UNIT-I

Introduction: Algorithm, Pseudo code for expressing algorithms, Performance Analysis-Space complexity, Time complexity, Asymptotic Notation- Big oh notation, Omega notation, Theta notation and Little oh notation ,Probabilistic analysis, Amortized analysis.

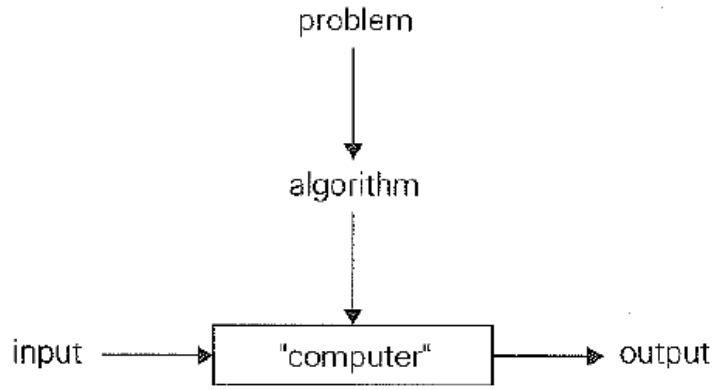
Divide and conquer: General method, applications-Binary search, Quick sort, Merge sort, Strassen's matrix multiplication.

1.1 ALGORITHM

Algorithm is a finite set of instructions that is followed, accomplishes a particular task.

(OR)

An algorithm is a sequence of unambiguous instructions for solving a problem, i.e., for obtaining a required output for any legitimate (lawful, legal, reasonable genuine) input in a finite amount of time.



There is something or someone capable of understanding and following the instructions given. We call this a "computer".

All algorithms must satisfy the following criteria:

1. **Input**: It may be zero or more quantities are externally supplied. But input not necessary for all Algorithms.
2. **Output**: At least one quantity is produced. It is must for all the algorithms
3. **Definiteness**: Each instruction is clear and unambiguous.
4. **Finiteness**: The instruction (Steps) of an algorithm must be finite. Means Algorithm terminate after finite number of steps (Instructions).
5. **Effectiveness**: Every instruction must be very basic so that it can be carried out, in principle, by a person using only pencil and paper. It is not enough that each operation be definite, it also must be feasible (possible or realistic).

Example:

Euclid's algorithm for computing gcd(m, *n)*

- Step 1 If $n = 0$, return the value of m as the answer and stop; otherwise, proceed to Step 2.
- Step 2 Divide m by n and assign the value of the remainder to r.

- Step 3 Assign the value of *n tom* and the value of *r ton.* Go to Step 1.

Validate algorithms:

If algorithm is devised (Means prepared), it is necessary to show that it computes the correct answer for all possible legal inputs. This process is called validation.

1.2 PSEUDOCODE

Algorithm can be described in many ways like

- English for specifying algorithm in step by step
- Flowchart for graphical representation.

But these two ways work well only if algorithm is small or simple. For large or big algorithm we are going to use pseudo code.

Pseudo code is a kind of structured English for describing algorithms. It allows the designer to focus on the logic of the algorithm without being distracted (diverted) by details of language syntax. At the same time, the pseudo code needs to be complete. It describes the entire logic of the algorithm so that implementation becomes a rote mechanical task of translating line by line into source code.

Alternatively, we can express the same algorithm in a **pseudocode**:

```
ALGORITHM Euclid(m, n)
//Computes gcd(m, n) by Euclid's algorithm
//Input: Two nonnegative, not -both-zero integers m and n
//Output: Greatest common divisor of m and n
while n!= 0 do
{
r :=m mod n;
m :=n;
n := r;
return m;
}
```

Pseudo code conventions:

1. " // " is used to provide comment line
2. Using of matching parenthesis "{}" to form blocks. A compound statement (i.e., Collection of simple statements) can be represented as a block.
3. An identifier begins with a letter. Data types of variables are not explicitly declared. But types will be clear from the context whether a variable is global or local to a procedure.
   For example:

```
node=record
{
        datatype_1 data_1;
                .
                .
        datatype_n data_n;
        node  *link;
}
```

4. Assignment of values to variables is done using assignment statement.
   <variable> := <expression>;
5. There are
   - Two Boolean values true and false.
   - Logical operator "and, or and not" instead of "&, ||,!" respectively.
   - Relation operator "$<, \leq, =, \neq, \geq$ and >".
6. Elements of multidimensional arrays are accessed using [and]. For example, if A is a two dimensional array, the (i, j)th element of the array is denoted as A[i, j]. Array indices (index) start at zero.
7. Looping statement are employed: **while** , **repeat until** and **for** as like below

General representation	Pseudo code representation

While (condition){
Stmt1;
Stmt n;
}
	While <condition> do {
	<stmt 1>;
	<stmt n>;
	}

do{
Stmt1;
Stmt n;
}
While(condition);
	repeat {
	<stmt 1>;
	<stmt n>;
	Until <condition>
	}

for(initialization;condition;expression)
{
Stmt1;
Stmt n;
}
	for variable :=value1 to n step step do
	{
	<stmt 1>;
	<stmt n>;
	}

8. A conditional statement: **if** and **switch** has the following forms:

General form	Pseudo code form

if (condition)
stmt1;
else
stmt 2
	If <condition> then
	<Stmt 1>:
	else
	<stmt 2>;

Switch (condition){
Case label: stmt1;
break;
.
default: stmt;
}
	case
	{
	:<condition 1> : <stmt1>
	.
	:<condition n>: <stmt n>
	:else: <stmt n+1>
	}

9. Input and output are done using the instructions read and write. No format is used to specify the size of input or output quantities.
10. There is only one type of procedure: **Algorithm**. An algorithm consists of a heading and a body. The heading takes the form

Algorithm Name (<parameter list>)

Algorithm for finds & returns the maximum of n given numbers	
In general way (program)	In algorithm (pseudo code)

A[n]={9,3,…};
Result=A[0];
for (i=1;i<=n-1;i++){
if A[i] > Result;
Result =A[i];
}
printf (Result);
	Algorithm max(A, n)
	// A is an array of size n
	{
	Result :=A[1];
	for i:=2 to n do{
	if A[i] > Result; then Result :=A[i];
	return Result;
	}

Algorithm for selection sorting	
Program	Algorithm

main(){
int s, i, j, temp, a[20];
for (i=0; i<s; i++){
for(j=i+1; j<s; j++){
temp=a[i];
a[i]=a[j];
a[j]=temp;
}
}
}
	Algorithm SelectionSort(a,n)
	// sor the array a of n-elements.
	{
	for i:=1 to n do {
	j:=i;
	for k:=i+1 to n do
	if(a[k]<a[j] then j:=k;
	t:=a[i];
	a[i];=a[j];
	a[j]:=t; }}

Recursive Algorithms:

A function is called itself then it called recursive function.

Void main(){
fun();
}
	Void fun(){
	fun();
	}

For example

int main(){
int i=5;
printf(fact(i));
}
	int fact(int i){
	if(i<=1){ return 1; }
	return i*fact(i-1);
	}

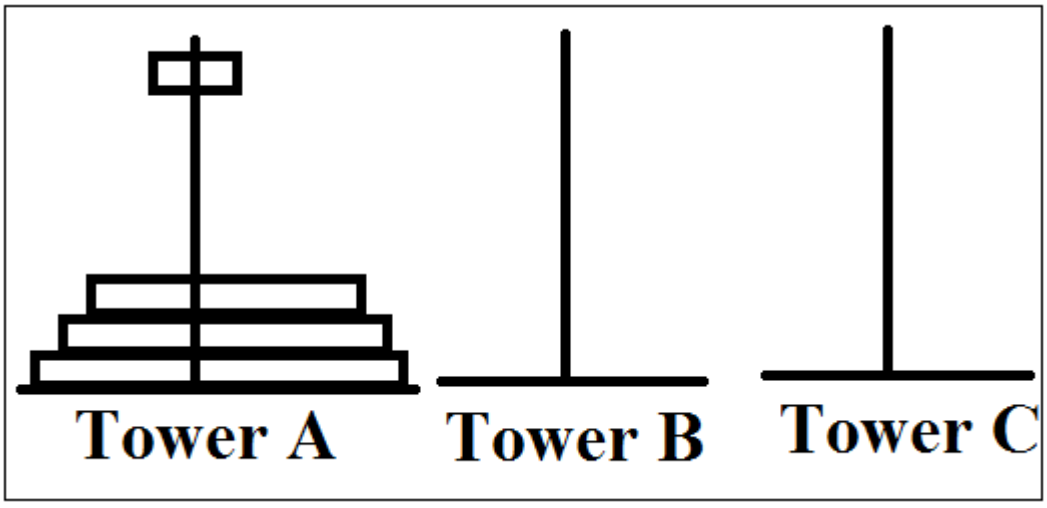
Similarly an algorithm is said to be recursive if the same algorithm is invoked in the body. There are two types of Recursive algorithms.

- Direct recursive algorithm→ An algorithm that calls itself is directive recursive.

- Indirect recursive algorithm→ An algorithm, if it calls another algorithm is indirect recursive. Means algorithm A is said to be indirect recursive if it calls another algorithm which in turn calls A.

Example for recursive algorithm:

Towers of Hanoi Problem:



- Tower A, B, and C are diamond towers.
- Tower A has 64 golden disks.
- The disks were of decreasing size and where stacked on the towers in decreasing order of size bottom to top.

Brahma priests have been attempting to move the disk from tower A to Tower B by using Tower C for intermediate storage.

A very elegant (Style or design or dress) solution for this towers of Hanoi by using Recursion.

Algorithm for solving "Towers of Hanoi" by using recursion algorithm:

```
Algorithm TowersOfHanoi(n, x, y, z)
//Move the top n disks form Tower x to Tower y
{
if(n≥1) then  TowerOfHanoi(n-1, x, z,y);
write("Move top disk form Tower x to top of tower y);
TowerOfHanoi(n-1,z,y,x);
}
}
```

Explanation:

Assume that the number of disks in n. To get the largest disk to the bottom of Tower y, we move the remaining n-1 disks to tower z. And then move the largest disk form tower x to tower b.

Now tower y has largest disk and tower x is empty and tower z has n-1 in decreasing order.

Here we left the remaining task that is moving n-1 disks form tower Z to tower Y. For this we use same procedure by using tower x is intermediate storage.

1.3 ANALYZE ALGORITHMS OR PERFORMANCE ANALYSIS

If an algorithm is executed, it used the

- Computer's CPU → to perform the operations.
- Memory (both RAM and ROM) → to hold the program & data.

Analysis of algorithms is the task of determining how much computing time and storage memory required for an algorithm.

They are many criteria upon which we can judge an algorithm to say it perform well.

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 sub functions?
5) Is the code readable?

Performance evaluation has two major phases

- Priori estimates→Performance analysis
- Posterior testing→ Performance measurements

Space Complexity:

The space complexity of an algorithm is the amount of memory it needs to run to completion. The space needed by algorithm is the sum of following components

- A fixed part→that is independent of the characteristics of input and output. Example Number & size. In this includes instructions space [space for code] + Space for simple variables + Fixed-size component variables + Space for constant & soon.

- A variable part→ that consists of the space needed by component variables whose size is dependent on the particular problem instance being solved + The space needed by referenced variables + Recursion stack space.

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

$S(P)=C+ S_P$

C→Constant

S_P→ Instance characteristics.

Find space complexity of Algorithm of sum of 'n' numbers.	**Find space complexity for Recursive algorithm:**

Algorithm:
Algorithm sum(a,n)
//Here a is array of Size n
{
S:=0;
if(n≤0)) then return 0;
for i:=1 to n do
s:=s+a[i];
return s;
}
	Algorithm:
	Algorithm RSUM(a,n)
	//a is an array of size n
	{
	If(n≤0) then return 0;
	Else return a[n]+RSUM(a,n-1);
	}

Space complexity S(P): s→1 word i→1 word n→1 word a→n word $S(P)\geq n+3$ Word is a String of bits stored in computer memory, its size is a 4 to 64 bits.	**Space Complexity:** The recursion stack space includes space for the formal parameters, the local variables and the return address. Return address requires→1 word memory Each call to RSUM requires→ at least 3words (It includes space for the values of n, the return address and a pointer to a[]). The depth recursion is n+1 The recursion stack space needed is ≥3(n+1)

Time complexity T(P):

Time complexity of an algorithm is the amount of computer time it needs to run to completion.

Here RUN means Compile + Execution.

Time Complexity $T(P)=t_c+t_p$

But we are neglecting t_c

Because the compile time does not depends on the instance characteristics. The compiled program will be run several times without recompilation.

So $T(P)= t_p$

Here t_p → instance characteristics.

For example the program p do some operations like ADD, SUB, MUL etc.

If we knew the characteristics of the compiler to be used, we could process to determine the number of additions, subtractions, multiplications, divisions, compares, loads, stores and so on.

We obtain $t_p(n)$ express as follow:

$t_p(n)=C_aADD(n)+ C_SSUB(n)+ C_mMUL(n)+ C_dDIV(n)+\ldots\ldots\ldots$

n→ Instance characteristics

$C_a, C_S, C_m, C_d, \ldots\ldots$ → time needed for an addition, Subtraction, multiplication, division, and etc.

ADD, SUBB, MUL, DIV → Are functions, whose values are the numbers of additions, subtractions, multiplications,.. etc, that are performed when the code for p is used on an instance with characteristics.

Program step: Program step is the syntactically or semantically meaningful segment of a program. And it has an execution time that is independent of the instance characteristics.

Example:

- For comment→ zero steps
- For assignment statements (Which does not involve any calls to other algorithms) →one step
- For iterative statements such as "for, while and until-repeat" statements, we consider the step counts only for control part of the statement.
  For while loop "while (<expr>) do " : the step count for control part of a while stmt is → Number of step counts for assignable to <expr>

For for loop ie for i:=<expr> to <expr1> do: The step count of control part of "for" statement is→ Sum of the count of <expr> & <expr1> and remaining execution of the "for" statement, i.e., one.

We determine the number of steps needed by a program to solve a particular problem. For this there are two methods.

1) **First method** is, introduce a new variable "count" in to the program for finding number of steps in program.
2) **Second method** is, building a table in which we list the total number of steps contributed by each statement.

Example for 1st method:

Find time complexity of Algorithm of sum of 'n' numbers.	**Find time complexity for Recursive algorithm:**

Algorithm:
Algorithm sum(a,n)
{
// count is global it is initially zero
S:=0;
Count:=count+1; // count for assignment
for i:=1 to n do
{
Count :=count+1; //for "for" loop
s:=s+a[i];
count :=count+1; //for assingment
}
Count :=count+1; //for last time of for loop
Count :=count+1; // for return stmt
return s;
}
	Algorithm:
	Algorithm RSUM(a,n)
	{
	Count :=count+1; // for if condition
	If(n≤0) then
	{
	Count:=count+1; // for return stmt
	return 0;
	}
	Else {
	Count :=count+1;
	//for the addition, function invocation & return
	return a[n]+RSUM(a,n-1);
	}
	}

Finally count values is =2n+3; So total number of steps= 2n+3	If n=0 then $t_{Rsum}(0)=2$ If n>0 then increases by 2, ie., $2+ t_{Rsum}(n-1)$ Means $t_{Rsum}(n)=2+ t_{Rsum}(n-1)$ $=2+2+ t_{Rsum}(n-2)$ $=2+2+2+ t_{Rsum}(n-3)$ . . $=2(n)+ t_{Rsum}(n-n)$ $=2n+ t_{Rsum}(0)=2n+2$

Example for 2nd method:

Find time complexity of Algorithm of sum of 'n' numbers.			
Statements	s/e	Frequency	Total steps
1. Algorithm sum(a,n) 2. { 3. S:=0; 4. for i:=1 to n do 5. s:=s+a[i]; 6. return s; 7. }	0 0 1 1 1 1 0	_ _ 1 n+1 n 1 _	0 0 1 n+1 n n 1 0
			2n+3

Find time complexity for Recursive algorithm					
Statements	s/e	Frequency		Total steps	
		n=0	n>0	n=0	n>0
1. Algorithm RSUM(a,n) 2. { 3. If(n≤0) then 4. return 0; 5. Else 6. return a[n]+RSUM(a,n-1); 7. }	0 0 1 1 0 1+x 0	_ _ 1 1 0 _	_ _ 1 0 1 _	_ _ 1 1 0 0 0	_ _ 1 0 1+x 0
				2	2+x

Here x= $t_{Rsum}(n-1)$

1.4 ASYMPTOTIC NOTATION

A problem may have numerous (many) algorithmic solutions. In order to choose the best algorithm for a particular task, you need to be able to judge how long a particular solution will take to run.

Asymptotic notation is used to judge the best algorithm among numerous algorithms for a particular problem.
Asymptotic complexity is a way of expressing the main component of algorithms like

- Cost
- Time complexity
- Space complexity

Etc.

Some Asymptotic notations are

- Big oh→O
- Omega→Ω
- Theta →θ
- Little oh→o
- Little Omega→ω

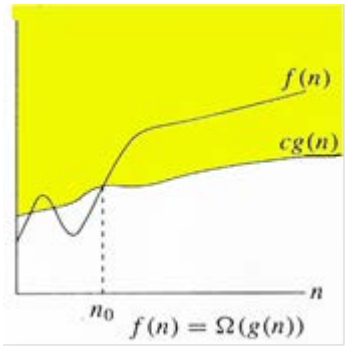
Big oh notation: O

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 **f(n)≤C*g(n)** for all n, n≥0

Example:

3n+2=O(n) as

3n+2 ≤4n for all n≥2



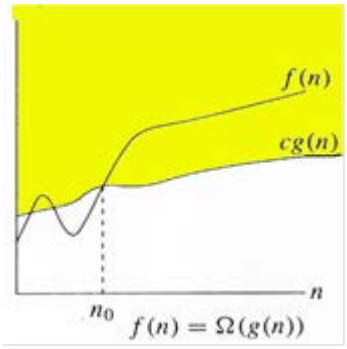
Omega notation: Ω

The function f(n)= Ω (g(n)) (read as "f of n is Omega of g of n") iff there exist positive constants c and n_0 such that **f(n)≥C*g(n)** for all n, n≥0

Example:

3n+2= Ω (n) as

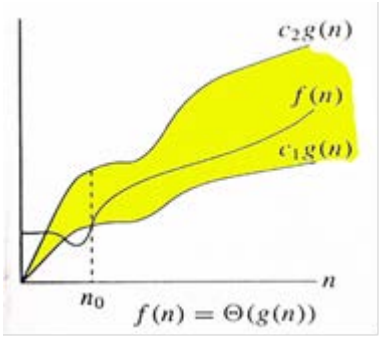
3n+2 ≥3n for all n≥1



Theta notation: θ

The function f(n)= θ (g(n)) (read as "f of n is theta of g of n") iff there exist positive constants c1, c2 and n_0 such that **C1*g(n) ≤f(n)≤C2*g(n)** for all n, n≥0

Example:
3n+2= **θ** (n) as
3n+2 ≥3n for all n≥2
3n+2 ≤3n for all n≥2
Here c1=3 and c2=4 and n_0=2



Little oh: o

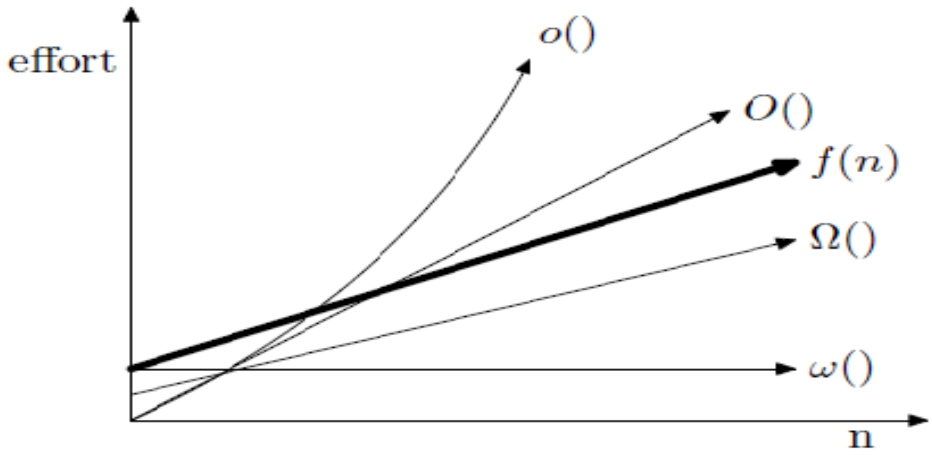
The function f(n)=o(g(n)) (read as "f of n is little oh of g of n") iff **Lim f(n)/g(n)=0**
n→∞
for all n, n≥0

Example:
3n+2= o(n^2) as **Lim ((3n+2)/n^2)=0**
n→∞

Little Omega: ω
The function f(n)= ω (g(n)) (read as "f of n is little omega of g of n") iff

Lim g(n)/f(n)=0 for all n, n≥0
n→∞
Example:
3n+2= o(n^2) as **Lim (n^2/(3n+2)) =0**
n→∞

Graph for visualize the relationships between these notations



1.5 PROBABILISTIC ANALYSIS

Probabilistic analysis of algorithms is an approach to estimate the computational complexity of an algorithm or a computational problem. It starts from an assumption about a probabilistic distribution of the set of all possible inputs. This assumption is then used to design an efficient algorithm or to derive the complexity of a known algorithm.

- We must know or make assumptions about the distribution of inputs.
- The expected cost is over this distribution.
- The analysis will give us ***average case*** running time.

We don't have this information for the Hiring Problem, but suppose we could assume that candidates come in random order. Then the analysis can be done by counting permutations:

- Each ordering of candidates (relative to some reference ordering such as a ranking of the candidates) is equally likely to be any of the n! permutations of the candidates.
- In how many do we hire once? twice? three times? ... $n-1$ times? n times?
- It depends on how many permutations have zero, one two ... $n-2$ or $n-1$ candidates that come before a better candidate.
- This is complicated!
- Instead, we can do this analysis with indicator variables (next section)

Probabilistic Analysis with Indicator Random Variables

Here we introduce technique for computing the expected value of a random variable, even when there is dependence between variables. Two informal definitions will get us started:

A **random variable** (e.g., X) is a variable that takes on any of a range of values according to a probability distribution.

The **expected value** of a random variable (e.g., E[*X*]) is the average value we would observe if we sampled the random variable repeatedly.

Indicator Random Variables

Given sample space *S* and event *A* in *S*, define the **indicator random variable**

$$I\{A\} = \begin{cases} 1 & \text{if } A \text{ occurs}, \\ 0 & \text{if } A \text{ does not occur} \end{cases}$$

We will see that indicator random variables simplify analysis by letting us work with the probability of the values of a random variable separately.

Lemma 1

For an event *A*, let $X_A = I\{A\}$. Then the expected value $\mathbf{E[X_A] = Pr\{A\}}$ (the probability of event *A*).

Proof: Let ¬*A* be the complement of *A*. Then

$$\begin{aligned} E[X_A] &= E[I\{A\}] \quad \text{(by definition)} \\ &= 1*Pr\{A\} + 0*Pr\{\neg A\} \quad \text{(definition of expected value)} \\ &= Pr\{A\}. \end{aligned}$$

Simple Example

What is the expected number of heads when flipping a fair coin once?

- Sample space *S* is {H, T}
- Pr{H} = Pr{T} = 1/2
- Define indicator random variable $X_H = I\{H\}$, which counts the number of heads in one flip.
- Since Pr{H} = 1/2, Lemma 1 says that $E[X_H] = 1/2$.

Less Simple Example

What is the expected number of heads when we flip a fair coin *n* times?

Let *X* be a random variable for the number of heads in *n* flips.

We could compute $E[X] = \sum_{i=0,n} i \Pr\{X=i\}$ -- that is, compute and add the probability of there being 0 heads total, 1 head total, 2 heads total ... n heads total.

Instead use indicator random variables to count something we *do* know the probability for: the probability of getting heads when flipping the coin once:

- For $i = 1, 2, \ldots n$ define X_i = I{the ith flip results in event H}.
- Then $X = \sum_{i=1,n} X_i$. *(That is, count the flips individually and add them up.)*
- Lemma 1 says that $E[X_i] = Pr\{H\} = 1/2$ for $i = 1, 2, \ldots n$.
- Expected number of heads is $E[X] = E[\sum_{i=1,n} X_i]$
- *Problem:* We don't have $\sum_{i=1,n} X_i$; we only have $E[X_1]$, $E[X_2]$, ... $E[X_n]$.
- *Solution:* **Linearity of expectation** (appendix C): ***expectation of sum equals sum of expectations.*** Therefore:

$$\begin{aligned} E[X] &= E\left[\sum_{i=1}^{n} X_i\right] \\ &= \sum_{i=1}^{n} E[X_i] \\ &= \sum_{i=1}^{n} 1/2 \\ &= n/2 . \end{aligned}$$

The key idea: if it's hard to count one way, use indicator random variables to count an easier way!

Hiring Problem and Cost

The book's example is a little strange but illustrates the points well. Suppose you are using an employment agency to hire an office assistant.

- The agency sends you one candidate per day: interview and decide.
- Cost to interview is c_i per candidate (fee to agency).
- Cost to hire is c_h per candidate (includes firing prior assistant and fee to agency).
- $c_h > c_i$
- You always hire the best candidate seen so far.

```
Hire-Assistant(n)
1  best = 0            // fictional least qualified candidate
2  for i = 1 to n
3     interview candidate i // paying cost  c_i
4     if candidate i is better than candidate best
```

5 best = i

6 hire candidate i // paying cost c_h

What is the cost of this strategy?

- If we interview n candidates and hire m of them, cost is $O(c_i n + c_h m)$
- We interview all n and c_i is small, so we focus on $c_h m$.
- $c_h m$ varies with each run and depends on interview order
- This is a common paradigm: finding the maximum or minimum in a sequence by examining each element, and changing the winner m times.

Best Case

If each candidate is worse than all who came before, we hire one candidate: $O(c_i n + c_h) = O(c_i n)$

Worst Case

If each candidate is better than all who came before, we hire all n ($m = n$): $O(c_i n + c_h n)=O(c_h n)$ since $c_h > c_i$

But this is pessimistic.

1.6 AMORTIZED ANALYSIS

In computer science, **amortized analysis** is a method for analyzing a given algorithm's time complexity, or how much of a resource, especially time or memory in the context of computer programs, it takes to execute. Unlike other analyses that focus on an algorithm's run time in a worst case scenario, known as asymptotic analysis, amortized analysis examines how an algorithm will perform in practice or on average.

Amortized analysis considers both the costly and less costly operations together over the whole series of operations of the algorithm. This may include accounting for different types of input, length of the input, and other factors that affect its performance.

Amortized analysis initially emerged from a method called aggregate analysis, which is now subsumed by amortized analysis. However, the technique was first formally introduced by Robert Tarjan in his 1985 paper Amortized Computational Complexity, which addressed the need for more useful form of analysis than the common probabilistic methods used. Amortization was initially used for very specific types of algorithms, particularly those involving binary trees and union operations. However, it is now ubiquitous and comes into play when analyzing many other algorithms as well.

There are generally three methods for performing amortized analysis: the aggregate method, the accounting method, and the potential method. All of these give the same answers, and their usage difference is primarily circumstantial and due to individual preference.

- Aggregate analysis determines the upper bound $T(n)$ on the total cost of a sequence of n operations, then calculates the amortized cost to be $T(n) / n$.
- The accounting method determines the individual cost of each operation, combining its immediate execution time and its influence on the running time of future operations. Usually, many short-running operations accumulate a "debt" of unfavorable state in small increments, while rare long-running operations decrease it drastically.
- The potential method is like the accounting method, but overcharges operations early to compensate for undercharges later.

Aggregate analysis

Stack operations

- PUSH(S, x): $O(1)$ each $\Rightarrow$ $O(n)$ for any sequence of n operations.
- POP(S): $O(1)$ each $\Rightarrow$ $O(n)$ for any sequence of n operations.

- MULTIPOP(S, k)
  while S is not empty and $k > 0$
  do POP(S)
  $k \leftarrow k - 1$

Running time of MULTIPOP:

- Linear in # of POP operations.
- Let each PUSH/POP cost 1.
- # of iterations of **while** loop is $\min(s, k)$, where $s =$ # of objects on stack.
- Therefore, total cost $= \min(s, k)$.

Sequence of n PUSH, POP, MULTIPOP operations:

- Worst-case cost of MULTIPOP is $O(n)$.
- Have n operations.
- Therefore, worst-case cost of sequence is $O(n^2)$.

Observation

- Each object can be popped only once per time that it's pushed.
- Have $\leq n$ PUSHes $\Rightarrow \leq n$ POPs, including those in MULTIPOP.
- Therefore, total cost $= O(n)$.
- Average over the n operations $\Rightarrow O(1)$ per operation on average.

Again, notice no probability.

- Showed *worst-case* $O(n)$ cost for sequence.
- Therefore, $O(1)$ per operation on average.

This technique is called ***aggregate analysis***.

Accounting method

Assign different charges to different operations.

- Some are charged more than actual cost.
- Some are charged less.

Amortized cost = amount we charge.

When amortized cost > actual cost, store the difference *on specific objects* in the data structure as ***credit***.

Use credit later to pay for operations whose actual cost > amortized cost.

Differs from aggregate analysis:

- In the accounting method, different operations can have different costs.
- In aggregate analysis, all operations have same cost.

Need credit to never go negative.

- Otherwise, have a sequence of operations for which the amortized cost is not an upper bound on actual cost.
- Amortized cost would tell us *nothing*.

Let c_i = actual cost of ith operation ,
$\widehat{c}_i$ = amortized cost of ith operation .

Then require $\sum_{i=1}^{n} \widehat{c}_i \geq \sum_{i=1}^{n} c_i$ for *all* sequences of n operations.

Total credit stored $= \sum_{i=1}^{n} \widehat{c}_i - \sum_{i=1}^{n} c_i \underbrace{\geq}_{\text{had better be}} 0$.

Stack

operation	actual cost	amortized cost
PUSH	1	2
POP	1	0
MULTIPOP	$\min(k, s)$	0

Intuition: When pushing an object, pay \$2.

- \$1 pays for the PUSH.
- \$1 is prepayment for it being popped by either POP or MULTIPOP.
- Since each object has \$1, which is credit, the credit can never go negative.
- Therefore, total amortized cost, $= O(n)$, is an upper bound on total actual cost.

Accounting method

Assign different charges to different operations.

- Some are charged more than actual cost.
- Some are charged less.

Amortized cost = amount we charge.

When amortized cost > actual cost, store the difference *on specific objects* in the data structure as ***credit***.

Use credit later to pay for operations whose actual cost > amortized cost.

Differs from aggregate analysis:

- In the accounting method, different operations can have different costs.
- In aggregate analysis, all operations have same cost.

Need credit to never go negative.

- Otherwise, have a sequence of operations for which the amortized cost is not an upper bound on actual cost.
- Amortized cost would tell us *nothing*.

Let c_i = actual cost of ith operation ,
$\widehat{c_i}$ = amortized cost of ith operation .

Then require $\sum_{i=1}^{n} \widehat{c_i} \geq \sum_{i=1}^{n} c_i$ for *all* sequences of n operations.

Total credit stored $= \sum_{i=1}^{n} \widehat{c_i} - \sum_{i=1}^{n} c_i \underbrace{\geq}_{\text{had better be}} 0$.

Stack

operation	actual cost	amortized cost
PUSH	1	2
POP	1	0
MULTIPOP	$\min(k, s)$	0

Intuition: When pushing an object, pay \$2.

- \$1 pays for the PUSH.
- \$1 is prepayment for it being popped by either POP or MULTIPOP.
- Since each object has \$1, which is credit, the credit can never go negative.
- Therefore, total amortized cost, $= O(n)$, is an upper bound on total actual cost.

1.7 DIVIDE AND CONQUER

It is a one of the process or design technique for solving big problems which requires large amount of input and produce heavy output.

GENERAL METHOD:

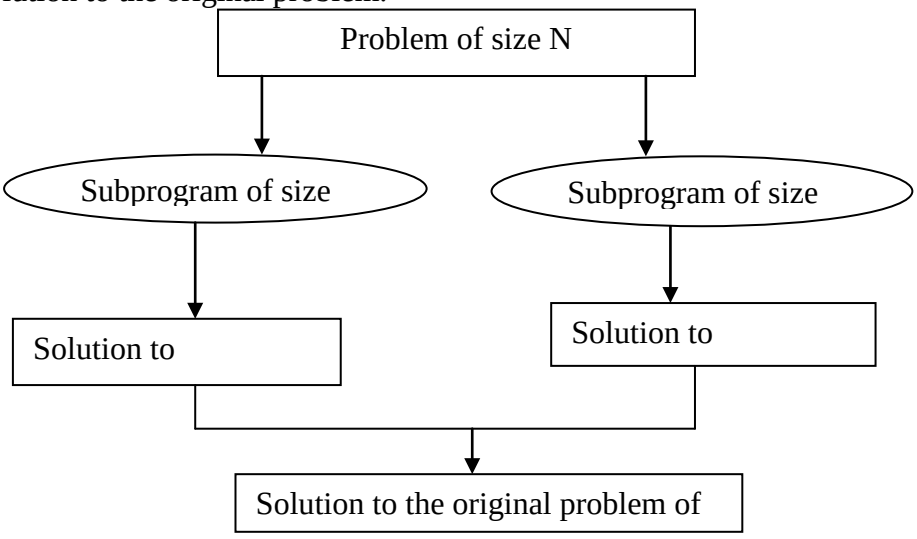
How the Divide and Conquer rule work:

In Divide and conquer rule

1st: A problem's instances are divided into several smaller instances of the same problem, in same size.

2nd: And then small instances are solved.

3rd: Then after, the solutions obtained for the smaller instances are combined to get a solution to the original problem.



Diagrammatic representation of Divide and conquer rule

Pseudo code Representation of Divide and conquer rule for problem "P"

Algorithm DAndC(P)
 	
{
if small(P) then return S(P)
 	
else{
 	
divide P into smaller instances P1,P2,P3…Pk;
 	
apply DAndC to each of these subprograms; // means DAndC(P1), DAndC(P2)….. DAndC(Pk)
 	
return combine(DAndC(P1), DAndC(P2)….. DAndC(Pk));

```
}

}
```

P➔Problem

Here small(P)➔ Boolean value function. If it is true, then the function S is invoked

Calculate time for DAndC:

If the problem P of size is n and K sub problems size is n1,n2,n3-----nk then

$$T(n) = g(n)$$
$$T(n1)+T(n2)+----+T(n_k)+f(n)$$

T(n)➔ Time for DAndC of any Input size n;
g(n)➔ Time to compute the answer directly for small inputs
T(n1)➔ Time for DAndC of small input size n1;
T(n2)➔ Time for DAndC of small input size n2;
*
*
$T(n_k)$➔ Time for DAndC of small input size nk;
f(n)➔ Time for dividing P and combining the solution of subproblems.

Time Complexity of DAndC algorithm:

$$T(n) = T(1) \text{ if } n=1$$
$$aT(n/b)+f(n) \text{ if } n>1$$

a,b➔ constants.
This is called the **general divide and-conquer recurrence.**

Advantages of DAndC:
The time spent on executing the problem using DAndC is smaller than other method.
This technique is ideally suited for parallel computation.
This approach provides an efficient algorithm in computer science.

Applications of Divide and Conquer rule or algorithm:

- Binary search,
- Quick sort,

- Merge sort,
- Strassen's matrix multiplication.

1.8 BINARY SEARCH OR HALF-INTERVAL SEARCH

1. This algorithm finds the position of a specified input value (the search "key") within an array sorted by key value.
2. In each step, the algorithm compares the search key value with the key value of the middle element of the array.
3. If the keys match, then a matching element has been found and its index, or position, is returned.
4. Otherwise, if the search key is less than the middle element's key, then the algorithm repeats its action on the sub-array to the **left** of the middle element or, if the search key is greater, then the algorithm repeats on sub array to the **right** of the middle element.
5. If the search element is less than the minimum position element or greater than the maximum position element then this algorithm returns not found.

Binary search algorithm by using recursive methodology:

```
Algorithm binary_search(A, key, low, high)
{
 if (high < low) then
return “array is empty”;
 if(key<low || key>high) then
return “element not in array list”
 else
  {
   mid = (low +high)/2;
    if (A[mid] > key) then
return binary_search(A, key, low, mid-1);
    else if (A[mid] < key) then
return binary_search(A, key, mid+1, high);
    else
    return mid;
  }
}
```

Time Complexity:

Data structure:- Array

For successful search	**Unsuccessful search**
Worst case→ O(log n) or θ(log n) Average case→ O(log n) or θ(log n) Best case→ O(1) or θ(1)	θ(log n):- for all cases.

Binary search algorithm by using iterative methodology:

1. Algorithm BinSearch(a,n,x)

```
2.  // Given an array a[1:n] of elements in non-decreasing
3.  //order, n>=0,determine whether 'x' is present and
4.  // if so, return 'j' such that x=a[j]; else return 0.
5.  {
6.  low:=1; high:=n;
7.  while (low<=high) do
8.  {
9.        mid:=[(low+high)/2];
10.       if (x<a[mid]) then high;
11.       else if(x>a[mid]) then
                low=mid+1;
12.    else return mid;
13.  }
14.   return 0;
15. }
```

- This BinSearch has 3 i/ps a,n, & x.
- The while loop continues processing as long as there are more elements left to check.
- At the conclusion of the procedure 0 is returned if x is not present, or 'j' is returned, such that a[j]=x.
- We observe that low & high are integer Variables such that each time through the loop either x is found or low is increased by at least one or high is decreased at least one.

- Thus we have 2 sequences of integers approaching each other and eventually low becomes > than high & causes termination in a finite no. of steps if 'x' is not present.

Example:

1) Let us select the 14 entries.
   -15,-6,0,7,9,23,54,82,101,112,125,131,142,151.

→ Place them in a[1:14], and simulate the steps Binsearch goes through as it searches for different values of 'x'.
→ Only the variables, low, high & mid need to be traced as we simulate the algorithm.
→ We try the following values for x: 151, -14 and 9.
for 2 successful searches &
1 unsuccessful search.

- Table. Shows the traces of Bin search on these 3 steps.

X=151	low	high	mid
	1	14	7
	8	14	11
	12	14	13
	14	14	14
			Found

x=-14	low	high	mid
	1	14	7
	1	6	3

	1	2	1
	2	2	2
	2	1	Not found

x=9	low	high	mid
	1	14	7
	1	6	3
	4	6	5
			Found

Theorem: Algorithm BinSearch(a,n,x) works correctly.

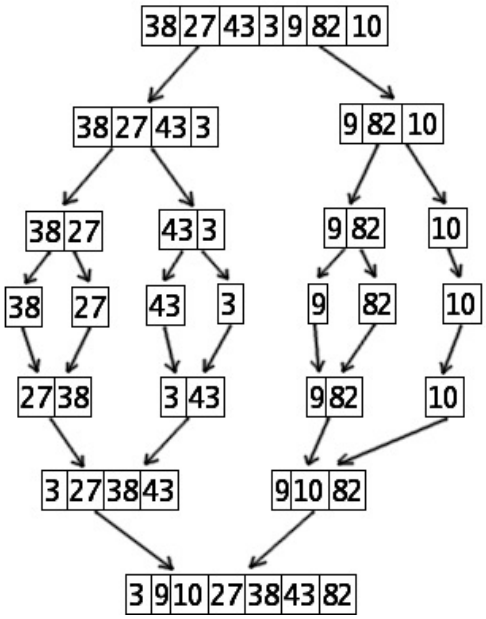
Proof:
We assume that all statements work as expected and that comparisons such as x>a[mid] are appropriately carried out.

- Initially low =1, high= n,n>=0, and a[1]<=a[2]<=……..<=a[n].
- If n=0, the while loop is not entered and is returned.

- Otherwise we observe that each time thro' the loop the possible elements to be checked of or equality with x and a[low], a[low+1],……..,a[mid],……a[high].
- If x=a[mid], then the algorithm terminates successfully.
- Otherwise, the range is narrowed by either increasing low to (mid+1) or decreasing high to (mid-1).
- Clearly, this narrowing of the range does not affect the outcome of the search.
- If low becomes > than high, then 'x' is not present & hence the loop is exited.

1.9 MERGE SORT

The merge sort splits the list to be sorted into two equal halves, and places them in separate arrays. This sorting method is an example of the DIVIDE-AND-CONQUER paradigm i.e. it breaks the data into two halves and then sorts the two half data sets recursively, and finally merges them to obtain the complete sorted list. The merge sort is a comparison sort and has an algorithmic complexity of O (n log n). Elementary implementations of the merge sort make use of two arrays - one for each half of the data set. The following image depicts the complete procedure of merge sort.



Advantages of Merge Sort:

1. Marginally faster than the heap sort for larger sets
2. Merge Sort always does lesser number of comparisons than Quick Sort. Worst case for merge sort does about 39% less comparisons against quick sort's average case.
3. Merge sort is often the best choice for sorting a linked list because the slow random-access performance of a linked list makes some other algorithms (such as quick sort) perform poorly, and others (such as heap sort) completely impossible.

Algorithm for Merge sort

Algorithm mergesort(low, high)
{
if(low<high) then
{
mid=(low+high)/2;
mergesort(low,mid);
mergesort(mid+1,high); //Solve the sub-problems
Merge(low,mid,high); // Combine the solution
}
}
void Merge(low, mid, high){

// Dividing Problem into Sub-problems and this "mid" is for finding where to split the set.

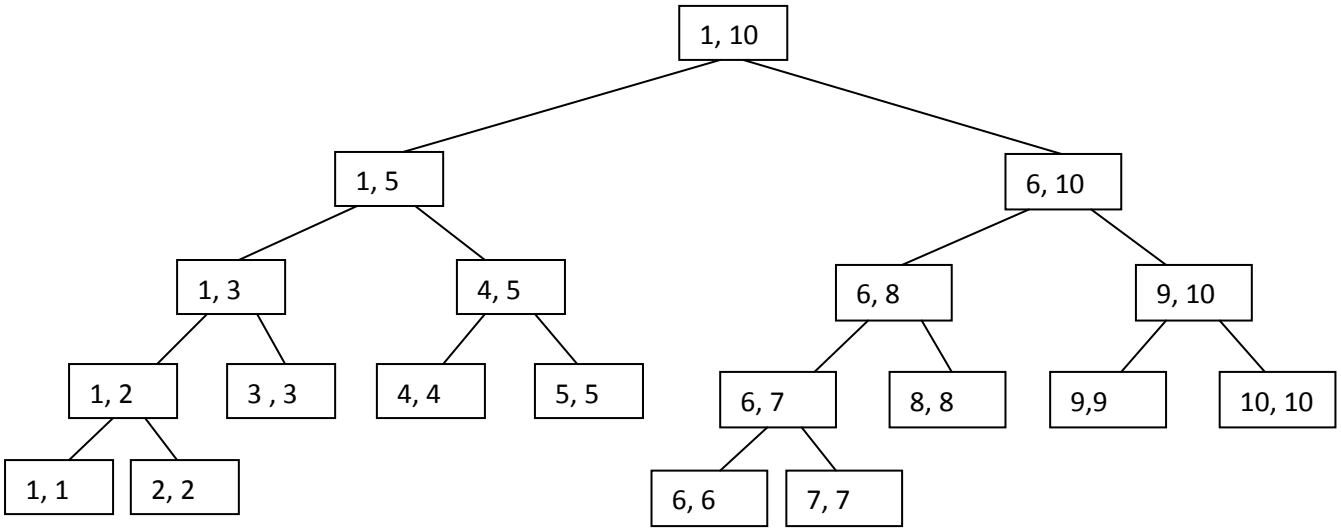
```
k=low;
i=low;
j=mid+1;
while(i<=mid && j<=high) do{
if(a[i]<=a[j]) then
{
temp[k]=a[i];
i++;
k++;
}
else
{
temp[k]=a[j];
j++;
k++;
}
}
while(i<=mid) do{
temp[k]=a[i];
i++;
k++;
}
while(j<=high) do{
temp[k]=a[j];
j++;
k++;
```

```
}
For k=low to high do
a[k]=temp[k];
}
For k:=low to high do a[k]=temp[k];
}
```

Tree call of Merge sort

Consider a example:

A[1:10]={310,285,179,652,351,423,861,254,450,520}



Tree call of Merge sort (1, 10)

Tree call of Merge Sort Represents the sequence of recursive calls that are produced by merge sort.

Computing Time for Merge sort:

The time for the merging operation in proportional to n, then computing time for merge sort is described by using recurrence relation.

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

Here c, a→Constants.

If n is power of 2, $n=2^k$

Form recurrence relation

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

$= 2[2T(n/4)+cn/2] + cn$

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

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

$= 2^3 T(n/8)+3cn$

$= 2^4 T(n/16)+4cn$

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

$= an+cn(\log n)$

By representing it by in the form of Asymptotic notation O is $T(n)=O(n\log n)$

1.10 QUICK SORT

Quick Sort: -

In the divide and conquer of quick sort we recursively divide the inputs into sub lists until the small (p) condition is occurred. The partitions in this are formed unlike the other problems involved in divide and conquer. In every execution of quick sort the partition location 'j' is determined where 'j' is the location of an element place in its sorted order. Then we form two partitions from 1st element of the partition to j-1 and j+1 to the lout element of the partition.

- **Steps in Partition algorithm**: -

Step - 1: - Initially we take 'p' pointing to the first element and q pointing to the last element i.e. p is the location of 1st element q is the location of last element.

Step - 2: - Initialize i:=p and j:=q+1

Step - 3: - Increment I until we find an element which is greater than the element present at location 'p'

Step - 4: - Decrement j by 1 until we find an element which is less than the element presser at location 'p'

Step - 5: - If i<j then swap (a(1), a(j))

Step - 6: - If i>j swap (a(i), a(p))

Step – 7: - If i>j then we get two partitions i.e. two sub lists from (p,j-1) and (j+1, q)

Step – 8: - Solve each sub problem recursively until p becomes greater than q

Algorithm Quick sort (p, q)

```
// sorts the elements a[p]……., a[q]
// which reside in global array a[1:n] in to 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 is more than one element
{               //divide p into sub problems
                        j:= partition (a,p,q+1);
                        //j is position of partitioning element
                        //solve the sub problems
                        quicksort (p,j-1);
                        quicksort (j+1,q)
                        //need not combine the solutions
 }
 }
```

Algorithm partition (a,m,p)

```
{
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;
}
```

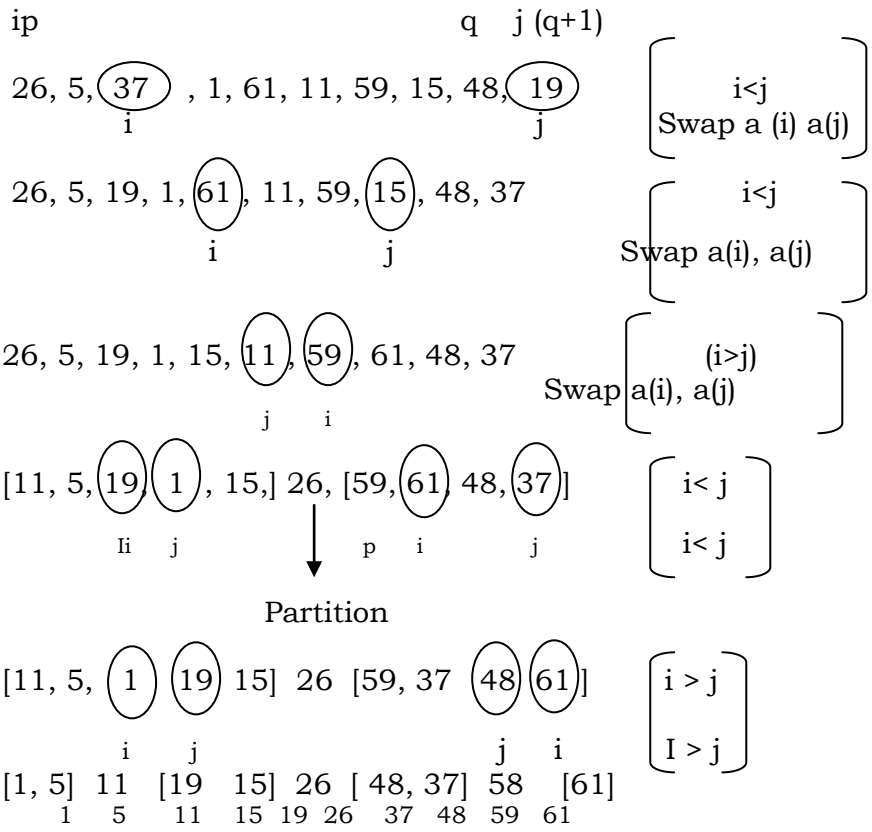
Algorithm Interchange (a,i,j)

```
//exchange a(i) with a (i)
{
  p: = a(i);
  a(i): = a(j);
  a(j): = p;
}
```

Example:

26, 5, 37, 1, 61, 11, 59, 15, 48, 19

1	2	3	4	5	6	7	8	9	10
26,	5,	37,	1,	61,	11,	59,	15,	48,	19



Analysis of Quicksort

In analyzing QUICKSORT, we shall count only the number of element comparisons C(n). It is easy to see that the frequency count of other operations is of the same order as C(n).

First, let us obtain the worst case value Cw(n) of C(n). The number of element comparisons in each call of PARTITION is at most p - m + 1. (Note that if the elements are not distinct then at most p - m + 2 comparisons may be made.) Let r be the total number of elements in all of the calls to PARTITION at any level of recursion. At level one only one call, PARTITION(1, n + 1) is made and r = n; at level two at most two calls are made and r = n - 1; etc. At each level of recursion, O(r) element comparisons are made by PARTITION. At each level r is at least one less than the r at the previous level as the partitioning elements of the previous level are eliminated. Hence Cw(n) is the sum on r as r varies from 2 to n or $O(n^2)$.

The average value $C_A(n)$ of $C(n)$ is much less than $C_w(n)$. Under the assumptions made earlier, the partitioning element v in the call to PARTITION(m, p) has an equal probability of being the ith smallest element $1 \le i \le p - m$, in $A(m:p-1)$. Hence the two subfiles remaining to be sorted will be $A(m:j)$ and $A(j+1:p-1)$ with probability $1/(p-m)$, $m \le j < p$. From this we obtain the recurrence

$$C_A(n) = n + 1 + \frac{1}{n} \sum_{1 \le k \le n} (C_A(k-1) + C_A(n-k)) \qquad (3.4)$$

$n + 1$ is the number of element comparisons required by PARTITION on its first call. Note that $C_A(0) = C_A(1) = 0$. Multiplying both sides of (3.4) by n we obtain

$$nC_A(n) = n(n+1) + 2(C_A(0) + C_A(1) + \cdots + C_A(n-1)) \qquad (3.5)$$

Replacing n by $n - 1$ in (3.5) gives

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

Subtracting this from (3.5) we get

$$nC_A(n) - (n-1)C_A(n-1) = 2n + 2C_A(n-1)$$

or

$$C_A(n)/(n+1) = C_A(n-1)/n + 2/(n+1)$$

Repeatedly using this equation to substitute for $C_A(n-1)$, $C_A(n-2)$, ... we get

$$\begin{aligned} C_A(n)/(n+1) &= C_A(n-2)/(n-1) + \frac{2}{n} + \frac{2}{n+1} \\ &= C_A(n-3)/(n-2) + \frac{2}{n-1} + \frac{2}{n} + \frac{2}{n+1} \\ &\vdots \\ &= C_A(1)/2 + 2 \sum_{3 \le k \le n+1} 1/k \\ &= 2 \sum_{3 \le k \le n+1} 1/k \qquad (3.6) \end{aligned}$$

Since

$$\sum_{3 \le k \le n+1}^{C_A(n)} 1/k \le \int_2^{n+1} 1/x\,dx = \log_e(n+1) - \log_e 2$$

(3.6) yields

$$C_A(n) \leqslant 2(n+1)[\log_e(n+2) - \log_e 2] = O(n \log n)$$

Name	Time Complexity			Space Complexity
	Best case	Average Case	Worst Case	
Bubble	O(n)	-	$O(n^2)$	O(n)
Insertion	O(n)	$O(n^2)$	$O(n^2)$	O(n)
Selection	$O(n^2)$	$O(n^2)$	$O(n^2)$	O(n)
Quick	O(log n)	O(n log n)	$O(n^2)$	O(n + log n)
Merge	O(n log n)	O(n log n)	O(n log n)	O(2n)
Heap	O(n log n)	O(n log n)	O(n log n)	O(n)

Comparison between Merge and Quick Sort:

- Both follows Divide and Conquer rule.
- Statistically both merge sort and quick sort have the same average case time i.e., O(n log n).
- Merge Sort Requires additional memory. The pros of merge sort are: it is a stable sort, and there is no worst case (means average case and worst case time complexity is same).
- Quick sort is often implemented in place thus saving the performance and memory by not creating extra storage space.
- But in Quick sort, the performance falls on already sorted/almost sorted list if the pivot is not randomized. Thus why the worst case time is $O(n^2)$.

1.11 STRASSEN'S MATRIX MULTIPLICAION

- Let A and B be the 2 n*n Matrix. The product matrix C=AB is calculated by using the formula,

  C (i ,j)= A(i,k) * B(k,j) for all 'i' and j between 1 and n.

- The time complexity for the matrix Multiplication is $O(n^3)$.

- Divide and conquer method suggest another way to compute the product of n*n matrix.

- We assume that N is a power of 2 .In the case N is not a power of 2 ,then enough rows and columns of zero can be added to both A and B .SO that the resulting dimension are the powers of two.

- If n=2 then the following formula as a computed using a matrix multiplication operation for the elements of A & B.

- If n>2,Then the elements are partitioned into sub matrix n/2*n/2..since 'n' is a power of 2 these product can be recursively computed using the same formula .This Algorithm will continue applying itself to smaller sub matrix until 'N" become suitable small(n=2) so that the product is computed directly .
- The formula are

$$\begin{pmatrix} A11 & A12 \\ A21 & A21 \end{pmatrix} * \begin{pmatrix} B11 & B12 \\ B21 & B22 \end{pmatrix} = \begin{pmatrix} C11 & C12 \\ C21 & C22 \end{pmatrix}$$

C11 = A11 B11 + A12 B21
C12 = A11 B12 + A12 B22
C21 = A21 B11 + A22 B21
C22 = A21 B12 + A22 B22

For EX:

$$4 * 4 = \begin{pmatrix} 2 & 2 & 2 & 2 \\ 2 & 2 & 2 & 2 \\ 2 & 2 & 2 & 2 \\ 2 & 2 & 2 & 2 \end{pmatrix} * \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix}$$

The Divide and conquer method

$$\left(\begin{array}{cc|cc} 2 & 2 & 2 & 2 \\ 2 & 2 & 2 & 2 \\ \hline 2 & 2 & 2 & 2 \\ 2 & 2 & 2 & 2 \end{array}\right) * \left(\begin{array}{cc|cc} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ \hline 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{array}\right) = \left(\begin{array}{cc|cc} 4 & 4 & 4 & 4 \\ 4 & 4 & 4 & 4 \\ \hline 4 & 4 & 4 & 4 \\ 4 & 4 & 4 & 4 \end{array}\right)$$

- To compute AB using the equation we need to perform 8 multiplication of n/2*n/2 matrix and from 4 addition of n/2*n/2 matrix.
- Ci,j are computed using the formula in equation →4
- As can be sum P, Q, R, S, T, U, and V can be computed using 7 Matrix multiplication and 10 addition or subtraction.
- The Cij are required addition 8 addition or subtraction.

$$T(n)=\begin{cases} b & n<=2 \text{ a \&b are} \\ 7T(n/2)+an^2 & n>2 \text{ constant} \end{cases}$$

Finally we get T(n) =O($n^{2.7}$)

Example

$$\begin{vmatrix} 4 & 4 \\ 4 & 4 \end{vmatrix} * \begin{vmatrix} 4 & 4 \\ 4 & 4 \end{vmatrix}$$

P=(4*4)+(4+4)=64
Q=(4+4)4=32
R=4(4-4)=0
S=4(4-4)=0
T=(4+4)4=32
U=(4-4)(4+4)=0
V=(4-4)(4+4)=0
C11=(64+0-32+0)=32
C12=0+32=32
C21=32+0=32
C22=64+0-32+0=32

So the answer c(i,j) is $\begin{vmatrix} 32 & 32 \\ 32 & 32 \end{vmatrix}$

since n/2n/2 &matrix can be can be added in Cn for some constant C, The overall computing time T(n) of the resulting divide and conquer algorithm is given by the sequence.

$$T(n)=\begin{cases} b & n<=2 \text{ a \&b are} \\ 8T(n/2)+cn^2 & n>2 \text{ constant} \end{cases}$$

That is T(n)=O(n^3)

* Matrix multiplication are more expensive then the matrix addition O(n^3).We can attempt to reformulate the equation for Cij so as to have fewer multiplication and possibly more addition .

- Strassen has discovered a way to compute the Cij of equation (2) using only 7 multiplication and 18 addition or subtraction.
- Strassen's formula are

P= (A11+A12)(B11+B22)
Q= (A12+A22)B11
R= A11(B12-B22)
S= A22(B21-B11)
T= (A11+A12)B22
U= (A21-A11)(B11+B12)
V= (A12-A22)(B21+B22)

C11=P+S-T+V
C!2=R+T
C21=Q+T
C22=P+R-Q+V

Previous papers Questions:

1. (a) Define an algorithm. What are the different criteria that satisfy the algorithm?
   (b) Explain how algorithms performance is analysed? Describe asymptotic notation?

2. (a) What are the different techniques to represent an algorithm? Explain.

   (b) Give an algorithm to solve the towers of Hanoi problem.

3. (a) Write an algorithm to find the sum of individual digits of a given number.

   (b) Explain the different looping statements used in pseudo code conventions

4. (a) What is meant by recursion? Explain with example, the direct and indirect recursive algorithms.

   (b) List the advantages of pseudo code convention over flow charts.

5. (a) Write an algorithm for Fibonacci series and discuss time complexity.

   (b) Write about the two methods to calculate time complexity with examples

6. What is meant by time complexity? Define different time complexity notations? Define example for one of each?

7. Write algorithm for Merge sort, Quick Sort, Selection sorting Algorithms?

8. (a) Draw the binary decision tree for the following set
   (3, 6, 9, 12, 15, 18, 21, 24, 27, 30, 33, 36, 39, 42, 45, 47)
   (b) Derive the time complexity for Quick Sort

9. (a) Draw the tree of calls of merge sort for the following set.
   (35, 25,15,10,45, 75, 85, 65, 55, 5, 20, 18)
   (b) Compare Quick sort algorithm performance from insertion sort algorithm.

10. (a) Discuss briefly about the randomized quick sort.
    (b) Draw the tree of calls of merge for the following set of elements
    (20, 30, 10, 40, 5, 60, 90, 45, 35, 25, 15, 55)

11. (a) Write an algorithm for quick sort by using recursive method.

(b) Explain Strassen is matrix multiplication algorithm with an example.