DESIGN AND ANALYSIS OF ALGORITHMS

UNIT- I

What is an algorithm

Definition: An algorithm is a finite set of instructions that, if followed, accomplishes a particular task. or solves a problem

A program is the expression of an algorithm in a programming language.

All algorithms must satisfy the following criteria (characteristics or properties of an algorithm).

1) Input: zero or more quantities are externally supplied as input. There may be some algorithms which do not take any input, but give an output.
Eg Program (algorithm) that displays "Hello world" message.

2) output: At least one quantity must be produced as output.

3) Definiteness: Each instruction is clear and unambiguous.
Each operation must be definite, meaning that it must be perfectly clear what should be done. The following type of instructions should not be present in an algorithm
"add 6 or 7 to x" or "compute 5/0"
It is not clear which of the two possibilities should be done or what the result is.

4) Finiteness: If we trace out the instructions of an algorithm, then for all cases, the algorithm must terminate after a finite number of steps. It should not go into an infinite loop.

5) Effectiveness: Every instruction must be very basic (not complex) so that it can be carried out, in principle, by a person using only pencil and paper.

Algorithms that are definite and effective are also called as computational procedures (functions).

We consider only computational procedures that always terminate.

Applications of algorithms

Nowadays algorithms (programs) are being used in every field

1) computer science
2) operations research
3) For analyzing complex electrical circuits.

steps in the study of algorithms

1) How to devise algorithms

For a given problem how to design an algorithm. For different problem there are different problem solving methods or algorithm design techniques. For example.

- Divide-and-conquer technique
- Backtracking method.
- Dynamic programming technique

- Greedy method
- Branch-and-bound method.

2) How to validate algorithms

once an algorithm is devised, it is necessary to show that it computes the correct answer for all possible inputs.

3) How to analyze algorithms.

If there are two or more algorithms to solve a problem, we select the best algorithm that requires minimum CPU time and system memory. This process is called performance analysis of algorithms.

4) How to test a program

After selecting the best algorithm, we will write the program in a programming language.

Program testing consists of two phases

Debugging is the process of executing a program on sample input data sets to determine whether any faulty results occur and if so correct them.

Performance measurement is the process of executing a correct program on input data sets and measuring the cpu time and system memory it takes to compute the results.

*The performance of an algorithm is estimated interms of its time complexity and space complexity.

ALGORITHM SPECIFICATION

We can describe an algorithm in many ways

1) we can use english language like statements

2) Graphic representations called flowcharts are another possibility, but they work well only if the algorithm is small and simple.

We present most of our algorithms by using a pseudo code that resembles C and Pascal language code.

Pseudo code conventions for expressing algorithms

```
while <condition> do
{
    <statement 1>
        :
    <statement n>
}
```

```
for  i:=1  to  n  do
begin
      Block of one or more statements
end;
```

```
if (condition) then
  <statement>
```

```
if(condition)  then
  <statement 1>
else
  <statement 2>
```

PERFORMANCE ANALYSIS OF AN ALGORITHM

There are many criteria upon which we can judge an algorithm

1) Does it do what we want it to do?
2) Does it work correctly according to the original specifications of the task?
3) Is there documentation that describes how to use it and how it works?
4) Are procedures created in such a way that they perform logical subfunctions?
5) Is the code readable? can any other person read and underst and your code easily?

Apart from these there are two main criteria for judging the performance of an algorithm. They are

Space complexity of an algorithm (program) is the amount of computer memory it (the algorithm) needs to run to completion.

Time complexity of an algorithm is the

amount of computer (CPU) time the algorithm needs to run to completion.

COMPUTING SPACE COMPLEXITY

The space needed by an algorithm is the sum of the following components.

1) A fixed part that is independent of the characteristics (e.g, number, size) of the inputs and outputs. This part typically includes the instruction space (i.e, space for the program code), space for simple variables and fixed-size component variables and constants [static data].

2) A variable part that consists of the space needed by component variables whose size is dependent on the particular problem instance being solved [Dynamic memory allocation]

The space requirement $S(P)$ of any algorithm P may be written as

$$\boxed{\underbrace{S}_{\text{space}}(P) = c + S_p(\text{instance characteristics})}$$

where c is a constant
when analyzing the space complexity of an algorithm, we concentrate solely on estimating Sp(instance characteristics).
-We compute the space complexity inte rms of number of memory words. A memory word is large enough (for example 64-bits) to store any value (int, long int, float, double).

Ex1

```
Algorithm abc(a,b,c)
{
return a+b+b*c+(a+b-c)/(a+b)+4.0;
}
```

The space needed for the above algorithm is 4 memory words (one each for a, b,c and result). The space needed by abc(a,b,c) algorithm is independent of instance characteristics

$S(abc) = c + S_{abc}$ (instance characteristics)

$S(abc) = 4 + 0$.

$S(abc) =$ <u>4 memory words.</u>

Example2 Iterative function for sum

```
Algorithm sum(a,n) // To find sum of the
                   // n elements of array a.
{
  s:=0.0;
  for i:=1 to n do
   {
     s:=s+a[i];
   }
  return s;
}
```

space needed is

one word for s

one word to store the value of i

one word for n

n words for storing n elements of array a.

$S(sum) = c + S_{sum}$ (instance characteristics)

$S(sum) = 3 + n = n+3$ words (memory words).

Example 3 Recursive function for sum

```
Algorithm Rsum(a,n)
{
 if(n≤0) then return 0.0;
 else
  return   Rsum(a,n-1)+a[n];
}.
```

The recursion stack space includes space for the formal parameters, the local variables, and the return address.

Each call for Rsum requires 3 words.

1 word for storing n value
1 word for storing pointer a[] } Rsum(a, n-1)
1 word for storing return address.

Rsum function will be called (n+1) times.

Rsum(a,n)
↓
Rsum(a,n-1) + a[n]
↓
Rsum(a,n-2) + a[n-1]
⋮
Rsum(a,1) + a[2]
↓
Rsum(a,0) + a[1].
↓
0.0 → sum is zero. since array size = 0.

∴ The recursion stack space needed
= 3(n+1).

COMPUTING TIME COMPLEXITY OF AN ALGORITHM

The time T(P) taken by a program P is the sum of the compile time and run (or execution) time. The compile time does not depend on the instance characteristics. We assume that a compiled program will be run (executed) several times without recompilation. We concern ourselves with just the run time of a program. This run time is denoted by t_p (instance characteristics).

A program step is defined as a syntactically or semantically meaningful segment of a program that has an execution time (takes some CPU time for execution).

Step counts To determine the number of steps needed by a program, we introduce a global variable count.

No. of executable statements —

Ex 1

Algorithm sum(a,n) // count:=0

{

s:=0.0; // count:= count+1 → 1 step.

for i:=1 to n do // count:= count+1 (n+1)

{

s:=s+a[i]; // count:= count+1 → n times → n steps

}

return s; // count:= count+1. → 1 step.

}

Total no. of steps required for algorithm

$= 1 + (n+1) + n + 1$

$= \underline{2n+3}$

Eg2

Algorithm Rsum(a,n) // count:= ~~count~~

{

if($n \leq 0$) then //count:=count+1

{ return 0.0; // count:= count+1

}

else

{

return $\underbrace{Rsum(a,n-1)}_{1 + tRsum(a,n-1)} + a[n]$;

}

}

step for adding a[n], function calling, and return.

When analyzing a recursive program for its step count, we obtain a recursive formula

$t_{Rsum}(0) = 2$

$$\boxed{t_{Rsum}(n) = \begin{cases} 2 & \text{if } n \leq 0 \\ 2 + t_{Rsum}(n-1) & \text{if } n > 0 \end{cases}}$$

These recursive formulas are referred to as recurrence relations. We derive the total step count by using recursive substitution

$t_{Rsum}(n) = 2 + \underline{t_{Rsum}(n-1)}$ → 1st step of derivation

$= 2 + 2 + t_{Rsum}(n-2) = 2 \times 2 + t_{Rsum}(n-2)$
2nd step

$= 2 \times 2 + 2 + t_{Rsum}(n-3) = 3 \times 2 + t_{Rsum}(n-3)$
3rd step.

Similarly at nth step, we can write

$= n \times 2 + t_{Rsum}(n-n)$

$= 2n + t_{Rsum}(0)$

$t_{Rsum}(n) = \underline{2n + 2}$

The runtime is proportional to n.

It grows linearly with n.

Eg3

Algorithm matrixadd (a, b, c, m, n)

```
{
for i:=1 to m do — (m+1) times → (m+1) steps
  {
    for j:=1 to n do — m(n+1)
      {
        c[i,j] := a[i,j] + b[i,j]; — mn
      }
  }
}
```

$C = A+B$

$$C = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & & & \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}_{m\times n} + \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1n} \\ b_{21} & b_{22} & \cdots & b_{2n} \\ \vdots & & & \\ b_{m1} & b_{m2} & \cdots & b_{mn} \end{bmatrix}_{m\times n}.$$

Total steps required

$$T(\text{matrixadd}) = (m+1) + m(n+1) + mn$$
$$= m+1 + mn + m + mn$$
$$= 2mn + 2m + 1.$$

$$T(\text{matrixadd}) \propto mn$$

$$\boxed{T(\text{matrixadd}) = O(mn)}.$$

if $m==n$, i.e matrices having same no. of rows and columns. Then time complexity of matrix addition $\boxed{T(\text{matrixadd}) = O(n^2)}$

Time complexity of matrix multiplication

```
Algorithm matrixmul(a,b,c,n,n)
{
for i:=1 to n do                              → (n+1)
{
  for j:=1 to n do                            → n(n+1)
  {
    for k:=1 to n do                          → n×n×(n+1)
    {
      c[i,j] := c[i,j] + a[i,k] × b[k,j];     – n×n×n
    }
  }
}
}
```

Total no. of steps required

$$T(\text{matrixmul}) = (n+1) + n(n+1) + n\times n\times(n+1) + n\times n\times n$$

$$= n+1+n^2+n+n^3+n^2+n^3$$

$$T(\text{matrixmul}) = 2n^3+2n^2+2n+1.$$

$$T(\text{matrixmul}) \propto n^3$$

$$\boxed{\therefore\ T(\text{matrixmul}) = O(n^3)}$$

STEP TABLE METHOD

The second method to determine the step count of an algorithm is to build a step table, in which we list the number of steps contributed by each statement (instruction).

s/e — steps per execution : no. of steps involved in an instruction

frequency: total number of times each statement will be executed.

Eg1

statement	s/e	frequency	Total steps
1. Algorithm Sum(a,n)	0	–	0
2. {	0	–	0
3. s:=0.0;	1	1	1
4. for i:=1 to n do	1	n+1	n+1
5. {	0	–	0
6. s:=s+a[i];	1	n	n
7. }	0	–	0
8. return s;	1	1	1
9. }	0	–	0
			2n+3

Eg2

statement	s/e	frequency $n\le 0$	frequency $n>0$	Total steps $n\le 0$	Total steps $n>0$
1. Algorithm Rsum(a,n)	0	–	–	0	0
2. {	0	–	–	0	0
3. if(n≤0) then	1	1	1	1	1
4. {	0	–	–	0	0
5. return 0.0;	1	1	0	1	0
6. }	0	–	–	0	0
7. else	0	–	–	0	0
8. {	0	–	–	0	0
9. return Rsum(a,n-1)+a[n]	1+x	0	1	0	1+x
10 }	0	–	–	0	0
11. }	0	–	–	0	0
$x = t_{Rsum}(n-1)$.				2	2+x.

statement	S/e	frequency	Total steps
Algorithm Add(a, b, c, m, n)	0	-	0
{	0	-	0
for i:=1 to m do	1	m+1	m+1
{	0	-	0
for j:=1 to n do	1	m(n+1)	mn+m
{	0	-	0
c[i, j] := a[i, j] + b[i, j];	1	mn	mn
}	0	-	0
}	0	-	0
}	0	-	0

Total no. of steps required = $2mn + 2m + 1$

BEST CASE, WORST CASE, AVERAGE CASE Time complexities of an algorithm

1) Best case time complexity of an algorithm is the minimum no. of steps required for the algorithm to solve the problem for the given parameter

2) worst case Maximum no. of steps —"—

3) Average case Average no. of steps —"—

Eg Linear (sequential) search

Given an array which need not be in sorted order.

search for an element x.

```
Algorithm    Linearsearch (a, n, x)
{
 for i:=1 to n do        size of array = n.
  {
   if (a[i] == x) then
   write(" element x found at position = i", i);
   return;
  }
 write(" element x not found in array a");
}
```

$n = 10$

	1	2	3	4	5	6	7	8	9	10
a	4	2	10	6	9	15	8	20	15	30

Best case search for $x = 4$

Luckly if the element is found in the first location of the array, it takes min.

no. of comparisons = 1.

$\therefore$ Time complexity is $O(1)$

worst case

search for $x = 30$

If the element is present in the last n^{th} location location of the array, then we need no. of comparisons = n

$\therefore$ Time complexity = $O(n)$.

Average case

To search for	No. of comparisons required
$x = 4$	1
$x = 2$	2
$\vdots$	
$x = 30$	10

Total no. of comparisons required $= \frac{n(n+1)}{2}$

Average no. of comparisons required $= \frac{\cancel{n}(n+1)/2}{\cancel{n}}$

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

αn

$\therefore$ Time complexity $= O(n)$.

ASYMPTOTIC NOTATIONS $(O, \Omega, \Theta, o, \omega)$

These notations are used to express the time complexities of algorithms.

Asymptotic Approaching a value or curve arbitrarily closely. Approaching nearer.

Definition [Big "oh"] The function $f(n) = O(g(n))$ (read as "f of n is big oh of g of n"). iff there exist positive constants c and n_0 such that $\boxed{f(n) \leq c \times g(n)}$ for all n, $n \geq n_0$.

1) $3n+2 = O(n)$ as $3n+2 \le 4n$ for all $n \ge 2$

$c = 4$. $n_0 = 2$,

2) $100n+6 = O(n)$ as $100n+6 \le 101n$ for all $n \ge 6$

c n_0

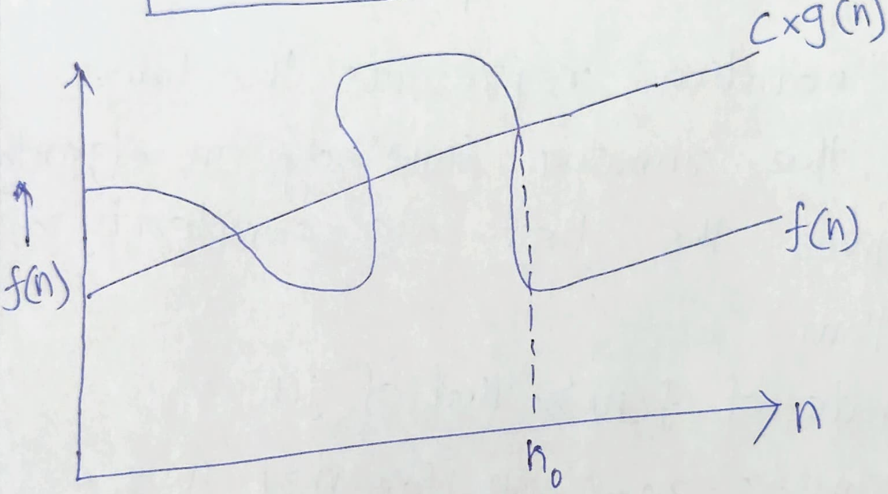
3) $10n^2+4n+2 = O(n^2)$ as $10n^2+4n+2 \le 11n^2, \forall n \ge 5$

c n_0

4) $6 \times 2^n + n^2 = O(2^n)$ as $6 \times 2^n + n^2 \le 7 \times 2^n, \forall n \ge 4$

c n_0

$$\boxed{f(n) \le c \times g(n)}$$



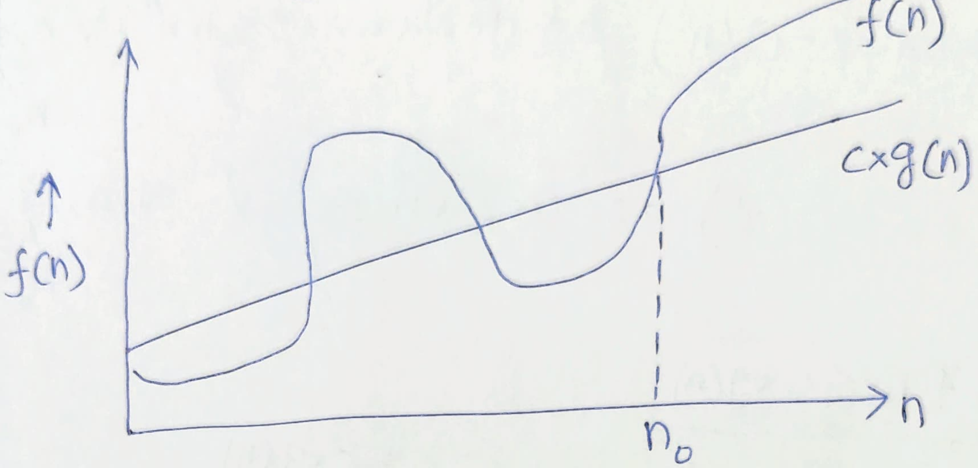
Big-Oh notation represents the upper bound of the running time of an algorithm. Thus it gives the worst-case complexity of an algorithm. Growth rate of $f(n)$ is $\le$ that of $g(n)$.

For $n \ge 2$, $3n+2 \le 2n^2$ we never say

$$3n+2 = O(n^2) \times$$

$$3n+2 \ne O(n^2).$$

Definition[Omega] The function $f(n) = \Omega(g(n))$ [read as "f of n is omega of g of n"] iff there exist positive constants c and n_0 such that $\boxed{f(n) \geq c \times g(n)}$ for all n, $n \geq n_0$



Omega notation represents the lower bound of the running time of an algorithm. Thus, it gives the best-case complexity of an algorithm.

Growth rate of $f(n) \geq$ that of $g(n)$.

$3n+2 = \Omega(n)$ as $3n+2 \geq 3n$ for $n \geq 1$ $n_0 = 1$, $c=3$

$100n+6 = \Omega(n)$ as $100n+6 \geq 100n$ for $n \geq 1$, $n_0=1$, $c=100$

$10n^2+4n+2 = \Omega(n^2)$ as $10n^2+4n+2 \geq 10n^2$, for $n \geq 1$, $c=10$.

$6 \times 2^n + n^2 = \Omega(2^n)$.

Definition [Theta] The function $f(n) = \Theta(g(n))$.
[read as "f of n is theta of g of n"]
iff there exist positive constants c_1, c_2 and n_0 such that $\boxed{c_1 \times g(n) \leq f(n) \leq c_2 \times g(n)}$ for all n, $n \geq n_0$

$3n+2 = \Theta(n)$ as $3n \leq 3n+2 \leq 4n$, for all $n \geq 2$

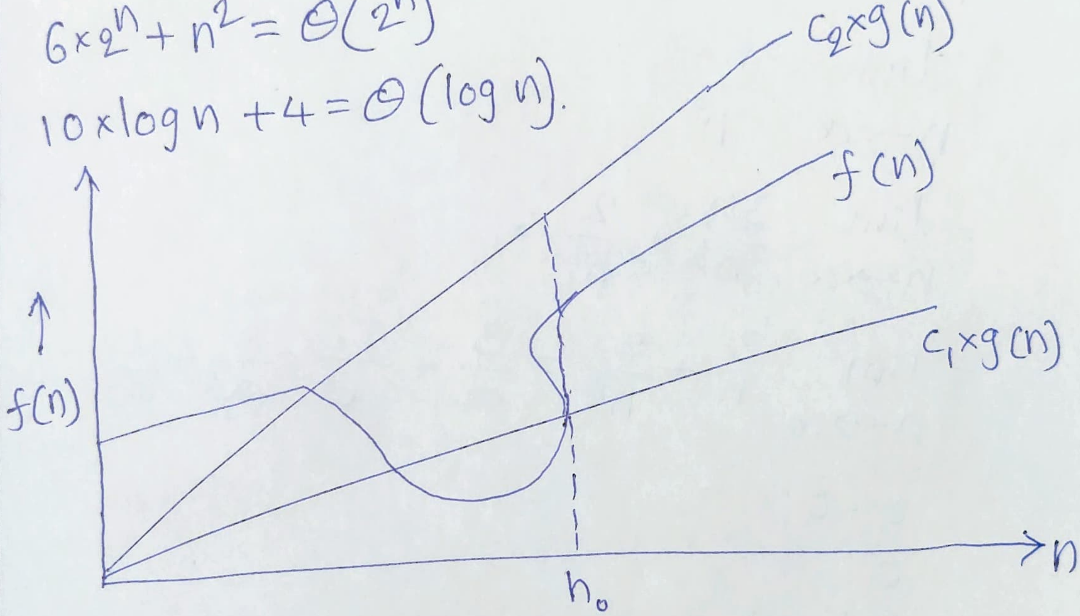
$3n+2 \geq 3n$ for all $n \geq 2$, $c_1 = 3$

$3n+2 \leq 4n$ for all $n \geq 2$, $c_2 = 4$, $n_0 = 2$

$10n^2 + 4n + 2 = \Theta(n^2)$

$6 \times 2^n + n^2 = \Theta(2^n)$

$10 \times \log n + 4 = \Theta(\log n)$.



Theta notation encloses the function $f(n)$ from above and below, since it represents the upper and the lower bound of the running time of an algorithm, it is used for analyzing the average-case

complexity of an algorithm
Growth rate of f(n) = growth rate of g(n).

Definition [Little "oh"] The function $f(n) = o(g(n))$ [read as "f of n is little oh of g of n"]

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

Eg $3n+2 = o(n^2)$

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

$= \lim_{n\to\infty} \frac{3n+2}{n^2}$

$= \lim_{n\to\infty} \frac{3\cancel{n}}{n^{\cancel{2}}} + \frac{2}{n^2}$

$= \lim_{n\to\infty} \frac{3}{n} + \frac{2}{n^2} = \frac{3}{\infty} + \frac{2}{\infty^2} = \frac{3}{\infty} + \frac{2}{\infty}$

$= 0 + 0$

$= 0$

$3n+2 = o(n \log n)$

Definition [Little omega] The function $f(n) = \omega(g(n))$ [read as "f of n is little omega of g of n"]

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

Eg $3n^2 = \omega(n)$

since $\lim\limits_{n \to \infty} \frac{g(n)}{f(n)}$

$= \lim\limits_{n \to \infty} \frac{\cancel{n}}{3n^{\cancel{2}}}$

$= \lim\limits_{n \to \infty} \frac{1}{3n} = \frac{1}{3 \times \infty} = \frac{1}{\infty} = 0.$

PRACTICAL COMPLEXITIES

The complexity function is useful in determining how the time requirements change. vary as the instance characteristics

The complexity function can also be used to compare two algorithms P and Q that perform the same task.

If there exist two algorithms P and Q to solve a problem and if P takes time $O(n)$ and Q takes time $O(n^2)$ then algorithm P is faster (better) than Q.

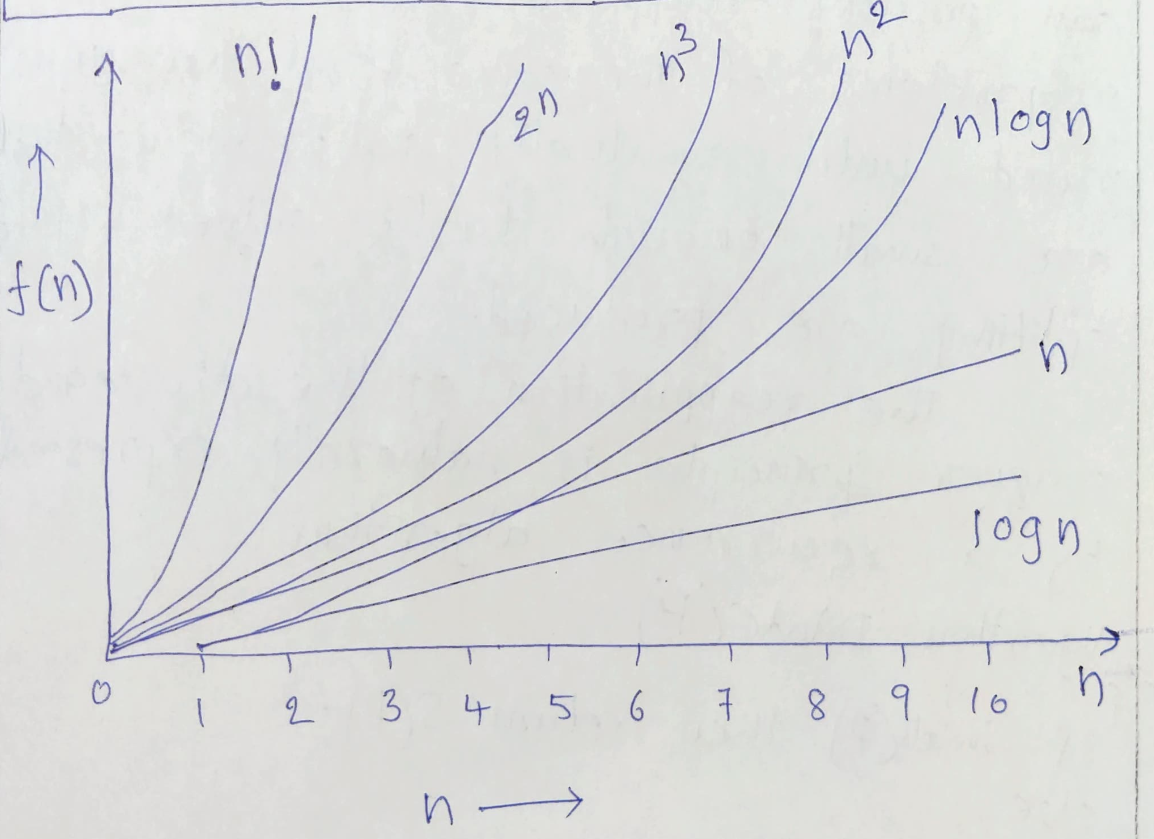
The algorithm which takes less time is the best algorithm.

Time f(n) complexity function	Function name	Example programs
$O(1)$	constant constant time algorithms. Takes same time for all n values Doesn't change.	Constant time complexity O(1) occurs when the program doesn't contain any loops, recursive function calls, or call to any other functions. The run time in this case won't change no matter what the input value is. - Find a number is even or odd - push(), pop() functions - Accessing an array element a[i] with index i
$O(\log_2 n)$	Logarithmic	Binary search
$O(n)$	Linear	Linear search
$O(n\log_2 n)$	Linearithmic	merge sort
$O(n^2)$	Quadratic	matrix addition Quick sort, Bubble sort
$O(n^3)$	Cubic	Matrix multiplication
$O(2^n)$	Exponential	Find all subsets of a given set Traveling salesperson problem
$O(n!)$	Factorial time complexity	Find all permutations of a given set/string.

$O(1)$	n	$\log_2 n$	$n\log_2 n$	n^2	n^3	2^n	$n!$
constant	1	0	0	1	1	2	1
	2	1	2	4	8	4	2
	4	2	8	16	64	16	24
	8	3	24	64	512	256	6720
	16	4	64	256	4096	65536	
	32	5	160	1024	32768	4294967296	

For sufficiently large value of n

$$\boxed{O(1) < O(\log n) < O(n) < O(n\log n) < O(n^2) < O(n^3) < O(2^n) < O(n!)}$$



$\log_2 1 = 0.$

DIVIDE-AND-CONQUER

GENERAL METHOD

Given a function to compute on n inputs The divide-and-conquer strategy suggests splitting the inputs into K distinct subsets $1<K\leq n$, yielding K subproblems. These subproblems must be solved, and then a method must be found to combine subsolutions into a solution of the whole.

If the subproblems are still large, then the divide-and-conquer strategy can possibly reapplied. Smaller and smaller subproblems of the same kind are generated until eventually subproblems that are small enough to be solved without splitting are produced.

The reapplication of the divideand conquer principle is naturally expressed by a recurrence algorithm

```
Algorithm DAndC(P)
{
if Small(P) then return S(P);
else
{
Divide P into smaller (instances)
subproblems P1, P2, ..., PK, K>=1
```

Apply DAndC to each of these subproblems;
return Combine(DAndC(P_1), DAndC(P_2), ..., DAndC(P_k));
}
}

Control abstraction for divide-and-conquer

P is the problem to be solved.

Small(P) is a Boolean-valued function that determines whether the input size is small enough that the answer can be computed without splitting.

If that is so, the function S is invoked.

S(P) - solution of problem P.

Combine is a function that determines the solution to P by using the solutions of the K subproblems.

If the size of P is n and the sizes of the K subproblems are $n_1, n_2, \ldots, n_K$ respectively. then

The computing time of the DAndC is described by the recurrence relation

$$\boxed{T(n) = \begin{cases} g(n) & \text{if } n \text{ is small} \\ T(n_1) + T(n_2) + \ldots + T(n_K) + f(n), & \text{otherwise} \end{cases}}$$

g(n) is the time to compute the answer directly for small input problems.
The function f(n) is the time for dividing p and combining the solutions of subproblems.
The complexity of many divide-and-Conquer algorithms is given by recurrence relations of the form

$$\boxed{T(n) = \begin{cases} T(1) & \text{if } n=1 \\ aT(n/b)+f(n) & \text{if } n>1 \end{cases}}$$

where a and b are known constants
we assume that T(1) is known and n is a power of b (i.e $n=b^k$).
Substitution method is used to solve recurrence relations.

Applications of Divide-and-Conquer

1) Binary search
2) Merge sort
3) Quick sort
4) strassen's matrix multiplication.

BINARY SEARCH

Let a_i, $1 \le i \le n$, be a list of elements that are sorted in nondecreasing (increasing) order.

Determine whether a given element x is present in the list. If x is present then determine a value j such that $a_j == x$.

search problem $P = (n, \underset{a_1}{\underset{\downarrow}{a_i}}, a_{i+1}, \ldots, \underset{a_n}{\underset{\downarrow}{a_l}}, x)$

Small(P) is true if $n=1$, in this case S(P) will take value i if $x = a_i$ otherwise (if $x \neq a_i$) it will take the value 0.

If P has more than one element, it can be divided into a new subproblem as follows.

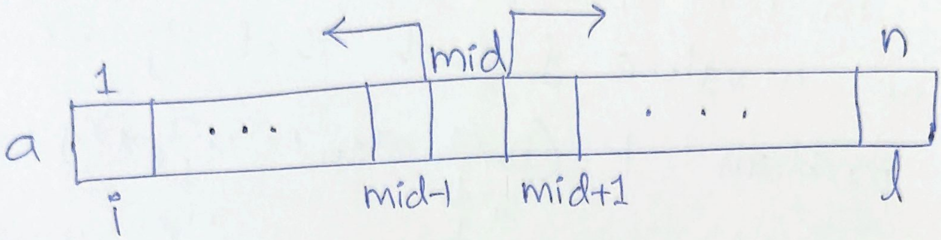
Pick an index q and compare x with a_q

Three possibilities

1) If $x == a_q$, P is immediately solved.
2) If $x < a_q$, search for x in the left subarray $a_i, a_{i+1}, \ldots a_{q-1}$
3) If $x > a_q$, search for x in the right subarray $a_{q+1}, a_{q+2}, \ldots a_l$.

Division of array into two subarrays takes only $O(1)$ time.

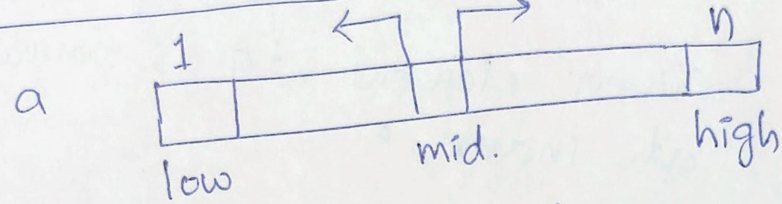
If q is always chosen such that a_q is the middle element that is $q = \lfloor (n+1)/2 \rfloor$, then the resulting algorithm is known as binary search. There is no need to combine the solutions.



Algorithm BinSrch (a, i, l, x)
// Given an array a[i : l] of elements in
// nondecreasing order; $1 \leq i \leq l$, determine
// whether x is present, and if so, return
// array index j such that x = a[j];
// else return 0.
{
if (l = i) then // If Small (P)
{ if (x = a[i]) then return i;
else return 0; // element x not found.
}
else
{ // Reduce P into a smaller subproblem
mid = $\lfloor (i+l)/2 \rfloor$;
if (x = a[mid]) then return mid;

```
else if(x<a[mid]) then
return BinSrch(a,i,mid-1,x); //search will
   // proceed in the left subarray
else return BinSrch(a,mid+1,l,x);
//search will proceed in the right subarray.
}
}
```

Recursive Binary search




```
Algorithm BinSearch(a,n,x)
// a[1:n] , n≥0
{
 low:=1 ; high:=n;
while(low≤high) do
{
   mid:= ⌊(low+high)/2⌋;
   if (x<a[mid]) then high:=mid-1;
   else if (x>a[mid]) then low:=mid+1;
   else return mid;
}
return 0; //element x not found in array a.
}
```

Iterative Binary search

Search for a given element x in the following array

a[1:14]

1	2	3	4	5	6	7	8	9	10	11	12	13	14
-15	-6	0	7	9	23	54	82	101	112	125	131	142	151

– search for $x = 9$

low	high	$mid = \left\lfloor \frac{low+high}{2} \right\rfloor$	
1	14	7	$9 < a[7] = 54$
1	6	3	$9 > a[3] = 0$
4	6	5	$9 = a[5]$

Given element $x=9$ is found at index 5

– search for $x = -14$

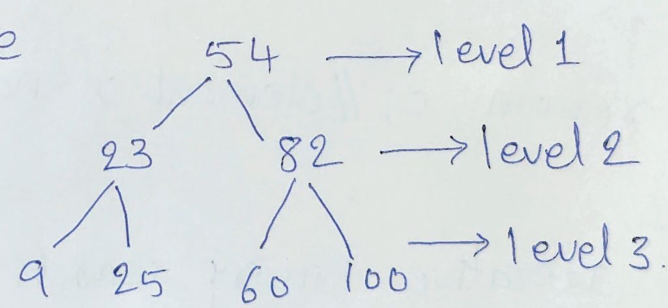
low	high	mid	
1	14	7	$-14 < a[7] = 54$
1	6	3	$-14 < a[3] = 0$
1	2	1	$-14 > a[1] = -15$
2	2	2	$-14 \neq a[2] = -6$

Given element $x = -14$ is not found in the array.

Time complexity a

1	2	3	4	5	6	7
9	23	25	54	60	82	100

Binary search tree



No. of comparisons required in Binary search = level of that element in its Binary search tree.

Best case time complexity

If given element $x = 54$ is matching with middle element [root] of the array then no. of comparisons required = 1

$\therefore$ Time complexity $= O(1)$.

Worst case

~~Leaf~~ element level $= \log_2(n+1)$

$= \log_2(7+1)$

$= \log_2 8$

$= 3$.

If given element (eg $x=9$) is matching with a leaf, then no. of comparisons required $= 3 = \log_2(n+1) =$ level of x.

$\therefore$ Time complexity $= O(\log_2 n)$ we neglect the constant +1.

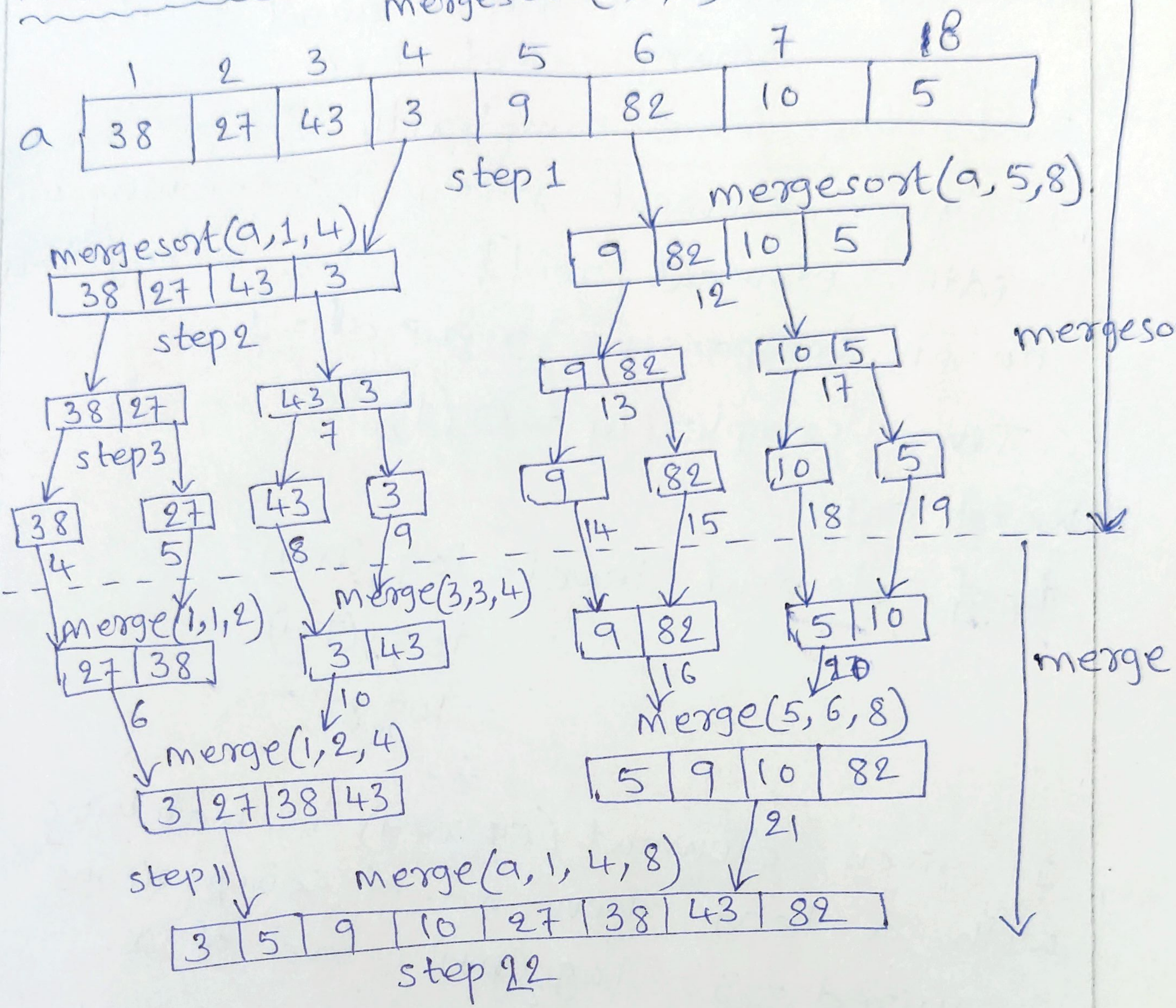
Average case

Avg. no. of comparisons $= \dfrac{1+2+2+3+3+3+3}{7} = \dfrac{17}{7}$

$= 2.43 \approx 3 = \log_2 8 = \log_2 n$

$\therefore$ Time complexity $= O(\log_2 n)$.

MERGE SORT



Given a sequence of n elements (also called keys) a[1], a[2], ... a[n]

Elements are to be sorted in nondecreasing (increasing) order $\boxed{\text{Eg } 5, 7, 9, 10, 10, 12, 14, 14, 15}$

split a[] into two sets (subarrays)

$a[1], a[2], \ldots a[\lfloor n/2 \rfloor]$ and $a[\lfloor n/2 \rfloor + 1], \ldots a[n]$.

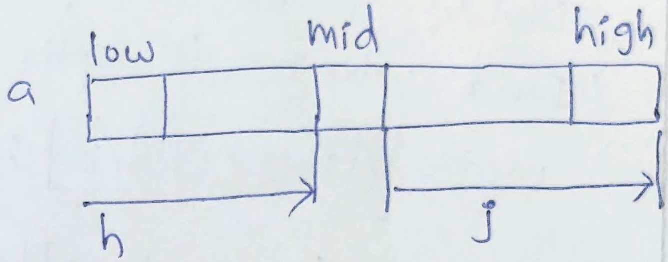
Each set is individually sorted, and the resulting sequences are merged to produce a single sorted sequence of n elements.

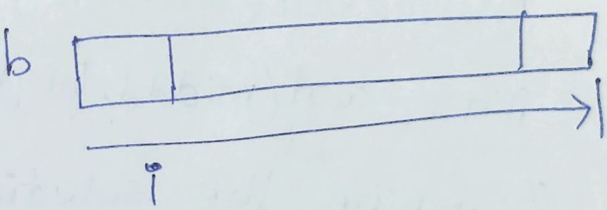
Algorithm Mergesort(low, high)

```
// a[low : high] is a global array to be sorted
// Small(P) is true, if there is only one
// element to sort. In this case the array
// is already sorted.
{
 if(low<high) then // If there are more than
                   // one elements
 {
  // Divide P into subproblems
  // Find where to split the array
    mid = ⌊(low+high)/2⌋;
  // solve the subproblems
    Mergesort(low, mid);       ] sort two
    Mergesort(mid+1, high);    ] subarrays
  // combine the solutions
  // merge two sorted subarrays
    merge(low, mid, high);
 }
}.
```

Algorithm Merge(low, mid, high)

```
{
//a[low:high] is a global array containing
// two sorted subarrays in a[low:mid] and
// in a[mid+1, high]. The goal is to merge
// these two subarrays into a single array
// residing in a[low:high]. we need an
// auxiliary (additional) array b[] for merging.
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;
}
```





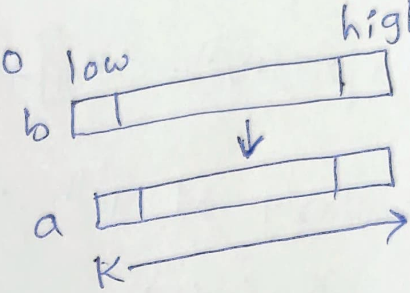
1 4 8 10 | 6 9 15 20

1 4 6 8 9 10

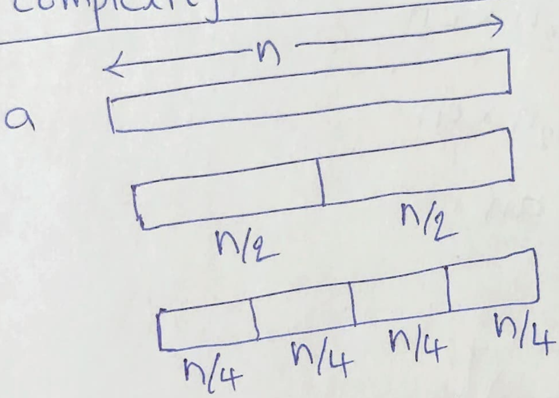
```
else
{
for k:=h to mid do
{
  b[i]:=a[k];
  i:=i+1;
}
for k:=low to high do
{
  a[k]:=b[k];
}
}
```

10 15 19 22 | 4 6 12 14

4 6 10 12 14 15



Time complexity of Mergesort (Derivation)



n elements

2 x n/2 elements

4 x n/4

If the time for the merging operation is proportional to n. Two sorted subarrays of size n/2 can be merged in time $O(n) \approx cn$. Then the computing time for merge sort is described by the recurrence relation

$$\boxed{T(n/2) = \begin{cases} a & \text{if } n=1;\ a \text{ is a constant} \\ 2T(n/2) + cn & \text{if } n>1;\ c \text{ is a constant} \end{cases}}$$

$T(1) = a.$

We assume that n is a power of 2

$n = 2^k$

$k = \log_2 n$

a

1	2	3
6	5	10

↓

a

1	2	3	4
6	5	10	0

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

$$= 2\left[2T(n/4) + c \cdot n/2\right] + cn = 4T(n/4) + 2n = 2^2 T(n/2^2) + 2cn$$

$$= 4\left[2T(n/8) + c \cdot n/4\right] + 2cn = 8T(n/8) + 3cn = 2^3 T(n/2^3) + 3cn$$

⋮

similarly at k^{th} step, we can write

$$T(n) = 2^k T(n/2^k) + kcn$$

$$= 2^k T(n/n) + kcn$$

$$= nT(1) + \log_2 n \times cn$$

$$= n \times a + \log_2 n \times cn$$

$$T(n) = cn\log n + an$$

$$T(n) \propto n\log n$$

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

This is the best case, average case, and worst case time complexity of merge sort.

Q: Write merge sort algorithm and derive its time complexity.

QUICK SORT

In Quicksort, the division into two subarrays is made so that the sorted subarrays do not need to be merged later.

Rearrange the elements in a[1:n] such that

$a[i] \leq a[j]$

for all i between 1 and m

for all j between m+1 and n for some m,

$1 \leq m \leq n$

Eg: The function is initially invoked (called) as Partition(a,1,10).

a[1]	a[2]	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)	i	j
65	70	75	80	85	60	55	50	45	+∞	2	9
pivot element											
65	45	75	80	85	60	55	50	70	+∞	3	8
65	45	50	80	85	60	55	75	70	+∞	4	7
65	45	50	55	85	60	80	75	70	+∞	5	6
65	45	50	55	60	85	80	75	70	+∞	6	5 (i>j) swap a[1]↔a[j]
60	45	50	55	$\boxed{65}$	85	80	75	70	+∞		
pivot					pivot						
Partition(a,1,4)					Partition(a,6,9).						

Function Partition produces two sets S_1 and S_2.

All elements in S_1 are ≤ All the elements in S_2.

Hence S_1 and S_2 can be sorted independently.

Each set is sorted by reusing the function Partion.

First element of the array is assumed to be pivot i.e partitioning element

---

Thus the elements in a[1:m] and a[m+1:n] can be independently sorted. No merge is needed.

The rearrangement of the elements is accomplished by picking some element of a[], say $t = a[s]$, and then reordering the other elements so that

All elements appearing before t in a[1:n] are $\leq t$ and

All elements appearing after t are $\geq t$.

This rearranging is referred to as partitioning.

Algorithm Quicksort(a, p, q)

```
// sorts the elements a[p], a[p+1], ..., a[q] which
// reside in the global array a[1:n] into
// ascending order; a[n+1] is considered to
// be defined and must be ≥ all the elements
//                          ∞
// in a[1:n].
{
if(p<q) then // If there are more than one
             // element
  {
  // divide P into subproblems(subarrays)
     j:=Partition(a, p, q+1);
```

```
// j is the position of the partitioning element.
// solve the subproblem. or sort the subarrays
  Quicksort(a, p, j-1);
  Quicksort(a, j+1, q);
// There is no need to merge the subarrays.
  }
}.
```

<u>Algorithm Partition(a, m, p)</u>

```
// within a[m], a[m+1], ..., a[p-1] the elements
// are rearranged in such a manner that
// if initially t = a[m], then after completion
// a[q] = t for some q between m and p-1,
// a[k] ≤ t for m ≤ k ≤ q, and a[k] ≥ t for
// q < k < p. q is returned. set a[p] = ∞.
{     pivot
  v := a[m]; i := m; j := p;
  repeat
  {
     repeat
         i := i+1;
     until(a[i] ≥ v);
     repeat
         j := j-1;
     until(a[j] ≤ v);
     if(i < j) then interchange(a, i, j);
  }
  until(i ≥ j);
```

a[m] := a[j]; a[j] := v; return j;

}

Algorithm interchange(a, i, j)

// swap a[i] and a[j]

{

temp := a[i];

a[i] := a[j];

a[j] := temp;

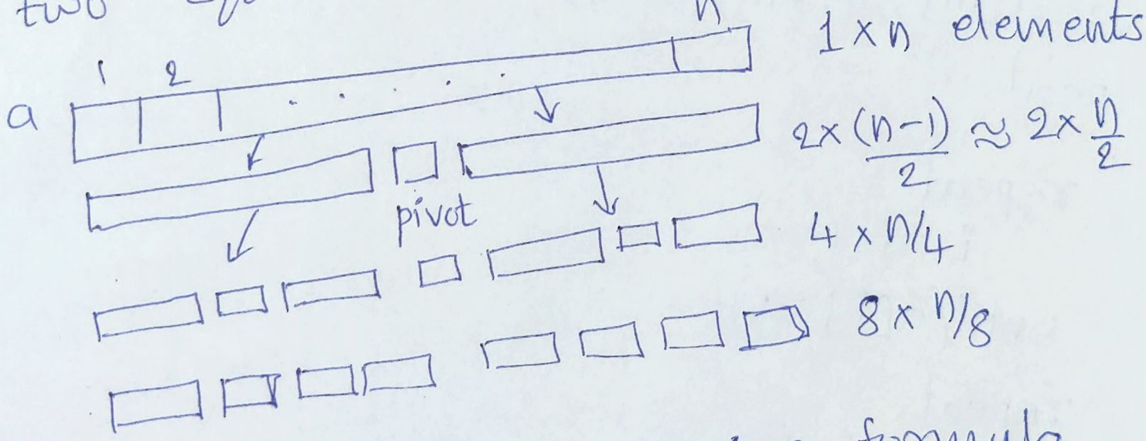
}

It is assumed that a[p] ≥ a[m] and that a[m] is the partitioning element.

Time complexity of Quicksort

Best case

In the best case, the pivot element is in the middle, which partitions the array into two equal sized subarrays.



∴ Time complexity recursive formula

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

cn – is the time taken for partitioning array

The running time = next Time taken for two recursive calls to sort subarrays + Linear time taken for partitioning the array. first

$n = 2^k$

$k = \log_2 n$

$T(n) = 2T(n/2) + cn$ 1st step

$= 2[2T(n/4) + c.n/2] + cn = 2^2 T(n/2^2) + 2cn$ 2nd step

$= 2^2[2T(n/8) + c.n/4] + 2cn = 2^3 T(n/2^3) + 3cn$ 3rd step

$\vdots$

similarly at k^{th} step, we can write

$T(n) = 2^k T(n/2^k) + kcn$

$= nT(n/n) + \log_2 n \times c \times n$

$= nT(1) + c \times n\log n$

$= na + c \times n\log n$

$T(n) = cn\log n + an$

$T(n) \propto n\log n$

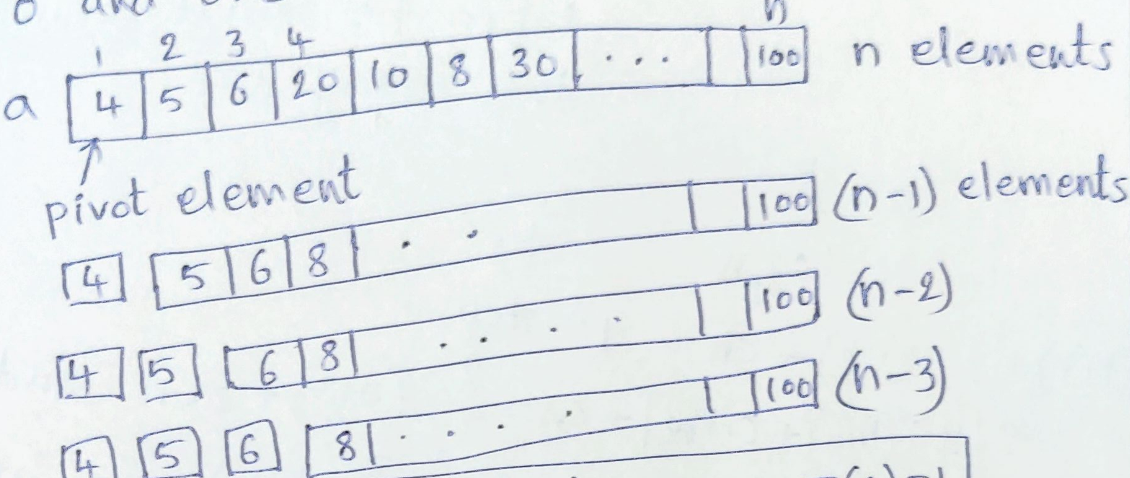
$\boxed{\therefore T(n) = O(n\log n)}$

Average case time complexity

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

Worst case

If we take pivot as smallest (1st element) the array would not be divided into two equal sized partitions, but one of length 0 and one of length (n-1)



$$\boxed{\begin{aligned} T(n) &= T(i) + T(n-i-1) + cn \qquad T(0)=1 \\ T(n) &= T(0) + T(n-1) + cn \qquad \text{if } i=0 \end{aligned}}$$

$$\begin{aligned} T(n) &= T(n-1) + cn \\ &= T(n-2) + c(n-1) + cn \\ &= T(n-3) + c(n-2) + c(n-1) + cn \\ &\vdots \end{aligned}$$

At nth step, we can write

$$\begin{aligned} T(n) &= T(n-n) + c\big[[n-(n-1)] + [n-(n-2)] + \cdots + n\big] \\ &= T(0) + c\left[1 + 2 + 3 + \cdots + (n-1) + n\right] \\ &= 1 + c \times \frac{n(n+1)}{2} \end{aligned}$$

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

$$T(n) \propto n^2$$

$$\boxed{\therefore\ T(n) = O(n^2)}$$

STRASSEN'S MATRIX MULTIPLICATION

1) Conventional matrix multiplication method

Let A and B be two $n \times n$ matrices

$C = A \times B$ is also an $n \times n$ matrix

$$\boxed{C(i,j) = \sum_{1 \le K \le n} A(i,K) \times B(K,j)}$$

To compute $C[i,j]$ using this formula we need n multiplications.

matrix C has $n \times n = n^2$ elements

The time for the resulting matrix multiplication algorithm is $\underline{O(n^3)}$

2) The divide-and-conquer strategy suggests another way to compute the product of two $n \times n$ matrices.

For simplicity we assume that n is a power of 2 (i.e $n = 2^k$). In case if n is not a power of 2, then enough rows and columns of zeros can be added to both A and B so that the resulting dimensions are a power of two.

$$\underset{3 \times 3}{\begin{bmatrix} 10 & 5 & 8 \\ 6 & 4 & 9 \\ 15 & 3 & 20 \end{bmatrix}} \longrightarrow \underset{\substack{4 \times 4 \\ 2^2 \times 2^2}}{\begin{bmatrix} 10 & 5 & 8 & 0 \\ 6 & 4 & 9 & 0 \\ 15 & 3 & 20 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}}$$

Imagine that A and B are each partitioned into four square submatrices having dimensions $\frac{n}{2} \times \frac{n}{2}$.

If AB is $\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \times \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$ then

$$\left.\begin{aligned} C_{11} &= A_{11}B_{11} + A_{12}B_{21} \\ C_{12} &= A_{11}B_{12} + A_{12}B_{22} \\ C_{21} &= A_{21}B_{11} + A_{22}B_{21} \\ C_{22} &= A_{21}B_{12} + A_{22}B_{22} \end{aligned}\right\} \text{— (1).}$$

This algorithm will continue applying itself to smaller-sized submatrices until n becomes suitably small (2×2) so that the product ~~can~~ can be computed directly.

To compute AB using (1) we need to perform eight multiplications of $n/2 \times n/2$ matrices and four additions of $n/2 \times n/2$ matrices. Two $n/2 \times n/2$ matrices can be added in time cn^2 for some constant c.

The overall computing time $T(n)$ of the resulting divide-and-conquer algorithm is given by the recurrence relation

$$\boxed{T(n) = \begin{cases} b & \text{if } n \le 2 \\ 8T(n/2) + cn^2 & \text{if } n > 2 \end{cases}}$$

Derivation of Time complexity

$$T(n) = 8\,\underline{T(n/2)} + cn^2$$
$$= 8\left[8T(n/4) + c\cdot\frac{n^2}{4}\right] + cn^2 = 8^2\,\underline{T(n/4)} + 3cn^2$$
$$= 8^2\left[8T(n/8) + c\cdot\frac{n^2}{4^2}\right] + 3cn^2 = 8^3\,T(n/2^3) + 7cn^2$$

⋮

At k^{th} step, we can write

$$T(n) = 8^k T(n/2^k) + (2^k - 1)cn^2$$

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

$$T(n) = 8^k T(n/n) + (n-1)cn^2$$
$$= (2^3)^k \times b + cn^3 - cn^2$$
$$= (2^k)^3 \times b + cn^3 - cn^2$$
$$T(n) = n^3 \times b + cn^3 - cn^2$$
$$T(n) \propto n^3$$
$$\therefore T(n) = O(n^3).$$

Again we got same $O(n^3)$. No improvement over the conventional matrix multiplication method has been made. We can attempt to reformulate the equations for C_{ij} so as to have fewer multiplications and possibly more additions.

Volker strassen's method for matrix multiplication

Volker strassen' has discovered a way to compute the C_{ij}s of equation (1) by using only 7 mutiplications and 18 additions or subtractions. His method involves first computing the seven $\frac{n}{2} \times \frac{n}{2}$ submatrices

P, Q, R, S, T, U and V.

$P = (A_{11} + A_{22})(B_{11} + B_{22})$

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

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

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

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

$U = (A_{22} - A_{11})(B_{11} + B_{12})$

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

$C_{11} = P + S - T + V$

$C_{12} = R + T$

$C_{21} = Q + S$

$C_{22} = P + R - Q + U$

The resulting recurrence relation for T(n) is

$$\boxed{T(n) = \begin{cases} b & \text{if } n \le 2 \\ 7T(n/2) + \underbrace{an^2}_{18 \times \frac{n}{2} \times \frac{n}{2}} & \text{if } n > 2 \end{cases}}$$

Time complexity derivation

$$T(n) = 7\,\underline{T(n/2)} + an^2$$

$$= 7\left[7T(n/4) + a\cdot\frac{n^2}{2^2}\right] + an^2 = 7^2\,\underline{T(n/4)} + an^2\left[1+\frac{7}{4}\right]$$

$$= 7^2\left[7T(n/8) + a\cdot\frac{n^2}{4^2}\right] + an^2\left[1+\frac{7}{4}\right] = 7^3T(n/2^3) + an^2\left[1+\frac{7}{4}+\frac{7^2}{4^2}\right]$$

$$\vdots$$

similarly at k^{th} step, we can write

$$T(n) = 7^k T(n/2^k) + an^2\left[1+\frac{7}{4}+\frac{7^2}{4^2}+\dots+\frac{7^{k-1}}{4^{k-1}}\right]$$

$$\approx 7^k T(n/n) + an^2\left(\frac{7}{4}\right)^k$$

$$\approx 7^k T(1) + an^2\,\frac{7^k}{4^k}$$

$$\approx 7^k \times b + an^2\,\frac{7^k}{4^k} \qquad n = 2^k,\quad k = \log_2 n$$

$$\approx 7^{\log_2 n} \times b + an^2 \times \frac{7^{\log_2 n}}{4^{\log_2 n}}$$

$$\approx 7^{\log_2 n} \times b + an^2 \times \frac{7^{\log_2 n}}{n^{\log_2 4}}$$

$$\approx 7^{\log_2 n} \times b + a\cancel{n^2} \times \frac{7^{\log_2 n}}{\cancel{n^2}}$$

$$\approx 7^{\log_2 n}(a+b) = c\,n^{\log_2 7} = c \times n^{2.81}$$

$$\boxed{\therefore\ T(n) \approx O(n^{2.81})}$$

Time complexity reduced from $O(n^3)$ to $O(n^{2.81})$.