Unit-1

Introduction

ALGORITHM:
Algorithm was first time proposed a purshian mathematician Al-Chwarizmi in 825 AD. According to web star dictionary, algorithm is a special method to represent the procedure to solve given problem.

OR

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 input into the output.

Formal Definition:

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

1. **Input**. Zero or more quantities are externally supplied.
2. **Output**. At least one quantity is produced.
3. **Definiteness**. Each instruction is clear and unambiguous.
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**. Every instruction must very basic so that it can be carried out, in principle, by a person using only pencil & paper.

Areas of study of Algorithm:

- *How to device or design an algorithm–* It includes the study of various design techniques and helps in writing algorithms using the existing design techniques like divide and conquer.
- *How to validate an algorithm–* After the algorithm is written it is necessary to check the correctness of the algorithm i.e for each input correct output is produced, known as algorithm validation. The second phase is writing a program known as program proving or program verification.
- *How to analysis an algorithm–*It is known as analysis of algorithms or performance analysis, refers to the task of calculating time and space complexity of the algorithm.
- How to test a program – It consists of two phases . 1. debugging is detection and correction of errors. 2. Profiling or performance measurement is the actual amount of time required by the program to compute the result.

Algorithm Specification:

Algorithm can be described in three ways.

1. Natural language like English:

2. Graphic representation called flowchart:

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

3. Pseudo-code Method:

In this method, we should typically describe algorithms as program, which resembles language like Pascal &algol.

Pseudo-Code for writing Algorithms:

1. Comments begin with // and continue until the end of line.
2. Blocks are indicated with matching braces {and}.
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 items of a record can be accessed with → and period.

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

```
<Variable>:= <expression>;
```

6. There are two Boolean values TRUE and FALSE.

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

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>
}
```

One step is a key word, other Step is used for increment or decrement.

repeat-until:

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

8. A conditional statement has the following forms.

   (1) If <condition> then <statement>

   (2) If <condition> then <statement-1>

       Else <statement-2>

   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 list>)

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

```
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;
    }
```

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

Performance Analysis.

There are many Criteria to judge an algorithm.

- Is it correct?
- Is it readable?
- How it works

Performance evaluation can be divided into two major phases.

1. Performance Analysis (machine independent)
   - space complexity: The space complexity of an algorithm is the amount of memory it needs to run for completion.
   - time complexity: The time complexity of an algorithm is the amount of computer time it needs to run to completion.

2 .Performance Measurement (machine dependent).

Space Complexity:

The Space Complexity of any algorithm P is given by $S(P)=C+S_P(I)$,C is constant.

1.Fixed Space Requirements (C)
Independent of the characteristics of the inputs and outputs

- It includes instruction space
- space for simple variables, fixed-size structured variable, constants

2. Variable Space Requirements ($S_P(I)$)
   depend on the instance characteristic I
   - number, size, values of inputs and outputs associated with I
   - recursive stack space, formal parameters, local variables, return address

Examples:

Program 1 :Simple arithmetic function

```
Algorithmabc( a,  b, c)
{
   return a + b + b * c + (a + b - c) / (a + b) + 4.00;
 }
```

$S_P(I)=0$

Hence**S(P)=Constant**

Program 2: Iterative function for sum a list of numbers

```
Algorithm sum( list[ ], n)
{
tempsum = 0;
  for i = 0 ton do
tempsum += list [i];
  return tempsum;

}
```

In the above example list[] is dependent on n. Hence $S_P(I)=n$. The remaining variables are i,n, tempsum each requires one location.

Hence **S(P)=3+n**

<u>***Program 3:** Recursive function for sum a list of numbers</u>

```
Algorithmrsum( list[ ],  n)
{

If (n<=0) then

return 0.0

else
 return rsum(list, n-1) + list[n];


 }
```

In the above example the recursion stack space includes space for formal parameters local variables and return address. Each call to rsum requires 3 locations i.e for list[],n and return address .As the length of recursion is n+1.

S(P)>=3(n+1)

Time complexity:

$$\mathbf{T(P)=C+T_P(I)}$$

It is combination of-Compile time (C)
independent of instance characteristics
-run (execution) time T_P
dependent of instance characteristics

Time complexity is calculated in terms of *program step* as it is difficult to know the complexities of individual operations.

Definition: A*program step* is a syntactically or semantically meaningful program segment whose execution time is independent of the instance characteristics.

Program steps are considered for different statements as : for comment zero steps . assignment statement is considered as one step. Iterative statements such as “for, while and until-repeat” statements, we consider the step counts based on the expression .

Methods to compute the step count:

1) Introduce variable count into programs
2) Tabular method
   - Determine the total number of steps contributed by each statement
     step per execution × frequency
   - add up the contribution of all statements

<u>Program 1.</u><u>with count statements</u>

```
Algorithm sum( list[ ], n)
{
tempsum := 0; count++; /* for assignment */
   for  i := 1  to n do {
count++;            /*for the for loop */
tempsum := tempsum + list[i]; count++;  /* for assignment */
   }
count++;        /* last execution of for */
   return tempsum;
count++;        /* for return */
```

Hence **T(n)=2n+3**

Program :Recursive sum

```
Algorithmrsum( list[ ],  n)
{
      count++;      /*for if conditional */
      if (n<=0) {
               count++; /* for return */
      return 0.0 }

else

returnrsum(list, n-1) + list[n];

      count++;/*for return and rsum invocation*/

}
```

T(n)=2n+2

Program for matrix addition

```
Algorithm add( a[ ][MAX_SIZE],  b[ ][MAX_SIZE],
                     c[ ][MAX_SIZE],  rows, cols )
{
  for i := 1 to rows do {
count++; /* for i for loop */
      for  j :=  1 to cols do {
count++; /* for j for loop */
        c[i][j] := a[i][j] + b[i][j];
count++; /* for assignment statement */
      }
count++;    /* last time of j for loop */
  }
count++;        /* last time of i for loop */
}
```

T(n)=2rows*cols+2*rows+1

II Tabular method.

Complexity is determined by using a table which includes steps per execution(s/e) i.e amount by which count changes as a result of execution of the statement.

Frequency – number of times a statement is executed.

Statement	s/e	Frequency	Total steps
Algorithm sum(list[ ], n)	0	-	0
{	0	-	0
tempsum := 0;	1	1	1
for i := 0 ton do	1	n+1	n+1
tempsum := tempsum + list [i];	1	n	n
return tempsum;	1	1	1
}	0	0	0
Total			2n+3

Statement	s/e	Frequency		Total steps	
		n=0	n>0	n=0	n>0
Algorithmrsum(list[ ], n)	0	-	-	0	0
{	0	-	-	0	0
If (n<=0) then	1	1	1	1	1
return 0.0;	1	1	0	1	0
else	0	0	0	0	0
return rsum(list, n-1) + list[n];	1+x	0	1	0	1+x
}	0	0	0	0	0
Total				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
for j:=1 to n do	1	m(n+1)	mn+m
c[i,j]:=a[i,j]+b[i,j];	1	mn	mn
}	0	-	0
Total			2mn+2m+1

Complexity ofAlgorithms

The complexity of an algorithm M is the function f(n) which gives the running time and/or storage space requirement of the algorithm in terms of the size 'n' of the input data. Mostly, the storage space required by an algorithm is simply a multiple of the data size 'n'. Complexity shall refer to the running time of thealgorithm.
The function f(n), gives the running time of an algorithm, depends not only on the size 'n' of the input data but also on the particular data. The complexity function f(n) for certain casesare:

1. Best Case : The minimum possible value of f(n) is called the bestcase.
2. Average Case : The average value off(n).
3. Worst Case : The maximum value of f(n) for any key possibleinput.

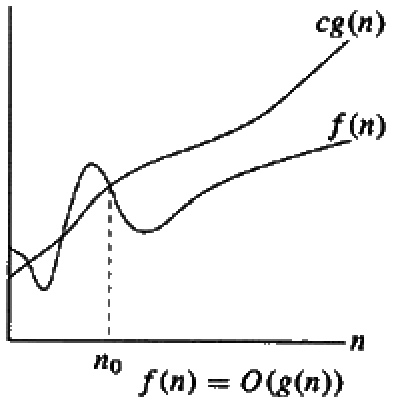
The field of computer science, which studies efficiency of algorithms, is known as analysis ofalgorithms.
Algorithms can be evaluated by a variety of criteria. Most often we shall be interested in the rate of growth of the time or space required to solve larger and larger instances of a problem. We will associate with the problem an integer, called the size of the problem, which is a measure of the quantity of inputdata.Rate ofGrowth:

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

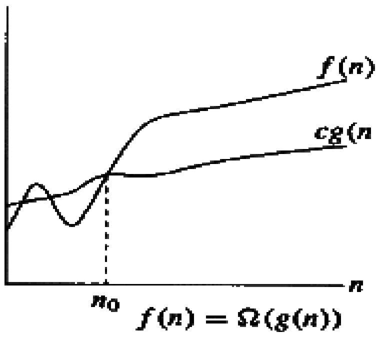
Asymptotic notation

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 **$f(n) \leq C*g(n)$** for all n, $n \geq 0$

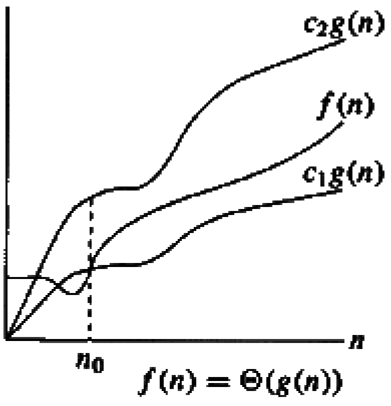
The value g(n)is the upper bound value of f(n).
Example:
3n+2=O(n) as
$3n+2 \leq 4n$ for all $n \geq 2$



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
The value g(n) is the lower bound value of f(n).
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 for all n, n≥0

n→~

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 ohomega 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→~

AnalyzingAlgorithms

Suppose 'M' is an algorithm, and suppose 'n' is the size of the input data. Clearly the complexity f(n) of M increases as n increases. It is usually the rate of increase of f(n) we want to examine. This is usually done by comparing f(n) with some standard functions. The most common computing timesare:

O(1), O($\log_2 n$), O(n), O(n. $\log_2 n$), O(n^2), O(n^3), O(2^n), n! andn^n

Numerical Comparison of DifferentAlgorithms

The execution time for six of the typical functions is givenbelow:

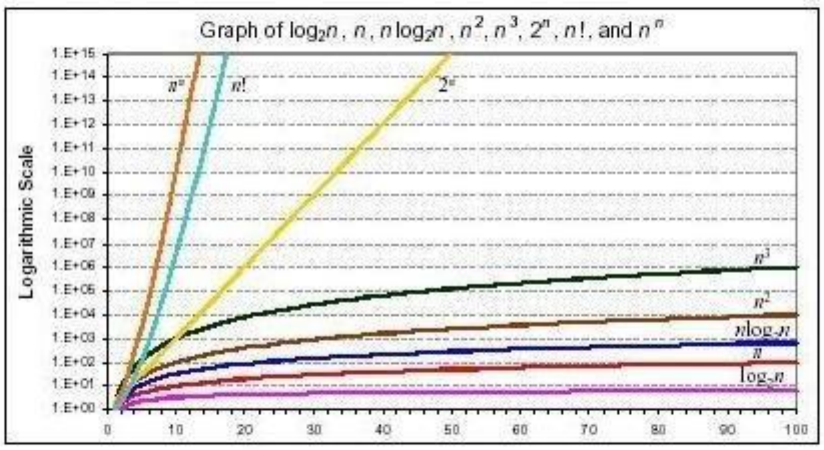
N	log2n	n*log2n	n^2	n^3	2^n

1	0	0	1	1	2
2	1	2	4	8	4
4	2	8	16	64	16
8	3	24	64	512	256
16	4	64	256	4096	65,536
32	5	160	1024	32,768	4,294,967,296
64	6	384	4096	2,62,144	Note1
128	7	896	16,384	2,097,152	Note2
256	8	2048	65,536	1,677,216	????????

Note1: The value here is approximately the number of machine instructions executed by a 1 gigaflop computer in 5000years.

Note 2: The value here is about 500 billion times the age of the universe in nanoseconds, assuming a universe age of 20 billionyears.

Graph of log n, n, n log n, n^2, n^3, 2^n, n! andn^n



One way to compare the function f(n) with these standard function is to use the functional 'O' notation, suppose f(n) and g(n) are functions defined on the positive integers with the property that f(n) is bounded by some multiple g(n) for almost all 'n'.Then,f(n) =O(g(n)) Which is read as "f(n) is of order g(n)". For example, the order of complexityfor:

- Linear search is O(n)
- Binary search is O (logn)
- Bubble sort is O(n^2)
- Merge sort is O (n logn)

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.

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, $1<k<=n$, 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.DAndC(Algorithm) is initially invoked as DandC(P), where ‘p’ is the problem to be solved.Small(P) is a Boolean-valued function that determines whether the i/p size is 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 …P_k are solved by recursive application of DAndC.Combine is a function that determines the solution to p using the solutions to 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 DAndC is described by the recurrence relation.

$T(n)= \{ g(n)$ n small

$T(n1)+T(n2)+\ldots\ldots\ldots\ldots+T(n_k)+f(n);$ otherwise.

Where T(n) is the time for DAndC 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 to sub problems.

```
Algorithm DAndC(P)
  {
  if small(P) then return S(P);
  else
  {
  divide P into smaller instances
          P1, P2… Pk, k>=1;

   Apply DAndC to each of these sub problems;
  return combine (DAndC(P1), DAndC(P2),…….,DAndC(Pk));
   }
  }
```

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

$T(n) = T(1)$ n=1

= $aT(n/b)+f(n)$ $n>1$

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:

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

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

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

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

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

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

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

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

$= 8T(n/8)+3n$

*
*

- In general, we see that $T(n)=2^iT(n/2^i)+in.$, for any $\log_2 n >=i>=1$.

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

Corresponding to the choice of $i=\log_2 n$

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, $2^0=1$]

$= 2n + n \log n$

$T(n)= n\log n+2n.$

The recurrence using the substitution method,it can be shown as

$$\mathbf{T(n)=n^{\log_b a}[T(1)+u(n)]}$$

h(n)	**u(n)**
$O(n^{r)},r<0$	O(1)

$\Theta((\log n)^i), i \geq 0$	$\Theta((\log n)^{i+1}/(i+1))$
$\Omega(n^r), r>0$	$\Theta(h(n))$

Applications of Divide and conquer rule or algorithm:

Binary search, Quick sort, Merge sort, Strassen’s matrix multiplication.

BINARY SEARCH

Given a list of n elements arranged in increasing order. The problem is to determine whether a given element is present in the list or not. If x is present then determine the position of x, otherwise position is zero.

Divide and conquer is used to solve the problem. The value Small(p) is true if n=1. S(P)= i, if x=a[i], a[] is an array otherwise S(P)=0.If P has more than one element then it can be divided into sub-problems. Choose an index j and compare x with $a_{j.}$ then there 3 possibilities (i). X=a[j] (ii) x<a[j] (x is searched in the list a[1]…a[j-1])
(iii) x>a[j] (x is searched in the list a[j+1]…a[n]).
And the same procedure is applied repeatedly until the solution is found or solution is zero.

```
Algorithm Binsearch(a,n,x)
    // Given an array a[1:n] of elements in non-decreasing
    //order, n>=0,determine whether ‘x’ is present and
    // if so, return ‘j’ such that x=a[j]; else return 0.
    {
    low:=1; high:=n;
    while (low<=high) do
    {
    mid:=[(low+high)/2];
    if (x<a[mid]) then high;
    else if(x>a[mid]) then
low:=mid+1;
    else return mid;
      }
    return 0;
    }
```

Algorithm, describes this binary search method, where Binsrch has 4 inputssa[], I , n& x.It is initially invoked as Binsrch (a,1,n,x)A non-recursive version of Binsrch is given below. This Binsearch has 3 i/psa,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 Binsearch on these 3 steps.

X=151	low	high	mid
	1	147	
	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.

The complexity of binary search is**successful searches** is
Worst case is O(log n) or θ(log n)
Average case is O(log n) or θ(log n)
Best case is O(1) or θ(1)

Unsuccessful searches is: θ(log n) for all cases.

MergeSort

Merge sort algorithm is a classic example of divide and conquer. To sort an array, recursively, sort its left and right halves separately and then merge them. The time complexity of merge sort in the *best case, worst case* and *average case* is O(n log n) and the number of comparisons used is nearlyoptimal.

This strategy is so simple, and so efficient but the problem here is that there seems to be no easy way to merge two adjacent sorted arrays together in place (The result must be build up in a separatearray).The fundamental operation in this algorithm is merging two sorted lists. Because the lists are sorted, this can be done in one pass through the input, if the output is put in a thirdlist.

```
Algorithm MERGESORT (low,high)

  // a (low : high) is a global array to besorted.
  {
          if (low <high)
          {
  mid := (low +high)/2;//finds where to split theset
  MERGESORT(low, mid); //sortonesubset
  MERGESORT(mid+1, high);  //sort the other subset
                  MERGE(low,mid,high);  // combine theresults
          }

  }
```

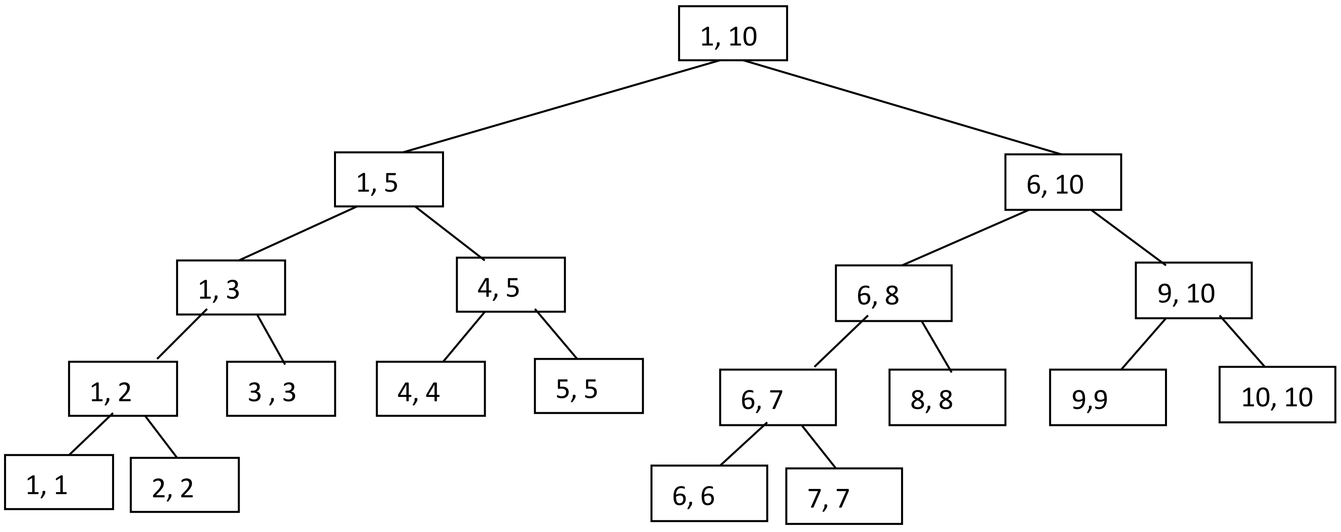
```
Algorithm MERGE (low, mid,high)

  // a (low : high) is a global array containing two sortedsubsets
  // in a (low : mid) and in a (mid + 1 :high).
  // The objective is to merge these sorted sets into singlesorted
  // set residing in a (low : high). An auxiliary array B isused.
  {
          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 highdo
                  {
                      b[i]       :=       a[k];       i       :=       i       +1
for k := h to middo
```

```
                {
                        b[i] := a[K];  i := i +l;
                }
        for k := low to highdo
                a[k] :=b[k];
}
```

Example

Tree call of Merge sort:

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



Tree call of Merge sort (1, 10)

Analysis of MergeSort

We will assume that ‘n’ is a power of 2, so that we always split into even halves, so we solve for the case n $=2^k$.

For n = 1, the time to merge sort is constant, which we will be denote by 1. Otherwise, the time to merge sort ‘n’ numbers is equal to the time to do two recursive merge sorts of size n/2, plus the time to merge, which is linear. The equation says thisexactly:

$T(1) = 1$
$T(n) = 2\,T(n/2) + n$

This is a standard recurrence relation, which can be solved several ways. We will solve by substituting recurrence relation continually on the right–handside.

We have, $T(n) = 2T(n/2) + n$

Since we can substitute n/2 into this mainequation

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

Wehave,

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

Again, by substituting n/4 into the main equation, we seethat

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

So wehave,

$$T(n/4) = 2T(n/8) + n$$
$$T(n) = 8T(n/8) + 3n$$

Continuing in this manner, weobtain:

$$T(n) = 2^k T(n/2^k) + K.n$$

As $n = 2^k$, $K = \log_2 n$, substituting this in the aboveequation

$T(n) = 2^{\log n} T(n/2^{\log n}) + \log n * n$

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

$= n + n \log n$

Representing in O-notation $T(n)=O(n \log n)$.

We have assumed that $n = 2^k$. The analysis can be refined to handle cases when 'n' is not a power of 2. The answer turns out to be almostidentical.

Although merge sort's running time is O(n log n), it is hardly ever used for main memory sorts. The main problem is that merging two sorted lists requires linear extra memory and the additional work spent copying to the temporary array and back, throughout the algorithm, has the effect of slowing down the sort considerably. *The Best and worst case time complexity of Merge sort is O(n logn).*

Strassen's MatrixMultiplication:

The matrix multiplication of algorithm due to Strassens is the most dramatic example of divide and conquer technique(1969).

Let A and B be two n×n Matrices. The product matrix C=AB is also a n×n matrix whose i, jth element is formed by taking elements in the ith row of A and jth column of B and multiplying them to get

The usual way$C(i, j)=\sum_{1\leq k\leq n} A(i,k)B(k,j)$

Here $1\leq i$ & $j \leq n$ means i and j are in between 1 and n.

To compute C(i, j) using this formula, we need n multiplications.

The divide and conquer strategy suggests another way to compute the product of two n×n matrices.For Simplicity assume n is a power of 2 that is $n=2^k$, k is a nonnegative integer.
If n is not power of two then enough rows and columns of zeros can be added to both A and B, so that resulting dimensions are a power of two.

To multiply two n x n matrices A and B, yielding result matrix 'C' as follows:
Let A and B be two n×n Matrices. Imagine that A & B are each partitioned into four square sub matrices. Each sub matrix having dimensions n/2×n/2.
The product of AB can be computed by using previous formula.
If AB is product of 2×2 matrices then

$$\begin{pmatrix} A11 & A12 \\ A21 & A22 \end{pmatrix} \begin{pmatrix} B11 & B12 \\ B21 & B22 \end{pmatrix} = \begin{pmatrix} 11 & C12 \\ C21 & C22 \end{pmatrix}$$

Then c_{ij}can be found by the usual matrix multiplicationalgorithm,

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

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

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

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

This leads to a divide–and–conquer algorithm, which performs nxn matrix multiplication by partitioning the matrices into quarters and performing eight (n/2)x(n/2) matrix multiplications and four (n/2)x(n/2) matrixadditions.

$T(1) = 1$
$T(n) = 8T(n/2)$

Which leads to T (n) = O (n^3), where n is the power of2.
Strassens insight was to find an alternative method for calculating the C_{ij}, requiring seven (n/2) x (n/2) matrix multiplications and eighteen (n/2) x (n/2) matrix additions andsubtractions:

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

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

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

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

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

$U = (A_{21} – A_{11}) (B_{11} + B_{12})$

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

$C_{11} = P + S – T + V$

$C_{12} = R + T$

$C_{21} = Q + S$

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

This method is used recursively to perform the seven (n/2) x (n/2) matrix multiplications, then the recurrence equation for the number of scalar multiplications performedis:

$$T(1) = 1$$
$$T(n) = 7T(n/2)$$

Solving this for the case of $n = 2^k$ iseasy:

$$T(2^k) = 7T(2^{k-1})$$
$$= 7^2T(2^{k-}$$
$$= \text{- - - - --}$$
$$= \text{- - - - --}$$
$$= 7^iT(2^{k-i})$$

Put i =k

$$=7^kT(2^0)$$

As k is the power of 2

That is, $T(n) = 7^{\log_2 n}$

$$= n^{\log_2 7}$$

$$=O(n^{\log_2 7})= O(n^{2.81})$$

So, concluding that Strassen's algorithm is asymptotically more efficient than the standard algorithm. In practice, the overhead of managing the many small matrices does not pay off until 'n' revolves thehundreds.

QuickSort

The main reason for the slowness of Algorithms in which all comparisons and exchanges between keys in a sequence w1, w2, ... , wn take place between adjacent pairs. In this way it takes a relatively long time for a key that is badly out of place to work its way into its proper position in the sortedsequence.

Hoare his devised a very efficient way of implementing this idea in the early 1960's that improves the $O(n^2)$ behavior of the algorithm with an expected performance that is O(n logn).In essence, the quick sort algorithm partitions the original array by rearranging it into two groups. The first group contains those elements less than some arbitrary chosen value taken from the set, and the second group contains those elements greater than or equal to the chosenvalue.

The chosen value is known as the *pivot element*. Once the array has been rearranged in this way with respect to the pivot, the very same partitioning is recursively applied to each of the two subsets. When all the subsets have been partitioned and rearranged, the original array issorted.

The function partition() makes use of two pointers 'i' and 'j' which are moved toward

each other in the followingfashion:
Repeatedly increase the pointer ‘i’ until a[i] >=pivot.
Repeatedly decrease the pointer ‘j’ until a[j] <=pivot.
If j > i, interchange a[j] witha[i]
Repeat the steps 1, 2 and 3 till the ‘i’ pointer crosses the ‘j’ pointer. If ‘i’ pointer crosses ‘j’ pointer, the position for pivot is found and place pivot element in ‘j’ pointerposition.
The program uses a recursive function quicksort(). The algorithm of quick sort function sorts all elements in an array ‘a’ between positions ‘low’ and‘high’.
It terminates when the condition low >= high is satisfied. This condition will be satisfied only when the array is completelysorted.Here we choose the first element as the ‘pivot’. So, pivot = x[low]. Now it calls the partition function to find the proper position j of the element x[low] i.e. pivot. Then we will have two sub-arrays x[low], x[low+1], ...
. . x[j-1] and x[j+1], x[j+2], x[high].It calls itself recursively to sort the left sub-array x[low], x[low+1], ... x[j-1] between positions low and j-1 (where j is returned by the partitionfunction).It calls itself recursively to sort the right sub-array x[j+1], x[j+2], ... x[high] between positions j+1 andhigh.

Algorithm

```
AlgorithmQUICKSORT(low,high)
// sorts the elements a(low), . . . . . , a(high) which reside in the global array A(1 :n) into
//ascending order a (n + 1) is considered to be defined and must be greater than all
//elements in a(1 : n); A(n + 1) = α*/
{
        If( low < high) then
        {
                j := PARTITION(a, low,high+1);
                                        // J is the position of the partitioningelement
                QUICKSORT(low, j –1);
                QUICKSORT(j + 1 ,high);
        }
}
```

```
Algorithm PARTITION(a, m,p)

{
        V :=a(m); i :=m; j:=p;
        // a (m) is thepartitionelement
        do
        {
                repeat
                 i := i +1;
               until (a(i)≥v);
                repeat
                 j := j –1;
                until (a(j)≤v);
                if (i < j) then INTERCHANGE(a, i,j)
         } while (i ≥j);
        a[m] :=a[j];a[j]:=V;
        returnj;
}
```

Algorithm INTERCHANGE(a, i,j)

```
{
    p:= a[i];
    a[i]:=a[j];
    a[j]:=p;
}
```

Example

Select first element as the pivot element. Move ‘i’ pointer from left to right in search of an element larger than pivot. Move the ‘j’ pointer from right to left in search of an element smaller than pivot. If such elements are found, the elements are swapped. This process continues till the ‘i’ pointer crosses the ‘j’ pointer. If ‘i’ pointer crosses ‘j’ pointer, the position for pivot is found and interchange pivot and element at ‘j’ position.

Let us consider the following example with 13 elements to analyze quicksort:

1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*Remarks*
38	08	16	06	79	57	24	56	02	58	04	70	45	
pivot				I						j			swap i &j
				04						79			
					i			j					swap i &j
					02			57					
						j	i						
(24	08	16	06	04	02)	**38**	(56	57	58	79	70	45)	swap pivot
pivot					j,i								swap pivot
(02	08	16	06	04)	**24**								
pivot , j	i												swap pivot
02	(08	16	06	04)									
	pivot	i		j									swap i &j
		04		16									
			j	i									
	(06	04)	**08**	(16)									swap pivot
	pivot , j	i											
	(04)	**06**											swap pivot
	04 pivot , j,i												
				16 pivot , j,i									
(02	**04**	**06**	**08**	**16**	**24)**	38							
							(56	57	58	79	70	45)	

							pivot	i				j	swap i
								45				57	
								j	i				
							(45)	**56**	(58	79	70	57)	swap pivot
							45 pivot , j,i						swap pivot
									(58 pivo	79 i	70	57) j	swap i
										57		79	
										j	i		
									(57)	**58**	(70	79)	swap pivot
									57 pivot , j,i				
											(70	79)	
											pivot , j	i	swap pivot
											70		
												79 pivot , j,i	
							(45	**56**	**57**	**58**	**70**	**79)**	
02	**04**	**06**	**08**	**16**	**24**	**38**	**45**	**56**	**57**	**58**	**70**	**79**	

Analysis of QuickSort:

Like merge sort, quick sort is recursive, and hence its analysis requires solving a recurrence formula. We will do the analysis for a quick sort, assuming a random pivot
We will take T (0) = T (1) = 1, as in merge sort.
The running time of quick sort is equal to the running time of the two recursive calls plus the linear time spent in the partition (The pivot selection takes only constant time). This gives the basic quick sortrelation:

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

Where, $i = |S_1|$ is the number of elements inS_1.

Worst CaseAnalysis

The pivot is the smallest element, all the time. Then i=0 and if we ignore T(0)=1, which is insignificant, the recurrenceis:

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

Using equation – (1) repeatedly,thus

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

T (n – 2) = T (n – 3) + C (n –2)

- - - - - - - --

T (2) = T (1) + C(2)

Adding up all these equationsyields

$=\mathbf{O(n^2)}$ - (3)

Best CaseAnalysis

In the best case, the pivot is in the middle. To simply the math, we assume that the two sub-files are each exactly half the size of the original and although this gives a slight over estimate, this is acceptable because we are only interested in a Big – oh answer.

T (n) = 2 T (n/2) +Cn - (4)

Divide both sides byn and Substitute n/2 for ‘n’

Finally,

Which yields, T (n) = C n log n + n = **O(n logn)** -

This is exactly the same analysis as merge sort, hence we get the sameanswer.

Average CaseAnalysis

The number of comparisons for first call on partition: Assume left_to_right moves over k smaller element and thus k comparisons. So when right_to_left crosses left_to_right it has made n-k+1 comparisons. So, first call on partition makes n+1 comparisons. The average case complexity of quicksort is

T(n) = comparisons for first call onquicksort

+

{Σ 1<=nleft,nright<=n [T(nleft) + T(nright)]}n = (n+1) + 2 [T(0) +T(1) + T(2) + ----- +T(n-1)]/n

nT(n) = n(n+1) + 2 [T(0) +T(1) + T(2) + ----- + T(n-2) +T(n-1)]

(n-1)T(n-1) = (n-1)n + 2 [T(0) +T(1) + T(2) + ----- + T(n-2)]\

Subtracting bothsides:

nT(n) –(n-1)T(n-1) = [n(n+1) – (n-1)n] + 2T(n-1) = 2n + 2T(n-1) nT(n) = 2n + (n-1)T(n-1) + 2T(n-1) = 2n +(n+1)T(n-1)

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

The recurrence relation obtained is:

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

Using the method ofsubstitution:

T(n)/(n+1)	=	2/(n+1) +T(n-1)/n
T(n-1)/n	=	2/n +T(n-2)/(n-1)
T(n-2)/(n-1)	=	2/(n-1) +T(n-3)/(n-2)
T(n-3)/(n-2)	=	2/(n-2) +T(n-4)/(n-3)
.		.
.		.
T(3)/4	=	2/4 +T(2)/3

T(2)/3 = 2/3 + T(1)/2 T(1)/2 = 2/2 +T(0)

Adding bothsides:

T(n)/(n+1) + [T(n-1)/n + T(n-2)/(n-1) +--------------+ T(2)/3 +T(1)/2]

= [T(n-1)/n + T(n-2)/(n-1) + ------------- + T(2)/3 + T(1)/2] + T(0)+ [2/(n+1)

+ 2/n + 2/(n-1) + ----------+2/4 +2/3]

Cancelling the commonterms:

T(n)/(n+1) = 2[1/2 +1/3+1/4+-------------+1/n+1/(n+1)]

Finally,

We will get,

O(n log n)