UNIT – I

Introduction: Algorithm Definition, Pseudocode Conventions, Space complexity and Time complexity, Asymptotic notations.
Divide and conquer: General method, Binary search, Merge sort, Quick sort, Strassen"s matrix multiplication.

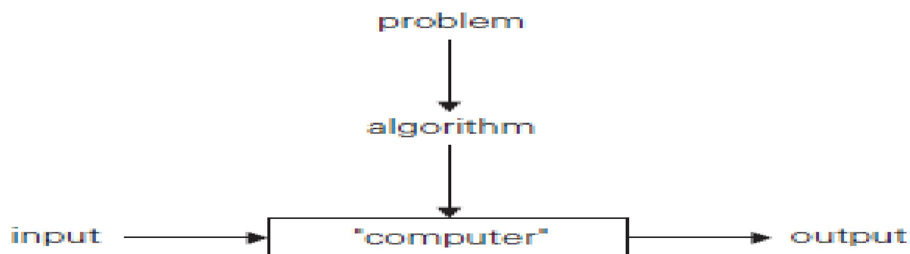
1. WHAT IS AN ALGORITHM:

Informal Definition:

An Algorithm is any well-defined computational procedure that takes some value or set of values as Input and produces a set of values or some value as output. Thus algorithm is a sequence of computational steps that transforms the i/p into the o/p.

Formal Definition:

An Algorithm is a finite set of instructions that, if followed, accomplishes aparticular task. In addition, all algorithms should satisfy the following criteria.



1. **INPUT** :Zero or more quantities are externally supplied.
   Example:
   Zero input: printf("welcome to c");
   More inputs: for(i=0;i<n;i++)

2. **OUTPUT**: At least one quantity is produced.
   i.e one or more outputs are produced

3. **DEFINITENESS:** Each instruction is clear and unambiguous.
   Example:add 6 to x -> clear
   Add 6 or 7 to x ->not clear

4. **FINITENESS**: If we trace out the instructions of an algorithm, then for all cases,the algorithm terminates after a finite number of steps.

5. **EFFECTIVENESS:** a finite amount of time to solve the problem.so that
   Every instruction must very basic so that it can be carriedout, in principle, by a person using only pencil & paper.

Issues or study of Algorithm:

1. How to device or design an algorithm creating and algorithm.
2. How to express an algorithm definiteness.
3. How to analysis an algorithm time and space complexity.
4. How to validate an algorithm fitness.

5. Testing the algorithm checking for error.

Advantages of algorithm:

1. easy to understand
2. Step by step representation to a given problem

1.2. ALGORITHM SPECIFICATIONS:Pseudocode Conventions

Algorithm can be described in three ways.

1. Natural language like English:

When this way is chooses care should be taken, we should ensure thateach & every statement is definite.

2. Graphic representation called flowchart:

This method will work well when the algorithm is small& simple.

3. Pseudocode:

- Algorithm can be represented in Text mode and Graphic mode
- Graphical representation is called Flowchart
- Text mode most often represented in close to any

High level language such as C,Pascal Pseudocode

- Pseudocode: High-level description of an algorithm.
- More structured than plain English.
- Less detailed than a program.
- Preferred notation for describing algorithms.
- Hides program design issues.

1.Pseudo-code Method:

Definition:Pseudocode is a high level informal description of operating principle of an alhorithm to perform a perticular task

In this method, we should typically describealgorithms as pro**gram, which resembles language like Pascal & algol.**

1.2.1. Pseudo-Code Conventions:

1. Comments begin with // and continue until the end of line.

Example a)single line comments://Addition of two numbers

b)Multi line comments: /*----------
-----------*\

2. Blocks are indicated with matching braces {and}.

Example:

```
Procedure()
{
Statement1;
Statement2;
      .
      .
      .
Statementn;
   }
```

3. An identifier begins with a letter. The data types of variables are not explicitly declared.

4. Compound data types can be formed with records. Here is an example,Node.

```
Record
{
   data type – 1        data-1;
          .
          .
          .
  data type – n      data – n;
  node * link;
}
```

Here link is a pointer to the record type node. Individual data itemsof a record can be accessed with and period.

5. Assignment of values to variables is done using the assignment statement.

<Variable>:= <expression>;
Example:num:=7;

6. There are two Boolean values TRUE and FALSE.
In order to produce these values

Logical Operators AND, OR, NOT
Relational Operators <, <=,>,>=, =, != are provided

7. The following looping statements are employed.

For, while and repeat-until

While Loop:

```
While < condition > do
{
        <statement-1>
              .
              .
              .
        <statement-n>
}
```

For Loop:

```
For variable: = value-1 to value-2 step step do
    {
      <statement-1>
           .
           .
           .
        <statement-n>
    }
```

repeat-until:

```
repeat
        <statement-1>
              .
              .
              .
        <statement-n>
until<condition>
```

8. A conditional statement has the following forms.

If <condition> then <statement>

If<condition> then <statement-1>Else <statement-1>

Case statement:

Case
{

}

: <condition-1> : <statement-1>
.
.
.
: <condition-n> : <statement-n>
: else : <statement-n+1>

9. Input and output are done using the instructions read & write.

10. There is only one type of procedure:
Algorithm, the heading takes the form,

Algorithm Name (Parameter lists)

As an example, the following algorithm fields & returns the maximum of'n' given numbers:

```
1.  algorithm Max(A,n)
2.  // A is an array of size n.
3 . {
4.  Result := A[1];
5.  for I:= 2 to n do
6. if A[I] > Result then
7. Result :=A[I];
8. return Result;
9 }
```

In this algorithm (named Max), A & n are procedure parameters.
Result & I are Local variables.

Next we present 2 examples to illustrate the process of translation problem into an algorithm.

1.3.PERFORMANCE ANALYSIS:

Space Complexity:

The space complexity of an algorithm is the amount of money itneeds to run to compilation.

Time Complexity:

The time complexity of an algorithm is the amount of computertime it needs to run to compilation.

1.3.1. Space Complexity:

Space Complexity Example:

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

The Space needed by each of these algorithms is seen to be the sum of the following component.

1.A fixed part that is independent of the characteristics (eg:number,size)of the inputs and outputs.

The part typically includes the instruction space (ie. Space for the code), space for simple variable and fixed-size component variables (also called aggregate) space for constants, and so on.

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 (to the extent that is depends oninstance characteristics), and the recursion stack space.

The space requirement s(p) of any algorithm p may therefore be written as,

S(P) = c+ Sp(Instance characteristics)

Where 'c' is a constant.

Example 2:

```
Algorithm sum(a,n)
{
    s=0.0;
    for I=1 to n do
    s= s+a[I];
    return s;
}
 S= 1 unit
 I=1 unit
 n=1 unit
 a=n units
 -----------
```

Time complexity =3+n   i.e O(n)

1. The problem instances for this algorithm are characterized by n,the number of elements to be summed. The space needed d by ‘n’ is one word, since it is of type integer.
2. The space needed by ‘a’ a is the space needed by variables of tyepe array of floating point numbers.
3. This is atleast ‘n’ words, since ‘a’ must be large enough to hold the ‘n’ elements to be summed.

4. So,we obtain Ssum(n)>=(n+s)
   [n for a[],one each for n,I a& s]

1.3.2.Time Complexity:

The time T(p) taken by a program P is the sum of the compile time and therun time(execution time)

The compile time does not depend on the instance characteristics. Also we mayassume that a compiled program will be run several times without recompilation .This rum time is denoted by tp(instance characteristics).

The number of steps any problem statement is assigned depends on the kind of statement.

For example, comments 0 steps.
Assignment statements 1 steps.
[Which does not involve any calls to other algorithms]

Interactive statement such as for, while & repeat-until Control part of the statement.

We introduce a variable, count into the program statement to increment count with initial value 0.Statement to increment count by the appropriate amount are introduced into the program.

This is done so that each time a statement in the original program is executes count is incremented by the step count of that statement.

Algorithm:

```
Algorithm sum(a,n)
{
        s= 0.0;
        count = count+1;
        for I=1 to n do
        {
         count =count+1;
        s=s+a[I];
        count=count+1;
        }
        count=count+1;
        count=count+1;
        return s;
        }
```

If the count is zero to start with, then it will be 2n+3 on termination.

2. The second method to determine the step count of an algorithm is to build a table in which we list the total number of steps contributes by each statement.

First determine the number of steps per execution (s/e) of the statement and the total number of times (ie., frequency) each statement is executed.

By combining these two quantities, the total contribution of allstatements, the step count for the entire algorithm is obtained.

Statement	S/e	Frequency	Total
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. s=s+a[I];	1	n	n
6. return s;	1	1	1
7. }	0	-	0
Total			2n+3

Complexity of Algorithms Analysis

Best case:

This analysis constrains on the input, other than size. Resulting in the fasters possible run time

Worst case:

This analysis constrains on the input, other than size. Resulting in the fasters possible run time

Average case:type of input. Complexity:

This type of analysis results in average running time over every

Complexity refers to the rate at which the storage time grows as a function of the problem size

Asymptotic analysis:

Expressing the complexity in term of its relationship to know function. This type analysis is called asymptotic analysis.

1.3 ASYMPTOTIC NOTATION

- Formal way notation to speak about functions and classify them

The following notations are commonly use notations in performance analysis and used to characterize the complexity of an algorithm:

1. Big–Oh (O) ,
2. Big–Omega (Ω),
3. Big–Theta(Θ)
4. Little–OH (o) and
5. Little omega(w)

It is a way to compare "sizes" of functions:

· O≈ ≤   Ω≈ ≥   Θ ≈ =   o ≈ <   ω ≈ >

Time complexity	Name	Example
O(1)	Constant	Adding an element to the front of a linked list
O(logn)	Logarithmic	Finding an element in a sorted array
O (n)	Linear	Finding an element in an unsorted array
O(nlog n)	Linear	Logarithmic Sorting n items by 'divide-and-conquer'-Mergesort
$O(n^2)$	Quadratic	Shortest path between two nodes in a graph
$O(n^3)$	Cubic	Matrix Multiplication
$O(2^n)$	Exponential	The Towers of Hanoi problem

.

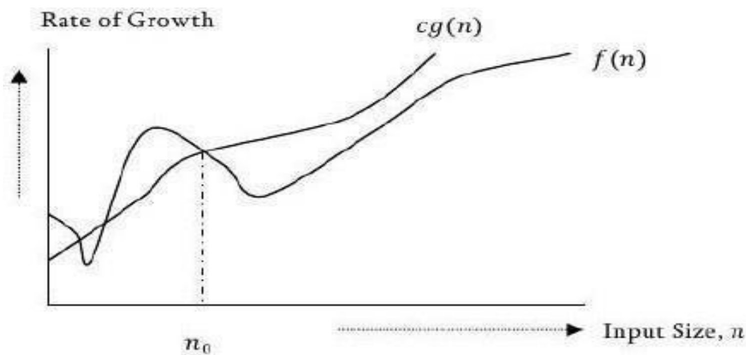
Big 'oh': the function f(n)=O(g(n)) iff there exist positive constants c and no suchthat f(n)≤c*g(n) for all n, n ≥ no.

Big-O Notation

This notation gives the tight upper bound of the given function. Generally we represent it as f(n) = O(g (11)). That means, at larger values of n, the upper bound of f(n) is g(n). For example, if $f(n) = n^4 + 100n^2 + 10n + 50$ is the given algorithm, then n^4 is g(n). That means g(n) gives the maximum rate of growth for f(n) at larger values of n.

O —notation defined as O(g(n)) = {f(n): there exist positive constants c and n_0 such that 0 <= f(n) <= cg(n) for all n >= n_0}. g(n) is an asymptotic tight upper bound for f(n). Our objective is to give some rate of growth g(n) which is greater than given algorithms rate of growth f(n).

In general, we do not consider lower values of n. That means the rate of growth at lower values of n is not important. In the below figure, n_0 is the point from which we consider the rate of growths for a given algorithm. Below n_0 the rate of growths may be different.



Note Analyze the algorithms at larger values of n only What this means is, below no we do not care for rates of growth.

Omega: the function f(n)=Ω(g(n)) iff there exist positive constants c and no suchthat f(n) ≥ c*g(n) for all n, n ≥ no.

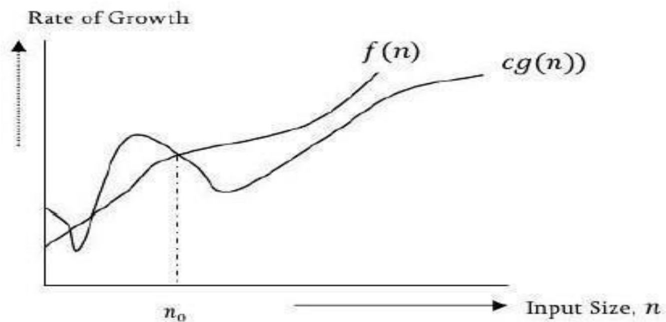
<u>Omega— Ω notation</u>

Omega: the function f(n)=Ω(g(n)) iff there exist positive constants c and no suchthat f(n) ≥ c*g(n) for all n, n ≥ no.

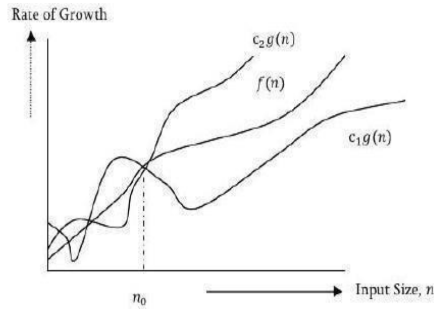
Similar to above discussion, this notation gives the tighter lower bound of the given algorithm and we represent it as f(n) = Ω (g(n)). That means, at larger values of n, the tighter lower bound of f(n) is g

For example, if $f(n) = 100n^2 + 10n + 50$, g(n) is $\Omega(n^2)$.

The . Ω. notation as be defined as Ω (g (n)) = {f(n): there exist positive constants c and n_0 such that 0 <= cg (n) <= f(n) for all n >= n_0}. g(n) is an asymptotic lower bound for f(n). Ω (g (n)) is the set of functions with smaller or same order of growth as f(n)



THETA-NOTATION



This notation decides whether the upper and lower bounds of a given function are same or not. The average running time of algorithm is always between lower bound and upper bound.

Theta: the function f(n)=ɵ(g(n)) iff there exist positive constants c1,c2 and no such that c1 g(n) ≤ f(n) ≤ c2 g(n) for all n, n ≥ no.

If the upper bound (O) and lower bound (Ω) gives the same result then Θ notation will also have the same rate of growth. As an example, let us assume that f(n) = 10n + n is the expression. Then, its tight upper bound g(n) is O(n). The rate of growth in best case is g (n) =
0(n). In this case, rate of growths in best case and worst are same. As a result, the average case will also be same.

None: For a given function (algorithm), if the rate of growths (bounds) for O and Ω are not same then the rate of growth Θ case may not be same.

Little Oh Notation:

The little Oh is denoted as o. It is defined as : Let, f(n} and g(n} be the non negative functions then

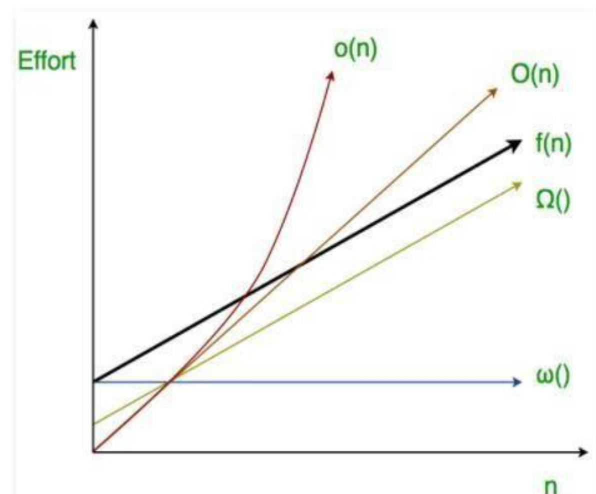
Little o asymptotic notation

Big-O is used as a tight upper-bound on the growth of an algorithm's effort (this effort is described by the function f(n)), even though, as written, it can also be a loose upper-bound. "Little-o" (o()) notation is used to describe an upper-bound that cannot be tight.

Definition : Let f(n) and g(n) be functions that map positive integers to positive real numbers. We say that f(n) is o(g(n)) (or f(n) E o(g(n))) if for **any real** constant c > 0, there exists an integer constant n0 ≥ 1 such that 0 ≤ f(n) < c*g(n).

Thus, little o() means **loose upper-bound** of f(n). Little o is a rough estimate of the maximum order of growth whereas Big-O may be the actual order of growth.

In mathematical relation,
f(n) = o(g(n)) means
lim f(n)/g(n) = 0
n→∞



1.3.3. Recursion:

Recursion may have the following definitions:
-The nested repetition of identical algorithm is recursion.
-It is a technique of defining an object/process by itself.
-Recursion is a process by which a function calls itself repeatedly until some specified condition has been satisfied.

When to use recursion:

Recursion can be used for repetitive computations in which each action is stated in terms of previous result. There are two conditions that must be satisfied by any recursive procedure.

1 .Each time a function calls itself it should get nearer to the solution.
2 .There must be a decision criterion for stopping the process.

In making the decision about whether to write an algorithm in recursive or non- recursive form, it is always advisable to consider a tree structure for the problem. If the structure is simple then use non- recursive form. If the tree appears quite bushy, with little duplication of tasks, then recursion is suitable.

The recursion algorithm for finding the factorial of a number is given below,

Algorithm : factorial-recursion
Input : n, the number whose factorial is to be found.Output : f,
the factorial of n
Method : if(n=0)
f=1
else
f=factorial(n-1) * n
if end
algorithm ends.

The general procedure for any recursive algorithm is as follows,

1. Save the parameters, local variables and return addresses.
2. If the termination criterion is reached perform final computation andgoto step 3 otherwise perform final computations and goto step 1

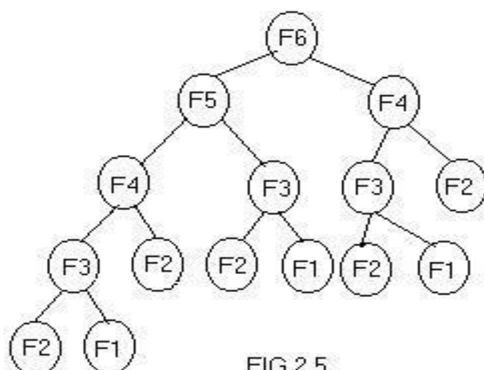


FIG 2.5

3. **Restore the most recently saved parameters, local variable and return address and goto the latest return address.**

<u>Iteration v/s Recursion</u>:

Demerits of recursive algorithms:

1. Many programming languages do not support recursion; hence, recursive mathematical function is implemented using iterative methods.

2. Even though mathematical functions can be easily implemented using recursion it is always at the cost of execution time and memory space. For example, the recursion tree for generating 6 numbers in a Fibonacci series generation is given in fig 2.5. A Fibonacci series is of the form 0,1,1,2,3,5,8,13,…etc, where the third number is the sum of preceding two numbers and so on. It can be noticed from the fig 2.5 that, f(n-2) is computed twice, f(n-3) is computed thrice, f(n-4) is computed 5 times.

3. A recursive procedure can be called from within or outside itself and to ensure its proper functioning it has to save in some order the return addresses so that, a return to the proper location will result when the return to a calling statement is made.
4. The recursive programs needs considerably more storage and will take more time.

Demerits of iterative methods :

1. Mathematical functions such as factorial and Fibonacci series generationcan be easily implemented using recursion than iteration.
2. In iterative techniques looping of statement is very much necessary. Recursion

is a top down approach to problem solving. It divides the problem intopieces or selects out one key step, postponing the rest.

Iteration is more of a bottom up approach. It begins with what is known and fromthis constructs the solution step by step. The iterative function

obviously uses time that is O(n) where as recursive function has an exponentialtime complexity.

It is always true that recursion can be replaced by iteration and stacks. It is alsotrue that stack can be replaced by a recursive program with no stack.

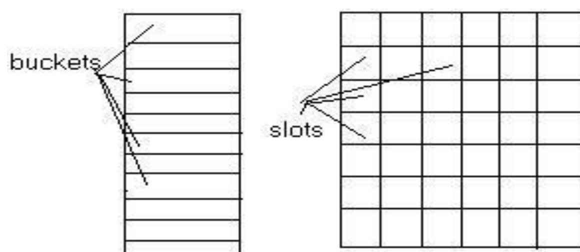


Fig 2.6

1.4. GENERAL METHOD:

Given a function to compute on ‘n’ inputs the divide-and-conquer strategy suggests splitting the inputs into ‘k’ distinct subsets, $1<k<=n$, yielding ‘k’ sub problems.

These sub problems must be solved, and then a method must be found to combine sub solutions into a solution of the whole.

If the sub problems are still relatively large, then the divide-and-conquer strategy can possibly be reapplied.

Often the sub problems resulting from a divide-and-conquer design are of the same type as the original problem.

For those cases the re application of the divide-and-conquer principle is naturally expressed by a recursive algorithm.

D And C(Algorithm) is initially invoked as D and C(P), where ‘p’ is the problem to be solved.

Small(P) is a Boolean-valued function that determines whether the i/p sizeis small enough that the answer can be computed without splitting.

If this so, the function ‘S’ is invoked.

Otherwise, the problem P is divided into smaller sub problems.

These sub problems P1, P2 …Pk are solved by recursive application of D And C.

Combine is a function that determines the solution to p using the solutionsto the ‘k’ sub problems.

If the size of ‘p’ is n and the sizes of the ‘k’ sub problems are n1,n2 ….nk,respectively, then the computing time of D And C is described by the recurrencerelation.T(n)= { g(n)

T(n1)+T(n2)+…............ +T(nk)+f(n);

n smallotherwise.

there T(n) is the time for D And C on any I/p of size ‘n’.

g(n) is the time of compute the answer directly for small I/ps.

f(n) is the time for dividing P & combining the solution tosub problems.

1) Algorithm D And C(P)

2) {
3) if small(P) then return S(P);
4) else
5) {
6) divide P into smaller instances
P1, P2… Pk, k>=1;
7) Apply D And C to each of these sub problems;
8) return combine (D And C(P1), D And C(P2),…….,D And C(Pk));
9) }
10) }

The complexity of many divide-and-conquer algorithms is given by recurrences of the form

$$T(n) = \begin{cases} T(1) & n=1 \\ AT(n/b)+f(n) & n>1 \end{cases}$$

Where a & b are known constants.

We assume that T(1) is known & 'n' is a power of b(i.e., n=b^k)

One of the methods for solving any such recurrence relation is called the substitution method.

This method repeatedly makes substitution for each occurrence of the function. T is the Right-hand side until all such occurrences disappear.

Example:

11) Consider the case in which a=2 and b=2. Let T(1)=2 & f(n)=n. Wehave,

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

= 4T(n/4)+2n
= 4[2T(n/4/2)+n/4]+2n
= 4[2T(n/8)+n/4]+2n
= 8T(n/8)+n+2n
= 8T(n/8)+3n
*
*
*

In general, we see that T(n)=2^iT(n/2^i)+in., for any log n >=I>=1. T(n)

=2^log n T(n/2^log n) + n log n

Corresponding to the choice of i=logn

Thus, T(n) = 2^log n T(n/2^log n) + n log n

= n. T(n/n) + n log n
= n. T(1) + n log n [since, log 1=0, 2^0=1] = 2n + n log n

1.5 BINARY SEARCH:

Applications of Divide and conquer rule or algorithm:

- Binary search,
- Quick sort,
- Merge sort,
- Strassen's matrix multiplication.

Binary search or Half-interval search algorithm or logarthimic search or binary chop

```
1.    Algorithm Bin search(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.    }
```

PROCEDURE STEPS:

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.

1.5. MERGE SORT

As another example divide-and-conquer, we investigate a sorting algorithmthat

has the nice property that is the worst case its complexity is O(n log n)

This algorithm is called merge sort

We assume throughout that the elements are to be sorted in non-decreasing order.

Given a sequence of ‘n’ elements a[1],…,a[n] the general idea is to imaginethen split into 2 sets a[1],…..,a[n/2] and a[[n/2]+1],….a[n].

Each set is individually sorted, and the resulting sorted sequences aremerged to produce a single sorted sequence of ‘n’ elements.

Thus, we have another ideal example of the divide-and-conquer strategy inwhich the splitting is into 2 equal-sized sets & the combining operation is the merging of 2 sorted sets into one.

Algorithm For Merge Sort:

```
1.      Algorithm MergeSort(low,high)
2.      //a[low:high] is a global array to be sorted
3.      //Small(P) is true if there is only one element
4.      //to sort. In this case the list is already sorted.5.
        {
6.      if (low<high) then //if there are more than one element7.
        {
8.      //Divide P into subproblems
9.      //find where to split the set
10.     mid = [(low+high)/2];
11.     //solve the subproblems.
12.     mergesort (low,mid);
13.     mergesort(mid+1,high);
14.     //combine the solutions .
15.     merge(low,mid,high);}}
```

Algorithm: Merging 2 sorted subarrays using auxiliary storage.

```
1.      Algorithm merge(low,mid,high)
2.      //a[low:high] is a global array containing
3.      //two sorted subsets in a[low:mid]
4.      //and in a[mid+1:high].The goal is to merge these 2 sets into
5.      //a single set residing in a[low:high].b[] is an auxiliary global array.6.   {
7.      h:=low; I:=low; j:=mid+1;
8.      while ((h<=mid) and (j<=high)) do
9.      {
10.     if (a[h]<=a[j]) then
11.     {
12.        b[I]:=a[h];
13.        h:= h+1;
14.     }
15.     else
16.     {
17.        b[I]:= a[j];
18.         j:=j+1;
```

```
19.          }
20.          :I=I+1;
21.          }
22.          if (h>mid) then
23.             for k:=j to high do
24.              {
25.                   b[I]:=a[k];
26.                   I:=I+1;
27.               }
28.           else
29.               for k:=h to mid do
30.               {
31.                    b[I]:=a[k];
32.                    :I=I+1;
33.                }
34.               for k:=low to high do a[k]:= b[k];
35.          }
```

Consider the array of 10 elements a[1:10] =(310, 285, 179, 652, 351, 423, 861, 254, 450, 520)

Algorithm Mergesort begins by splitting a[] into 2 sub arrays each of sizefive (a[1:5] and a[6:10]).

The elements in a[1:5] are then split into 2 sub arrays of size 3 (a[1:3]) and 2(a[4:5])

Then the items in a a[1:3] are split into sub arrays of size 2 a[1:2] & one(a[3:3])

The 2 values in a[1:2} are split to find time into one-element sub arrays, andnow the merging begins.

(310| 285| 179| 652, 351| 423, 861, 254, 450, 520)

Where vertical bars indicate the boundaries of sub arrays.

Elements a[I] and a[2] are merged to yield,

(285, 310|179|652, 351| 423, 861, 254, 450, 520)

Then a[3] is merged with a[1:2] and

(179, 285, 310| 652, 351| 423, 861, 254, 450, 520)

Next, elements a[4] & a[5] are merged.

(179, 285, 310| 351, 652 | 423, 861, 254, 450, 520)

And then a[1:3] & a[4:5]

(179, 285, 310, 351, 652| 423, 861, 254, 450, 520)

Repeated recursive calls are invoked producing the following sub arrays.(179, 285, 310, 351, 652| 423| 861| 254| 450, 520)

Elements a[6] &a[7] are merged.

Then a[8] is merged with a[6:7]
(179, 285, 310, 351, 652| 254,423, 861| 450, 520)

Next a[9] &a[10] are merged, and then a[6:8] & a[9:10] (179,
285, 310, 351, 652| 254, 423, 450, 520, 861)

At this point there are 2 sorted sub arrays & the final merge producesthe
fully sorted result.
(179, 254, 285, 310, 351, 423, 450, 520, 652, 861)

IF THE TIME FOR THE MERGING OPERATIONS IS PROPORTIONAL TO 'N',THEN THE COMPUTING TIME FOR MERGE SORT IS DESCRIBED BY THE RECURRENCE RELATION.

T(N)={A N=1,'A' A

CONSTANT2T(N/2)+CN N>1,'C' A

CONSTANT.

When 'n' is a power of 2, n= 2^k, we can solve this equation by successive substitution.

T(n) =2(2T(n/4) +cn/2) +cn
= 4T(n/4)+2cn
= 4(2T(n/8)+cn/4)+2cn
*
*
= 2^k T(1)+kCn.
= an + cn log n.

It is easy to see that if s^k<n<=2^k+1, then T(n)<=T(2^k+1). Therefore,

T(n)=O(n log n)

QUICK SORT

The divide-and-conquer approach can be used to arrive at an efficientsorting method different from merge sort.

In merge sort, the file a[1:n] was divided at its midpoint into sub arrays which were independently sorted & later merged.

In Quick sort, the division into 2 sub arrays is made so that the sorted subarrays do not need to be merged later.

This is accomplished by rearranging the elements in a[1:n] such that a[I]<=a[j] for all I between 1 & n and all j between (m+1) & n for some m, 1<=m<=n.

Thus the elements in a[1:m] & a[m+1:n] can be independently

sorted.No merge is needed. This rearranging is referred to as

partitioning.

Function partition of Algorithm accomplishes an in-place partitioning of the elements of a[m:p-1]

It is assumed that a[p]>=a[m] and that a[m] is the partitioning element. If m=1 & p-1=n, then a[n+1] must be defined and must be greater than or equal to allelements in a[1:n]

The assumption that a[m] is the partition element is merely for convenience, other choices for the partitioning element than the first item in theset are better in practice.

The function interchange (a,I,j) exchanges a[I] with a[j].

Algorithm: Partition the array a[m:p-1] about a[m]

```
1.    Algorithm Partition(a,m,p)
2.    //within a[m],a[m+1],…..,a[p-1] the elements
3.    // are rearranged in such a manner that if
4.    //initially t=a[m],then after completion
5.    //a[q]=t for some q between m and
6.    //p-1,a[k]<=t for m<=k<q, and
7.    //a[k]>=t for q<k<p. q is returned
8.    //Set a[p]=infinite.
9.    {
10.   v:=a[m];I:=m;j:=p;


11.   repeat
12.   {
13.   repeat
14.
      I:=I+1;
15.    until(a[I]>=v);
16.   repeat
17.   j:=j-1;
18.   until(a[j]<=v);
19.   if (I<j) then interchange(a,i.j);
20.   }until(I>=j);
21.   a[m]:=a[j]; a[j]:=v;
22.   retun j;
23.   }
```

```
1.    Algorithm Interchange(a,I,j)
2.    //Exchange a[I] with a[j]
3.    {
```

```
4.      p:=a[I];
5.      a[I]:=a[j];
6.      a[j]:=p;
7.      }
```

Algorithm: Sorting by Partitioning

```
1.      Algorithm Quicksort(p,q)
2.      //Sort the elements a[p],….a[q] which resides
3.      //is the global array a[1:n] into ascending
4.      //order; a[n+1] is considered to be defined
5.      // and must be >= all the elements in a[1:n]6.
        {
7.      if(p<q) then // If there are more than one element8.
        {
9.      // divide p into 2 subproblems
10.     J:=partition(a,p,q+1);
11.     //'j' is the position of the partitioning element.
12.     //solve the subproblems.
13.     quicksort(p,j-1);
14.     quicksort(j+1,q);
15.     //There is no need for combining solution.
16.     }
17.     }
```

STRASSEN'S MATRIX MULTIPLICAION

1. 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 and j between 1 and n.

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

3. Divide and conquer method suggest another way to compute the productof n*n matrix.

4. 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.

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

6. 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 .

7. **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 + A12B21
C12 = A11 B12 + A12B22
C21 = A21 B11 + A22B21
C22 = A21 B12 + A22B22

For EX:

4*4= 2222 2222 2222 2222 * 1 1 1 1 1111 1 1 1 1 1 1 1 1

The Divide and conquer method

2 2 | 2 2 * 1 1 | 1 1 = 4 4 | 4 4
2 2 | 2 2 1 1 | 1 1 4 4 | 4 4
2 2 | 2 2 1 1 1 1 4 4 4 4
2 2 | 2 2 1 1 | 1 1 4 4 | 4 4

8. To compute AB using the equation we need to perform 8 multiplication ofn/2*n/2 matrix and from 4 addition of n/2*n/2 matrix.

9. Ci,j are computed using the formula in equation 4

10. As can be sum P, Q, R, S, T, U, and V can be computed using 7 Matrix multiplication and 10 addition or subtraction.

11. The Cij are required addition 8 addition or subtraction.

T(n)= { b
7T(n/2)+an^2 n<=2 a &b are
n>2 constant

Finally we get T(n) =O(n ^log27)

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)=	b	n<=2 a &b are
	8T(n/2)+cn^2	n>2 constant

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 .

12. Stressen has discovered a way to compute the Cij of equation (2) usingonly 7 multiplication and 18 addition or subtraction.

13. 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
C12=R+T
C21=Q+S
C22=P+R- Q+U