+4.0;

}

Fixed part

3 var. a, b, c (does not depend on other)

$S(P) = 3 + 0 = 3$

If array occurs it will vary

O(1)

a, b, c - 1+1+1 = 3

1 word = 4 byte

Eg: Algorithm sum (a[], n)

$S(P) = C + S_P$

{

s = 0

arr - n x 4 byte

for i = 1 to n do

s = s + a[i];

return s; → var. part

}

Variable part

Var a is array, size of array vary

↓

no: of elements in array

(each ele. will take space)

a[]

s, i, n

Fixed part

s, n, i } - 1 word each.

3 word

Space needed to store n = 1 word.

Space needed to store a[] = n words.

Space needed to store i and s = 2 words.

$S(P) = (3+n) \Rightarrow O(n)$

Eg:

```
Algorithm Rsum(a, n)
{
if (n<=0) then
return 0;
else
return Rsum(a, n-1) + a[n]
}
```

```
Algo.
int fact = 1;
for (int i = 1; i
i++)
{
fact = fact * i;
return fact
}
```

i, fact, n

O(3)

O(1)

Recursion

If an array have n elements, compute the sum of the elements in the array. Stack is used in recursion. A recursion have 3 variables

formal parameter, local variables,

return address.

If we call the fn. first time, we have to store the formal parameter, local var., and return val.

If we call next time, then also store formal, local & return addr.

↓ 1 word ↓ 1 word ↓ 1 word.

3 * no: of times recursion executed.

3 * n+1

3(n+1)

If n = 2

we have to call 3 times

$S(P) = 3(n+1) \longrightarrow O(n)$

Eg: Algo. fact (n)

```
{
if (n==0) then
return 1
else
return n * fact(n-1);
}
```

$$S(P) = \underline{3(n+1)} \rightarrow \underline{O(n)}$$

Time Complexity

Time taken by a program to complete its task.

Consists of two parts

* compile time (fixed) independent.
* run time (variable) depends on particular problem instance (t_p)

$$T(P) = c + t_p(\text{instance})$$

compile time does not depend upon instance characteristics (c). Once compiled pgm will run several times without recompilation.

Runtime depends on particular problem instance (t_p)

Two methods

1. Operation count
2. step count.

Operation count

select one or more operations such as addition, multiplication and comparison to determine how many of each is done.

Step count

No: of steps each statement in the program executes.

How to analyze an algorithm:

1. Time
2. Space
3. N/W
4. Power
5. CPU registers.

Eg: Algorithm swap (a, b)

```
{
    temp = a;   ——— 1
    a = b       ——— 1
    b = temp;   ——— 1
}
```

Every statement in the algo. takes one unit of time.

$f(n) = 3$ constant time, so.

$\Rightarrow O(1)$.

comments - Zero step
Assg. stt - one step
conditional - one step
while (expr) -
Loop (condi) - n+1
Body - n

Eg: Algorithm sum (a, n)

```
{
    s = 0                ——— 1
    for i = 1 to n       ——— n+1
    s = s + a[i]         ——— n
    return s             ——— 1
}
```

$f(n) = 1 + n+1 + n + 1 = 2n+3 \Longrightarrow O(n)$

Eg: Algorithm sum (a[ ], n, m)

```
{
for i = 1 to n do ——— n+1
    for j = 1 to m do — n(m+1) ⇒ mn+n
        s = s + a[i][j]; —— nm
    return s; ——— 1
}
```

$f(n) = n+1+mn+n+mn+1$

$= 2mn+2n+2$ $\Longrightarrow$ $O(mn)$

Eg: To add two arrays

```
Algorithm sum (a, n)
{
for i = 1 to m do ——— m+1
  {
    for j = 1 to n do — m(n+1) ⇒ mn+m
    {
    c[i][j] = a[i][j] + b[i][j]; — mn
    }
  }
}
```

$f(n) = m+1+mn+m+mn$

$= 2mn+2m+1$ $\Longrightarrow$ $O(mn)$

Eg: Sum of two matrices

```
Algorithm Add (A, B, n)
{
for (i=0; i<n; i++)  ---- n+1
{
  for (j=0; j<n; j++)  ---- n(n+1) ⇒ n²+n
  {
    c[i,j] = A[i,j] + B[i,j]  ---- n*n ⇒ n²
  }
}.
```

Here the highest degree is 2 so n^2.

$$f(n) = n+1+n^2+n+n^2$$
$$= 2n^2+2n+1 \Longrightarrow O(n^2)$$

Eg: Multiplication of two matrices.

```
Algorithm Mult (A, B, n)
{
for (i=0; i<n; i++)  ---- n+1
{
  for (j=0; j<n; j++)  ---- n(n+1) ⇒ n²+n
  {
    c[i,j] = 0;  ---- n*n ⇒ n²
    for (k=0; k<n; k++)  ---- n*n(n+1) ⇒ n³+n²
    {
      c[i,j] = A[i,k] * B[k,j]  ---- n*n*n ⇒ n³
    }
  }
}
```

$$f(n) = n+1+n^2+n+n^2+n^3+n^2+n^3$$
$$\Longrightarrow O(n^3)$$

n=10 ⇒ loop executes 5 times

Eg:
```
for (i=0; i<n; i=i+2)
{
  stmt;
}
```
stmt; ——— n/2

degree of polynomial is n so O(n)

O(n)

10 10>0 9

Eg:
```
for (i=n; i>0; i--)
{
  stt;
}
```
stt; ——— n

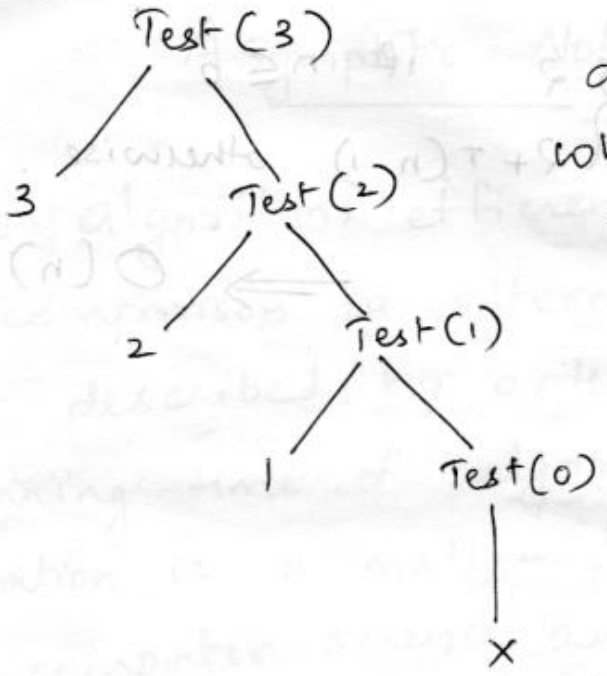
O(n)

Assume time taken by algo. ——— T(n) Tot. amt of time

Eg:-
```
void Test (int n)
{
  if (n > 0) — const. time
  {
    printf ("%d", n) ——— 1
    Test (n-1) ——— T(n-1)
  }
}
```
count = count + 1

Time taken for this algo. (n-1) i/p

If the algo. takes T(n) amt. of time, then Test (n-1) takes T(n-1) time. Condition takes const. amt. of time.



Printing a value and calling a function. what is the time taken for printf : just one unit of time. In each call it is taking 1 unit of call. Here 3 No: of calls is 4.

If I pass n, then total n+1 calls/times. printf executed n times.

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

$$T(n) = \begin{cases} 1 & n = 0 \\ T(n-1) + 1 & n > 0 \end{cases} \Longrightarrow O(n)$$

Recursive sum of n numbers. ele. in an array using recursion.

Eg:
```
Algorithm Rsum(a, n)
{
  if (n ≤ 0) then          // true. if this is true count = count+1
    return 0;              // count = count+1
  else                     // count = count+1
    return Rsum(a, n-1) + a[n];
}
```

if this is a recursion calling itself

$$T(n) = \begin{cases} 2 & \text{if } n \le 0 \\ 2 + T(n-1) & \text{otherwise.} \end{cases}$$

$$T(n) = \begin{cases} 2 & \text{if } n \leq 0 \\ 2 + T(n-1) & \text{otherwise.} \end{cases}$$

$\Longrightarrow O(n)$

Asymptotic Notation

The algorithm efficiency and performance in comparison to alternate algorithm is best described by order of growth of running time of algorithm. Asymptotic notation is a mathematical notation used in computer science and mathematics to describe the limiting behavior of function as their input sizes approach infinity or some other limit. It provides a way to analyze the efficiency and performance of algorithms by characterizing how the algorithm's resources usage (typically time or space) scales with the size of the input.

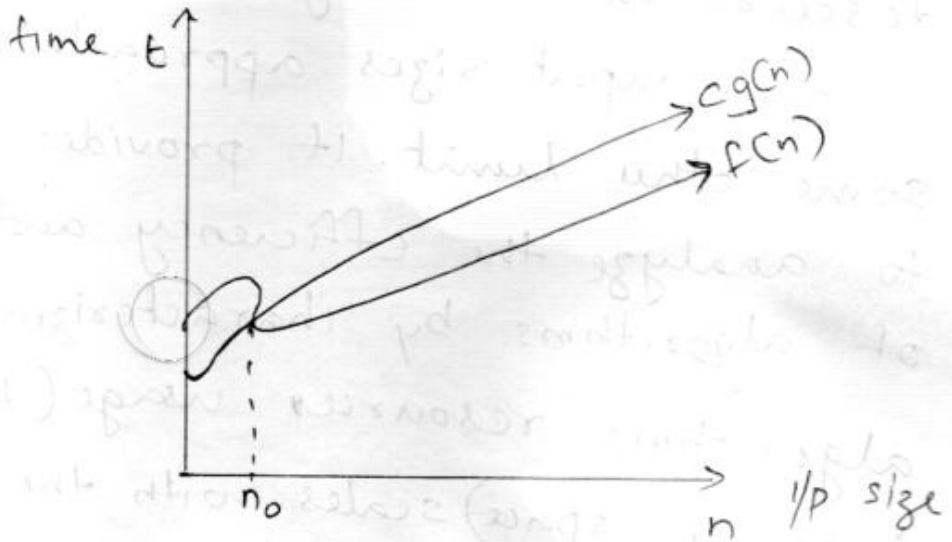
There are three common asymptotic notations.

1. Big O Notation (O)
2. Omega Notation (Ω)
3. Theta Notation (Θ)
4. Little oh (o)
5. Little omega (ω)

Big O Notation (O)

represented as $O(f(n))$, provides an upper bound on the growth rate of an algorithm's resource usage in the worst case scenario. It describes the maximum rate of growth of the function as the input.

$g(n)$ is an asymptotic upper bound for $f(n)$



Functions in order of increasing growth rate are

$$O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3) < O(k^n)$$

where k is a constant.

Given functions $f(n)$ and $g(n)$, we say that $f(n)$ is $O(g(n))$ if there are positive constants c and n_0 such that

$$f(n) \leq c \cdot g(n) \quad \text{for all } n \geq n_0$$

$$c > 0$$
$$n_0 \geq 1$$

If the above function is satisfied, then we can write

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

$g(n)$ is asymptotically greater than $f(n)$.

Eg: Show that $2n+7$ is $O(n)$

$f(n) = 2n+7$

$g(n) = n$

$f(n) \leq c \cdot g(n)$

$c = 3 \quad n_0 = 7$

$2n+7 \leq c \cdot g(n)$

$2 \times 7 + 7 \leq 3 \cdot 7$

$21 \leq 21$ ✓

So $2n+7$ is $O(n)$

$f(n) \leq c \cdot g(n) \quad n=2$

$2n+7 \leq 3 \cdot n$

$4+7 \leq 3 \cdot 2$ ✗

If $n=3 \quad c=3$

$2 \times 3 + 7 \leq 3 \cdot 3$

$13 \leq 9$ ✗

If $n=7 \quad c=3$

$14+7 \leq 3 \cdot 7$

$21 \leq 21$ ✓

If $n=8$

$24 \leq 24$ ✓

Eg: Show that $3n+2$ is $O(n)$.

$g(n) = n$.

$$f(n) \leq c \cdot g(n)$$

$$3n+2 \leq c \cdot n$$

If $n=3$ $c=4$

$$3 \times 3 + 2 \leq 4 \times 3$$

$$11 \leq 12 \quad \checkmark$$

If $n=2$ $c=4$

$$8 \leq 8 \quad \checkmark$$

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

$\therefore 3n+2$ is $O(n)$.

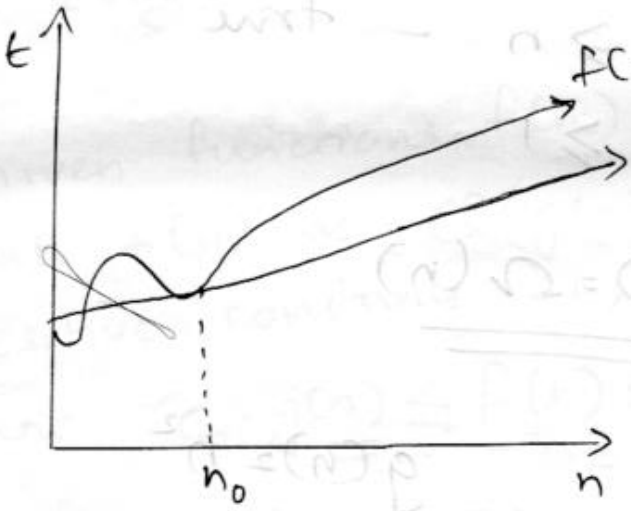
Eg:

Big Omega (Ω)

It represents lower bounds and best case. (tightest lower bound)

Running time of an algorithm with respect to its input size n. ($f(n)$)

We want to find a function which is lower bound to fn. $f(n)$.



$f(n) \geq c \cdot g(n)$

we want to find a fn. $g(n)$ such that $c \cdot g(n) \leq f(n)$ after a particular value n_0.

$$f(n) \geq c \cdot g(n), \quad n \geq n_0$$
$$c > 0$$
$$n_0 \geq 1$$

If these conditions satisfy then

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

- $g(n)$ is asymtotically smaller than $f(n)$

Eg: $f(n) = 3n+2$ $\quad g(n) = n$

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

$$f(n) \geq c \cdot g(n)$$

$$3n+2 \geq c \cdot n$$

if $c=1$ $\quad n \geq 1$

$3n+2 \geq n$ – true

$6 \geq 1$

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

Eg: $f(n) = 3n+2$ $\quad g(n) = n^2$

$$f(n) \geq c \cdot g(n)$$

linear in n $\leftarrow 3n+2 \geq c \cdot n^2$

if we substitute any values for c and n_0, we can't get

$$f(n) \geq c \cdot g(n).$$

so $f(n) \neq \Omega(g(n))$

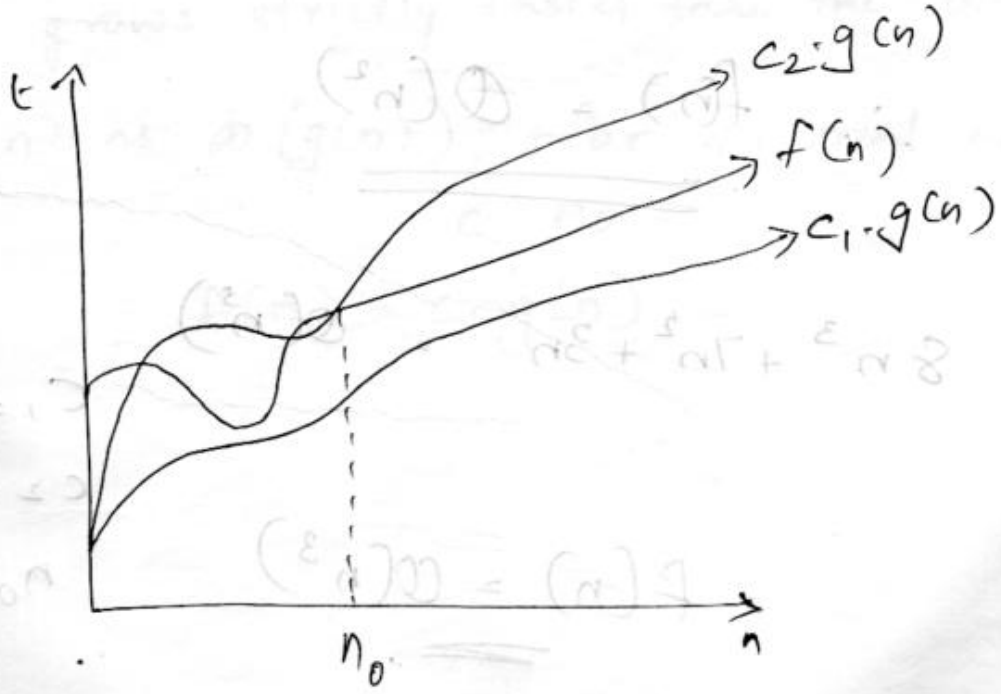
we can't lower bound n^2 to n.

$$n > \log n > \log\log n$$

Big Theta (Θ)

* Average case
* tight bound of a function.

Given functions $f(n)$ and $g(n)$, we say that $f(n)$ is $\Theta(g(n))$, if there are positive constants c_1, c_2 and n_0 such that $c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$ for all $n_0 \geq 0$, $n \geq n_0$, $c_1, c_2 \geq 0$



Eg: Show that $8n^2 + 7n = \Theta(n^2)$

$f(n) = 8n^2 + 7n \quad g(n) = n^2$.

c_1, c_2 and n_0 are positive constants.

$$c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n) \qquad n \geq n_0$$

Left hand inequality $c_1(n^2) \leq 8n^2 + 7n$ holds for any value of $n \geq 1$ and $c_1 = 8$

The right hand inequality holds for any value of $n \geq n_0 \geq 7$ and $c_2 = 9$

$$8 * n^2 \leq 8n^2 + 7n \leq 9(n^2) \qquad \text{for } c_1 = 8,\ c_2 = 9,\ n_0 = 7$$

$\therefore f(n) = \Theta(n^2)$

Eg: $8n^3 + 7n^2 + 3n = \Theta(n^3)$

$c_1 = 8$

$c_2 = 9$

$n_0 = 3$.

$\therefore f(n) = \Theta(n^3)$

Little (o) notation

"Little o" (o) notation is used to describe an upper bound that cannot be tight.

When we say that a function $f(n)$ is $o(g(n))$, we are essentiall stating that $f(n)$ grows slower than $g(n)$ as n approaches infinity.

If $f(n) = o(g(n))$, it means that $g(n)$ grows faster than $f(n)$.

$$f(n) < c \cdot g(n)$$

Little (ω) notation

Little ω notation is used to describe the relationship between two functions when one grows strictly faster than the other.

$f(n)$ is $\omega(g(n))$, for all real constants c, n_0.

$$f(n) > c \cdot g(n).$$

Recurrence relation

A recurrence relation, in mathematics and computer science, is a way to define a sequence of values using one or more previous terms in the sequence. It is a mathematical equation or expression that describes the relationship between the current term in the sequence and one or more of its preceding terms. They are frequently used to analyze the time complexity of algorithms and to describe recursive algorithms.

$$T(n) = \begin{cases} 1 & n = 0 \\ T(n-1)+1 & , \; n > 0 \end{cases}$$

Iteration Method

We solve recurrence relations using the iteration method. In this method, we keep substituting the smaller terms again and again until we reach the base condition. Thus the base term can be replaced by its value, and we get the value of the expression.

Eg)

$$T(n) = \begin{cases} 1 & n = 0 \\ T(n-1) + 1 & n > 0 \end{cases}$$

$$T(n) = T(n-1) + 1 \quad \text{---(1)}$$

substitute $n-1$ → $T(n)$ in (1)

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

Substitute

$$T(n) = T(n-2) + 1 + 1$$
$$= T(n-2) + 2 \quad \text{---(2)}$$

Substitute $n-2$ in (1)

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

$$T(n) = T(n-3) + 1 + 2$$
$$= T(n-3) + 3 \quad \text{---(3)}$$

continue for k times

$= T(n-k) + k$

put $n-k = 0$

$\therefore k = n$

$T(n-k) + k$

$= T(n-n) + n$

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

$\Rightarrow \underline{O(n)}$

Eg:-)

$$T(n) = \begin{cases} 1 & n = 0 \\ T(n-1) + n & n > 0 \end{cases}$$

$$T(n) = T(n-1) + n \quad \text{——— (1)}$$

Substituting we get

$$\begin{cases} T(n) = T(n-1) + n \\ T(n-1) = T(n-1-1) + n-1 \\ \quad = T(n-2) + n-1 \end{cases}$$

$$T(n) = T\left[(n-2) + n-1\right] + n$$

$$= T(n-2) + (n-1) + n \quad \text{——— (2)}$$

Substituting we get

$$\begin{cases} T(n-2) = T(n-2-1) + n- \\ \quad = T(n-3) + n-2 \end{cases}$$

$$T(n) = T\left[\left[(n-3) + (n-2)\right] + (n-1) + n\right.$$

$$= T(n-3) + (n-2) + (n-1) + n \quad \text{——— (3)}$$

continuing for k times

$$T(n) = T(n-k) + (n-(k-1)) + (n-(k-2)) +$$
$$\cdots\cdots + (n-1) + n \quad \text{---------- (4)}$$

Assume $n - k = 0$

$\therefore \quad n = k.$

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

$$= T(0) + 1 + 2 + 3 + \cdots + (n-1) + n$$

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

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

Eg:

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

$$T(n) = 2T(n-1)+1 \quad \text{---(1)}$$

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

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

$$= 2^2 T(n-2) + 2 + 1 \quad \text{---(2)}$$

$$= 2^2[2T(n-3)+1]+2+1$$

$$= 2^3 T(n-3) + 2^2 + 2 + 1 \quad \text{---(3)}$$

$$\vdots$$

K times

$$= 2^k T(n-k) + \underbrace{2^{k-1} + 2^{k-2} + \cdots + 2^2 + 2 + 1}_{2^k - 1}$$

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

put $n-k=0$

$k=n$

$$= 2^n T(0) + 2^n - 1$$

$$= 2^n + 2^n - 1$$

$$= 2^{n+1} - 1 \Rightarrow \underline{O(2^n)}$$

$2^n + 2^n = 2^{n+1}$

$2^2 + 2^2 = 8$

$2^2 + 2^2 = 2^{2+1}$

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

$T(n) = 2T(n/2) + n$ ——— (1)

$= 2\left[2T\left(\frac{n}{2^2}\right) + \frac{n}{2}\right] + n$ ($2n/2 = n$)

$= 2^2 T\left(\frac{n}{2^2}\right) + n + n$ ——— (2)

$= 2^2\left[2T\left(\frac{n}{2^3}\right) + \frac{n}{2^2}\right] + 2n$

$= 2^3 T\left(\frac{n}{2^3}\right) + n + 2n$

$= 2^3 T\left(\frac{n}{2^3}\right) + 3n$ ——— (3)

Let us take $\frac{n}{2^k} = 1$

k times

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

$\frac{n}{2^k} = 1$

$2^k = n$

$\log_2 n = k$.

$2^4 = 16$

$\log_2 16 = 4$

Assume $\frac{n}{2^k} = 1$ $\therefore n = 2^k$

$k = \log n$

$T(n) = 2^k T(1) + kn$

$= n \times 1 + n \log n$

$= O(n \log n)$

Let $n = 16$

$\log 16 = 4$

$2^4 = 16$

$\log_2 16 = 4 \Rightarrow 2^4 = 16$.

$2^{\log n} = n$ Fundamental relation

$\log_2 8 = 3$ because

$= 2^3 = 8$

$\log_2 n = x$ we have

$x = \log_2 n$

$2^x = n$.

Divide and Conquer

Breaking large problems into smaller sub problems.

Divide and conquer paradigm consists of three steps at each level of recursion

* Divide

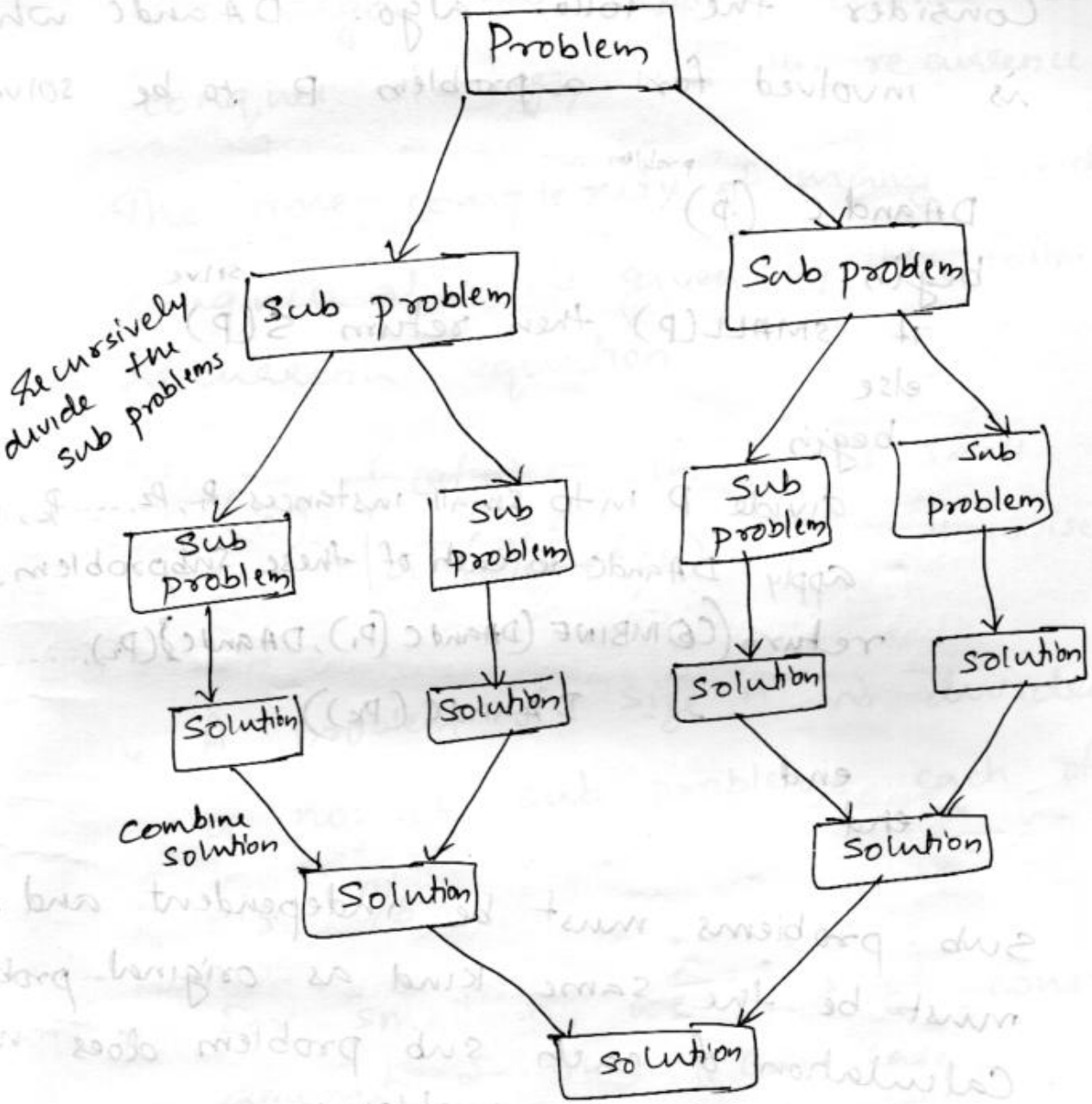
  Divide the problem into a no: of sub problems that are smaller instances of the same problem.

* Conquer

  Conquer the sub problem by solving them recursively. If they are small enough, solve the sub problems as base case

* Combine

  Combine the solutions to the sub problems into the solution for the original



Control Abstraction

A control abstraction is a procedure whose flow of control is clear but whose primary operations are specified by other procedures whose precise meanings are left undefined.

Consider the follo. algo. DAandC which is involved for a problem P to be solved.

```
DAandC (P)        problem
begin
  if SMALL(P) then return S(P)      solve
  else
   begin
     divide P into small instances P1, P2, .... Pk, k≥1
     apply DAandC to each of these subproblems
     return (COMBINE (DAandC (P1), DAandC (P2), ....
                ... DAandC (Pk))
   end
end
```

sub problems must be independent and it must be the same kind as original problem. Calculation of each sub problem does not affect other sub problems.

Eg: A larger no: of objects to be counted. We assign some peoples to count the object (sub problems).
Independent means count of each person's is independent. It does not depend on others count.

An algo. that applies divide and conquer strategy uses the recurrence relation

The time complexity of many Divide and conquer algo. is given by the follo. general recurrence equation.

$$T(n) = \begin{cases} O(1) & \text{if } n \text{ is small} \\ aT(n/b) + f(n) & \text{if otherwise} \end{cases}$$

* A problem of size n is divided in a no: of sub problems each of size 'n/b'
* For small n, we need a constant amount of time (Base case)
* f(n) time is needed for dividing and combining.

$$f(n) = D(n) + C(n)$$

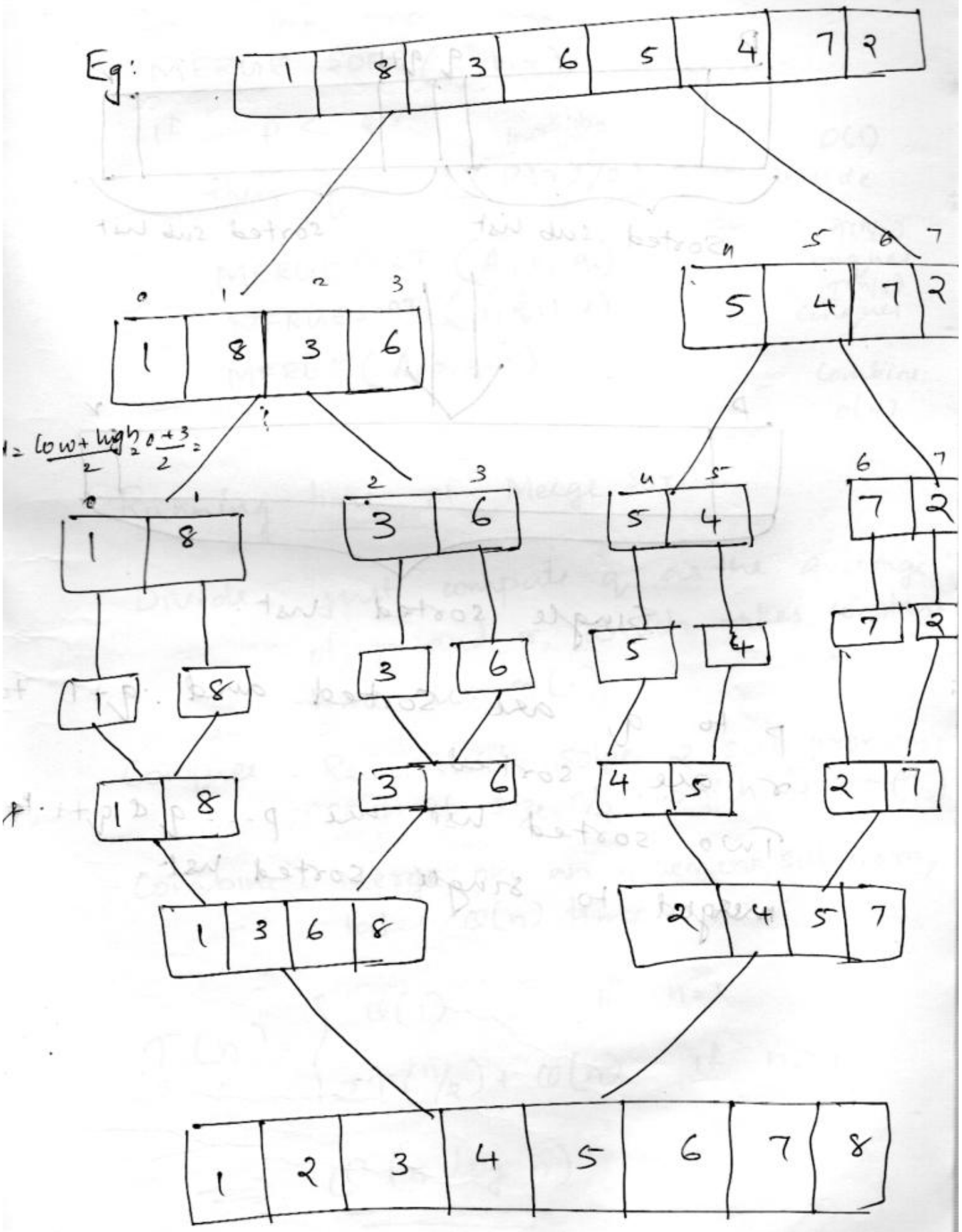
Merge Sort

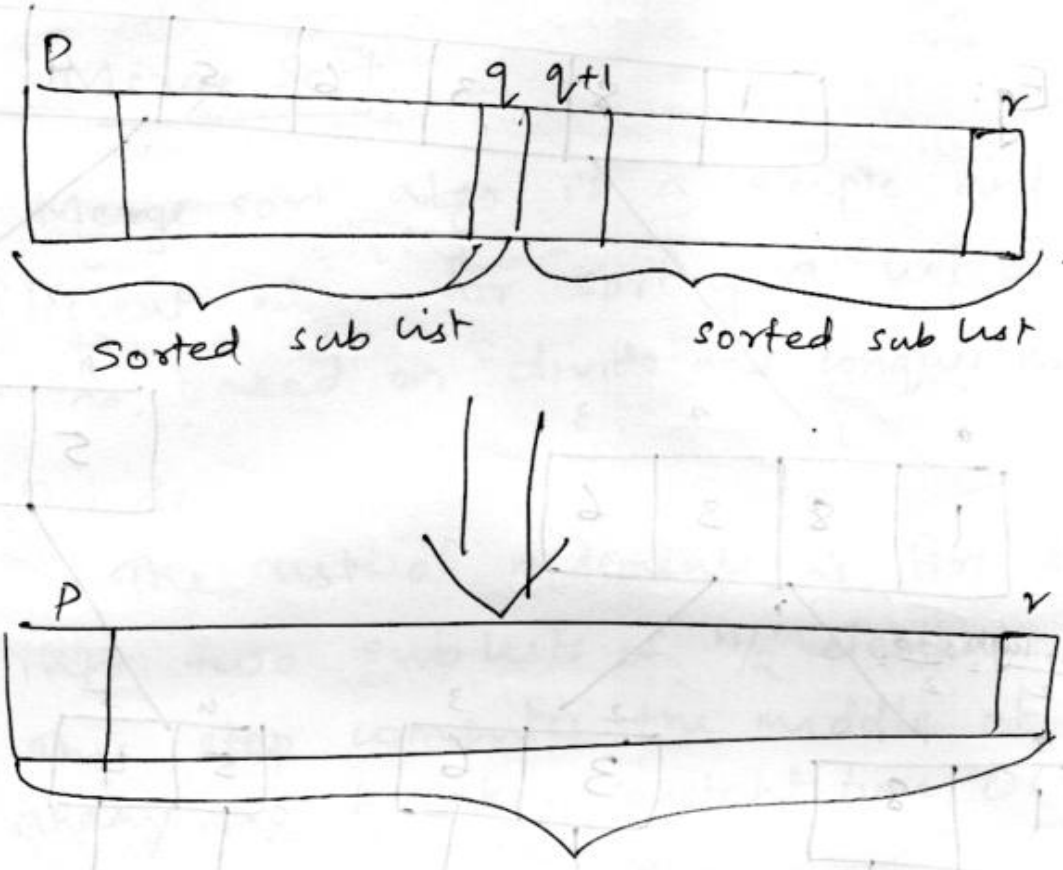
Merge sort algo. is a simple and very efficient algo. for sorting a list of numbers It is based on divide and conquer paradigm.

1. Divide -
The list of n elements is first divided into two sublists of $n/2$ elements. This step computes the middle of the array, so it takes constant time, $O(1)$.

2. Conquer -
Each half is sorted independently by following the same three steps for each half. As each sub problem of size $(n/2)$ is recursively solved in this step, so this step contributes $T(n/2) + T(n/2)$ to the running time

3. Combine -
Then the two sorted halves are merged to obtain a sorted sequence. This requires the merging of n elements into one list, so it contributes $O(n)$ to the running time.


Eg:
1 8 3 6 5 4 7 2
0 1 2 3
1 8 3 6
5 4 7 2
= low+high / 2 = 0+3 / 2 =
1 8
3 6
5 4
7 2
1
8
3
6
5
4
7
2
1 8
3 6
4 5
2 7
1 3 6 8
2 4 5 7
1 2 3 4 5 6 7 8



p to q are sorted and q+1 to r are sorted.

Two sorted list are p...q & q+1...r merged to single sorted list.

```
MERGE SORT (A, p, r)
if  p < r                                          O(1)
  then q ← ⌊(p+r)/2⌋   (mid; low+high/2)   - Divide
  MERGESORT (A, p, q)                      - Conquer  T(n/2)
  MERGESORT (A, q+1, r)                    - Conquer  T(n/2)
  MERGE (A, p, q, r)                       - Combine  O(n)
```

Running time of Merge sort

Divide : just compute q as the average of p and r, which takes constant time i.e. $\Theta(1)$

Conquer - Recursively solve 2 sub problems each of size $n/2$, which is $2T(n/2)$

Combine - Merge on an n-element sub array takes $\Theta(n)$ time.

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

$$\Rightarrow \underline{\Theta(n \log n)}$$

Quick sort

It uses divide and conquer approach

* Pick an arbitrary element of the array (the pivot)
* Divide the array into two segments, those that are smaller and those that are greater with the pivot in between
* Recursively sort the segments to the left and right of the pivot.

Eg: (35) 50 15 25 80 20 90 45 *

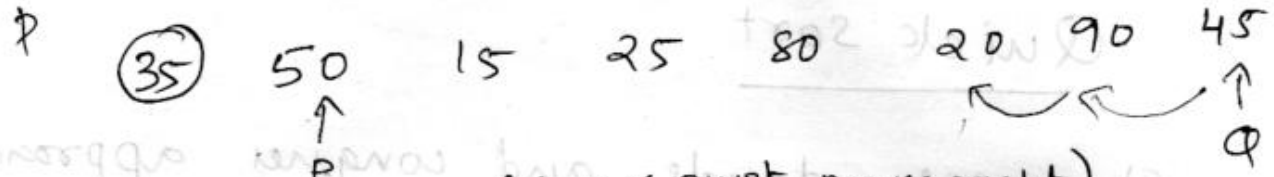
P → ↑ ← Q

Here pivot = 35

P points to 50 & Q points to 45

P moves right until greater than pivot
Q moves left until less than pivot.

ie. If P > pivot - P stops
If Q < pivot - Q stops.
If P & Q crossed, exchange Q with pivot. otherwise exchange P & Q.



Check P (if P ≤ pivot moves right)

I 50 > 35 - stops.

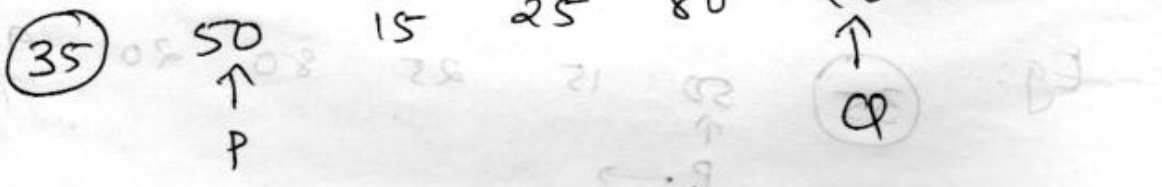
Then check Q (Q ≥ pivot moves left)

45 ≥ 35 - Yes - Decrement / move.

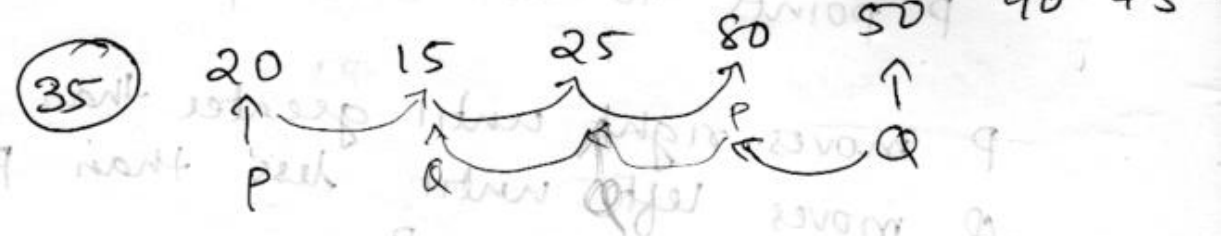
90 ≥ 35 - Yes - "

20 ≥ 35 - No stops.

Now Q is at 20



Swap 50 and 20



II Check

15 ≥ 35 - No - Move

25 ≥ 35 - No - Move

80 ≥ 35 - Yes - stops

P is now at 80

50 ≥ 35 - Y - Move left.

80 ≥ 35 - Y - "

25 ≥ 35 - No - stops

(35) 20 15 25 80 50 90 45 +∞.

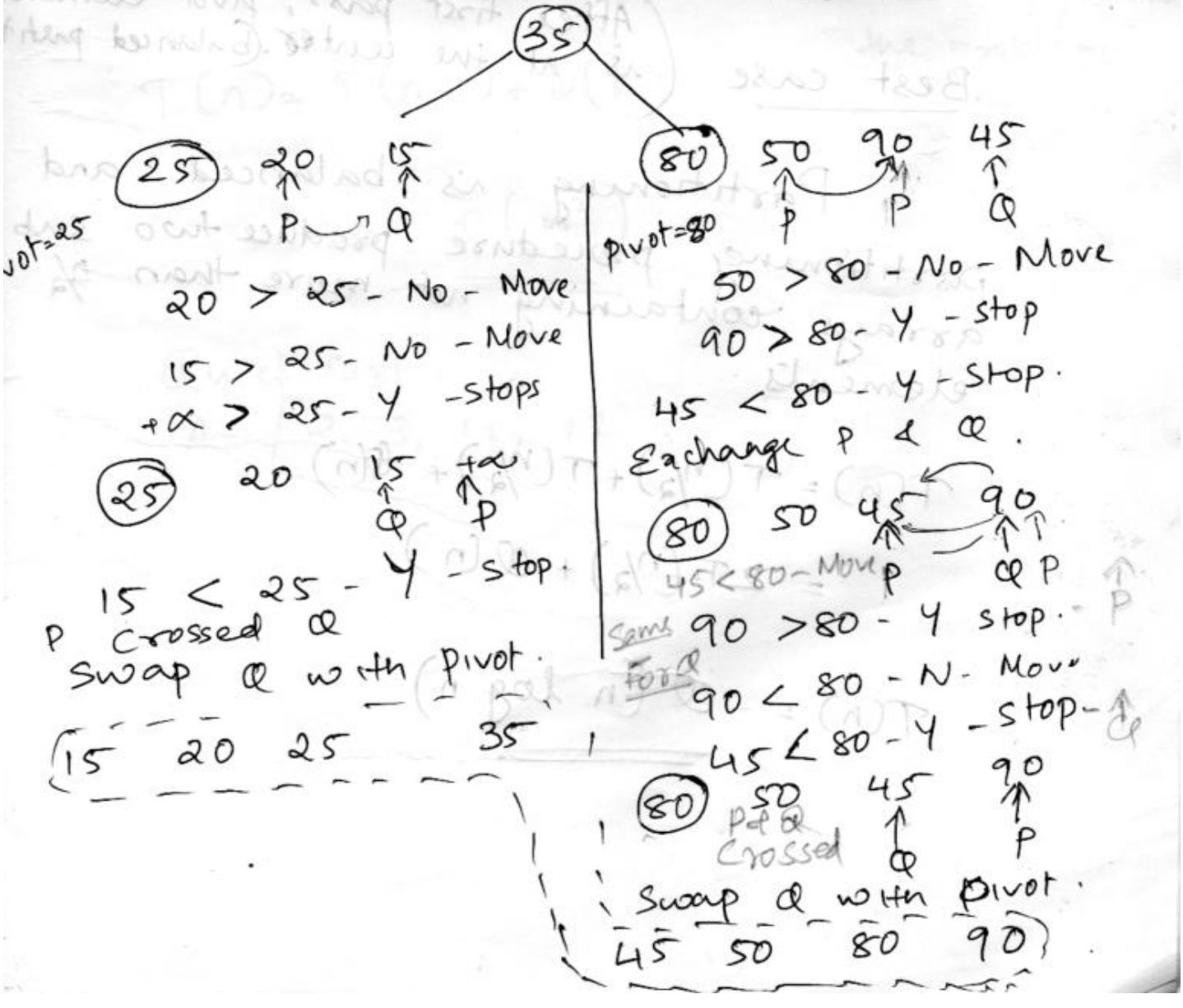
P and Q crossed. Swap Q with 25

35 pivot

ass 1

25 20 15 (35) 80 50 90 45 +∞.

Now pivot is at right place.
Less than pivot element is at the left,
and greater than pivot is at the right.



In quick sort dividing the problem into sub problems will be linear time.

```
QUICKSORT (A, p, r)
  if p < r
    q = PARTITION(A, p, r)
    QUICKSORT (A, p, q-1)
    QUICKSORT (A, q+1, r)
```

Analysis

Best case (After first pass, pivot element is at the center. (balanced partition)

Partitioning is balanced and partitioning procedure produce two sub arrays containing not more than $n/2$ elements.

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

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

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

Worst case

When the list is in the reverse order other than the desired order, the quick sort algo. will partition the list into two sub lists such that one list will contain zero/one element and other will contain n-1 elements.

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

Partition algo.

$O(n)$ - Scan entire array / divide the problem.

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

$$\Rightarrow \underline{\underline{O(n^2)}}$$

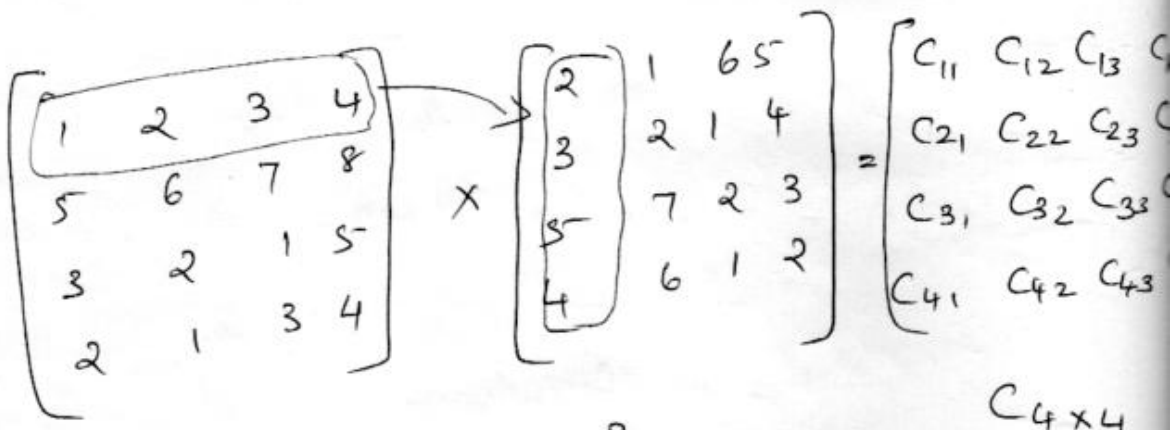
Quick sort.

? $A = [2, 3, 18, 17, 5, 1]$

Matrix Multiplication

We can multiply two matrices A and B if number of columns of A = Number of rows of B.

We consider only square matrices of size n (n is power of 2). If n is not a power of 2, matrices can be padded with zeros.



$C_{11} = 1\times2 + 2\times3 + 3\times5 + 4\times4$

$C_{12} = 1\times1 + 2\times2 + 3\times7 + 4\times6$

```
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];
```

Time complexity $O(n^3)$

Matrix Multiplication - Divide & Conquer approach

1. Divide matrices A and B (each of size $n \times n$) in 4 sub-matrices of size $n/2 \times n/2$
2. Calculate the values recursively.
   $ae+bg$, $af+bh$, $ce+dg$ and $cf+dh$

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \times \begin{bmatrix} e & f \\ g & h \end{bmatrix} = \begin{bmatrix} ae+bg & af+bh \\ ce+dg & cf+dh \end{bmatrix}$$

a is of size $n/2$
b is of size $n/2$
c, d, ... h is of size $n/2$

a, b, c, & d are sub matrices of A, of size $n/2 \times n/2$
e, f, g & h " B, "

In this method, we do 8 matrix multiplications for matrices of size $n/2 \times n/2$ and 4 matrix additions

```
MMult (A, B, n)
{
if n=1, output AxB // base case
else
{ 1. Compute a, b, c, d, e, f, g, h submatrices
        of size n/2 by dividing A and B
   2. X11 ← MMult(a, e, n/2) + MMult(b, g, n/2)
      X12 ← MMult(a, f, n/2) + MMult(b, h, n/2)
      X21 ← MMult(c, e, n/2) + MMult(d, g, n/2)
      X22 ← MMult(c, f, n/2) + MMult(d, h, n/2)
   3. Output X // Result matrix
   }
}
```

* Addition of two matrices takes $O(n^2)$

* Recurrence equation

$$T(n) = \begin{cases} O(1) & \text{if } n = 1 \\ 8T(n/2) + O(n^2), & \text{otherwise} \end{cases}$$

$$O(n^3)$$

Strassen's Method

* The main idea of Strassen's method is to reduce the number of recursive calls to 7.

* Strassen's method is similar to above simple divide and conquer method in the sense that this method also divide matrices to sub matrices of size $n/2 \times n/2$. But in this method, the four sub matrices of result are calculated using following formulae.

$$P_1 = a(f-h) \qquad P_2 = (a+b)h$$

$$P_3 = (c+d)e \qquad P_4 = d(g-e)$$

$$P_5 = (a+d)(e+h) \qquad P_6 = (b-d)(g+h)$$

$$P_7 = (a-c)(e+f)$$

$A \times B$ can be calculated using above seven multiplications.

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \times \begin{bmatrix} e & f \\ g & h \end{bmatrix} = \begin{bmatrix} P_5 + P_4 - P_2 + P_6 & P_1 + P_2 \\ P_3 + P_4 & P_1 + P_5 - P_3 - P_7 \end{bmatrix}$$

A B X

In this method we do

7 matrix multiplications for matrices of size $n/2 \times n/2$ and some matrix additions and subtractions.

Algorithm

```
Strassen (A, B, n)
{
if n = 1, output A x B // base case
else
{
1. compute a, b, c, d, e, f, g, h sub matrices
      of size n/2 by dividing A and B
2. P1 ← strassen(a, f-h, n/2)
   P2 ← strassen(a+b, h, n/2)
```

$P_3 \leftarrow$ Strassen $(c+d, e, n/2)$

$P_4 \leftarrow$ Strassen $(d, g-e, n/2)$

$P_5 \leftarrow$ Strassen $(a+d, e+h, n/2)$

$P_6 \leftarrow$ Strassen $(b-d, g+h, n/2)$

$P_7 \leftarrow$ Strassen $(a-c, e+f, n/2)$

3. $X_{11} \leftarrow P_5 + P_4 - P_2 + P_6$

$X_{12} \leftarrow P_1 + P_2$

$X_{21} \leftarrow P_3 + P_4$

$X_{22} \leftarrow P_1 + P_5 - P_3 - P_7$

4. Output X // Result Matrix

}

}

Analysis

In this method, for multiplying two matrices of size $n \times n$ we do 7 matrix multiplications and for matrices of size $n/2 \times n/2$ and some Matrix additions and subtractions

Addition and Subtraction of two matrices takes $O(n^2)$ time

Recurrence equation of approach is

$$T(n) = \begin{cases} a(1) & \text{if } n=1 \\ 7T(n/2) + a(n^2) & \text{, otherwise} \end{cases}$$

$$\Rightarrow O(n^{\log 7}) = O(n^{2.8074})$$