UNIT – I

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 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.

Issues or study of Algorithm:

• How to device or design an algorithm ➔ creating and algorithm.
• How to express an algorithm ➔ definiteness.
• How to analysis an algorithm ➔ time and space complexity.
• How to validate an algorithm ➔ fitness.
• Testing the algorithm ➔ checking for error.

Algorithm Specification:

Algorithm can be described in three ways.

1. Natural language like English:
   When this way is choused care should be taken, we should ensure that each & every statement is definite.

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 Conventions:

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>
}

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:
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.

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

Selection Sort:

- Suppose we Must devise an algorithm that sorts a collection of n>=1 elements of arbitrary type.

- A Simple solution given by the following.

- ( From those elements that are currently unsorted ,find the smallest & place it next in the sorted list.)

Algorithm:

1. For i:= 1 to n do
2. {
3.     Examine a[I] to a[n] and suppose the smallest element is at a[j];
4.     Interchange a[I] and a[j];
5. }

→ Finding the smallest element (sat a[j]) and interchanging it with a[ i ]

- We can solve the latter problem using the code,

        t := a[i];
        a[i]:=a[j];
        a[j]:=t;
The first subtask can be solved by assuming the minimum is a[I]; checking a[I] with a[I+1], a[I+2], ……, and whenever a smaller element is found, regarding it as the new minimum. a[n] is compared with the current minimum.

Putting all these observations together, we get the algorithm Selection sort.

**Theorem:**
Algorithm selection sort(a,n) correctly sorts a set of n≥1 elements. The result remains in a[1:n] such that a[1] ≤ a[2] ≤ … ≤ a[n].

**Selection Sort:**
Selection Sort begins by finding the least element in the list. This element is moved to the front. Then the least element among the remaining elements is found out and put into second position. This procedure is repeated till the entire list has been studied.

**Example:**

LIST L = 3,5,4,1,2

1 is selected, → 1,5,4,3,2
2 is selected, → 1,2,4,3,5
3 is selected, → 1,2,3,4,5
4 is selected, → 1,2,3,4,5

**Proof:**
- We first note that any i, say i=q, following the execution of lines 6 to 9, it is the case that a[q] ≥ a[r], q < r ≤ n.
- Also observe that when ‘i’ becomes greater than q, a[1:q] is unchanged. Hence, following the last execution of these lines (i.e. i=n). We have a[1] ≤ a[2] ≤ … ≤ a[n].
- We observe this point that the upper limit of the for loop in the line 4 can be changed to n-1 without damaging the correctness of the algorithm.

**Algorithm:**

1. Algorithm selection sort (a,n)
2. // Sort the array a[1:n] into non-decreasing order.
3. {
4. for I:=1 to n do
5. {
6. j:=I;
7. for k:=i+1 to n do
8. if (a[k]<a[j])
9. t:=a[I];
10. a[I]:=a[j];
11. a[j]:=t;
Recursive Algorithms:

- A Recursive function is a function that is defined in terms of itself.
- Similarly, an algorithm is said to be recursive if the same algorithm is invoked in the body.
- An algorithm that calls itself is Direct Recursive.
- Algorithm ‘A’ is said to be Indirect Recursive if it calls another algorithm which in turns calls ‘A’.
- The Recursive mechanism, are externally powerful, but even more importantly, many times they can express an otherwise complex process very clearly. Or these reasons we introduce recursion here.
- The following 2 examples show how to develop a recursive algorithms.

  ➔ In the first, we consider the Towers of Hanoi problem, and in the second, we generate all possible permutations of a list of characters.

1. Towers of Hanoi:

   • It is Fashioned after the ancient tower of Brahma ritual.
   • According to legend, at the time the world was created, there was a diamond tower (labeled A) with 64 golden disks.
   • The disks were of decreasing size and were stacked on the tower in decreasing order of size bottom to top.
   • Besides these tower there were two other diamond towers(labeled B & C)
   • Since the time of creation, Brehman priests have been attempting to move the disks from tower A to tower B using tower C, for intermediate storage.
   • As the disks are very heavy, they can be moved only one at a time.
   • In addition, at no time can a disk be on top of a smaller disk.
   • According to legend, the world will come to an end when the priest have completed this task.
A very elegant solution results from the use of recursion.

Assume that the number of disks is ‘n’.

To get the largest disk to the bottom of tower B, we move the remaining ‘n-1’ disks to tower C and then move the largest to tower B.

Now we are left with the tasks of moving the disks from tower C to B.

To do this, we have tower A and B available.

The fact, that towers B has a disk on it can be ignored as the disks larger than the disks being moved from tower C and so any disk can be placed on top of it.

Algorithm:

1. Algorithm TowersofHanoi(n,x,y,z)
2. //Move the top ‘n’ disks from tower x to tower y.
3. {
   .
   .
   .
4. if(n>=1) then
5. {
6.    TowersofHanoi(n-1,x,z,y);
7.    Write("move top disk from tower " X "," to top of tower ",Y);
8.    Towersofhanoi(n-1,z,y,x);
9. }
10. }

2 Permutation Generator:

Given a set of n>=1 elements, the problem is to print all possible permutations of this set.

For example, if the set is {a,b,c} ,then the set of permutation is,

{ (a,b,c),(a,c,b),(b,a,c),(b,c,a),(c,a,b),(c,b,a) }

It is easy to see that given ‘n’ elements there are n! different permutations.

A simple algorithm can be obtained by looking at the case of 4 statement(a,b,c,d)

The Answer can be constructed by writing

1. a followed by all the permutations of (b,c,d)
2. b followed by all the permutations of(a,c,d)
3. c followed by all the permutations of (a,b,d)
4. d followed by all the permutations of (a,b,c)

Algorithm:

Algorithm perm(a,k,n)
if(k=n) then write (a[1:n]); // output permutation
else //a[k:n] ahs more than one permutation
    // Generate this recursively.
for I:=k to n do
{
    t:=a[k];
    a[k]:=a[I];
    a[I]:=t;
    perm(a,k+1,n);
    //all permutation of a[k+1:n]
    t:=a[k];
    a[k]:=a[I];
    a[I]:=t;
}

Performance Analysis:

1. **Space Complexity:**
   The space complexity of an algorithm is the amount of money it needs to run
to compilation.

2. **Time Complexity:**
   The time complexity of an algorithm is the amount of computer time it needs
to run to compilation.

Space Complexity:

Space Complexity Example:

```plaintext
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.

2. 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 on instance characteristics), and the recursion stack space.

- The space requirement $s(p)$ of any algorithm $p$ may therefore be written as,
  \[ S(P) = c + S_p(\text{Instance characteristics}) \]
  Where ‘$c$’ is a constant.

**Example 2:**

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

- The problem instances for this algorithm are characterized by $n$, the number of elements to be summed. The space needed by ‘$n$’ is one word, since it is of type integer.
- The space needed by ‘a’ is the space needed by variables of type array of floating point numbers.
- This is at least ‘$n$’ words, since ‘a’ must be large enough to hold the ‘$n$’ elements to be summed.
- So, we obtain $S_{\text{sum}}(n) \geq (n+s)$

**Time Complexity:**

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

- The compile time does not depend on the instance characteristics. Also we may assume that a compiled program will be run several times without recompilation. This run time is denoted by $tp(\text{instance characteristics})$.
- The number of steps any problem statement is assigned depends on the kind of statement.

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

Interactive statement such as for, while & repeat-until \(\rightarrow\) Control part of the statement.
1. 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. So each invocation of sum execute a total of 2n+3 steps.

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 (i.e., frequency) each statement is executed.

→ By combining these two quantities, the total contribution of all statements, the step count for the entire algorithm is obtained.
### AVERAGE –CASE ANALYSIS

- Most of the time, average-case analysis are performed under the more or less realistic assumption that all instances of any given size are equally likely.
- For sorting problems, it is simple to assume also that all the elements to be sorted are distinct.
- Suppose we have ‘n’ distinct elements to sort by insertion and all n! permutation of these elements are equally likely.
- To determine the time taken on an average by the algorithm, we could add the times required to sort each of the possible permutations, and then divide by n! the answer thus obtained.
- An alternative approach, easier in this case is to analyze directly the time required by the algorithm, reasoning probabilistically as we proceed.
- For any I, 2 ≤ I ≤ n, consider the sub array, T[1...I].
- The partial rank of T[I] is defined as the position it would occupy if the sub array were sorted.
- For Example, the partial rank of T[4] in [3, 6, 2, 5, 1, 7, 4] in 3 because T[1...4] once sorted is [2, 3, 5, 6].
- Clearly the partial rank of T[I] does not depend on the order of the element in Sub array T[1...I-1].

### Analysis

**Best case:**
This analysis constrains on the input, other than size. Resulting in the fastest possible run time

**Worst case:**
This analysis constrains on the input, other than size. Resulting in the fastest possible run time

**Average case:**
This type of analysis results in average running time over every type of input.

---

<table>
<thead>
<tr>
<th>Statement</th>
<th>S/e</th>
<th>Frequency</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Algorithm Sum(a,n)</td>
<td>0</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>2. {</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3. S=0.0;</td>
<td>0</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>4. for I = 1 to n do</td>
<td>1</td>
<td>n+1</td>
<td>n+1</td>
</tr>
<tr>
<td>5. s = s + a[I];</td>
<td>1</td>
<td>n</td>
<td>n</td>
</tr>
<tr>
<td>6. return s;</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>7. }</td>
<td>0</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td></td>
<td></td>
<td><strong>2n+3</strong></td>
</tr>
</tbody>
</table>
Complexity:

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.

Asymptotic notation:

Big ‘oh’: the function \( f(n) = O(g(n)) \) iff there exist positive constants \( c \) and \( n_0 \) such that \( f(n) \leq c \cdot g(n) \) for all \( n, n \geq n_0 \).

Big–OH \( O \) (Upper Bound)

\[ f(n) = O(g(n)), \text{ (pronounced order of or big oh), says that the growth rate of } f(n) \text{ is less than or equal (≤) that of } g(n). \]

Omega: the function \( f(n) = \Omega(g(n)) \) iff there exist positive constants \( c \) and \( n_0 \) such that \( f(n) \geq c \cdot g(n) \) for all \( n, n \geq n_0 \).

Big–OMEGA \( \Omega \) (Lower Bound)

\[ f(n) = \Omega(g(n)), \text{ (pronounced omega), says that the growth rate of } f(n) \text{ is greater than or equal to (≥) that of } g(n). \]

Theta: the function \( f(n) = \Theta(g(n)) \) iff there exist positive constants \( c_1, c_2 \) and \( n_0 \) such that \( c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n) \) for all \( n, n \geq n_0 \).

Big–THETA \( \Theta \) (Same order)

\[ f(n) = \Theta(g(n)), \text{ (pronounced theta), says that the growth rate of } f(n) \text{ equals (=) the growth rate of } g(n) \text{ [if } f(n) = O(g(n)) \text{ and } T(n) = \Omega(g(n)]. \]
Little–OH (o)

\( T(n) = o(p(n)) \) (pronounced little oh), says that the growth rate of \( T(n) \) is less than the growth rate of \( p(n) \) [if \( T(n) = O(p(n)) \) and \( T(n) \neq \Theta(p(n)) \)].

Analyzing Algorithms

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 times are:

- \( O(1) \)
- \( O(\log_2 n) \)
- \( O(n) \)
- \( O(n \cdot \log_2 n) \)
- \( O(n^2) \)
- \( O(n^3) \)
- \( O(2^n) \)
- \( n! \)
- \( n^n \)

Numerical Comparison of Different Algorithms

The execution time for six of the typical functions is given below:

<table>
<thead>
<tr>
<th>n</th>
<th>log2 n</th>
<th>n*log2n</th>
<th>n2</th>
<th>n3</th>
<th>2n</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>8</td>
<td>16</td>
<td>64</td>
<td>16</td>
</tr>
<tr>
<td>8</td>
<td>3</td>
<td>24</td>
<td>64</td>
<td>512</td>
<td>256</td>
</tr>
<tr>
<td>16</td>
<td>4</td>
<td>64</td>
<td>256</td>
<td>4096</td>
<td>65,536</td>
</tr>
<tr>
<td>32</td>
<td>5</td>
<td>160</td>
<td>1024</td>
<td>32,768</td>
<td>4,294,967,296</td>
</tr>
<tr>
<td>64</td>
<td>6</td>
<td>384</td>
<td>4096</td>
<td>2,62,144</td>
<td>Note 1</td>
</tr>
<tr>
<td>128</td>
<td>7</td>
<td>896</td>
<td>16,384</td>
<td>2,097,152</td>
<td>Note 2</td>
</tr>
<tr>
<td>256</td>
<td>8</td>
<td>2048</td>
<td>65,536</td>
<td>1,677,216</td>
<td>????????</td>
</tr>
</tbody>
</table>

Note 1: The value here is approximately the number of machine instructions executed by a 1 gigaflop computer in 5000 years.
Note 2: The value here is about 500 billion times the age of the universe in nanoseconds, assuming a universe age of 20 billion years.

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

O(log n) does not depend on the base of the logarithm. To simplify the analysis, the convention will not have any particular units of time. Thus we throw away leading constants. We will also throw away low–order terms while computing a Big–Oh running time. Since Big-Oh is an upper bound, the answer provided is a guarantee that the program will terminate within a certain time period. The program may stop earlier than this, but never later.

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 complexity for:

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

The rule of sums

Suppose that T1(n) and T2(n) are the running times of two programs fragments P1 and P2, and that T1(n) is O(f(n)) and T2(n) is O(g(n)). Then T1(n) + T2(n), the running time of P1 followed by P2 is O(max f(n), g(n)), this is called as rule of sums.

For example, suppose that we have three steps whose running times are respectively O(n^2), O(n^3) and O(n log n). Then the running time of the first two steps executed sequentially is O (max(n^2, n^3)) which is O(n^3). The running time of all three together is O(max (n^3, n log n)) which is O(n^3).
The rule of products

If \( T1(n) \) and \( T2(n) \) are \( O(f(n)) \) and \( O(g(n)) \) respectively. Then \( T1(n) \times T2(n) \) is \( O(f(n) \times g(n)) \). It follows term the product rule that \( O(c \times f(n)) \) means the same thing as \( O(f(n)) \) if ‘c’ is any positive constant. For example, \( O(n/2) \) is same as \( O(n) \).

Suppose that we have five algorithms A1–A5 with the following time complexities:

- A1: \( n \)
- A2: \( n \log n \)
- A3: \( n^2 \)
- A4: \( n^3 \)
- A5: \( 2^n \)

The time complexity is the number of time units required to process an input of size ‘\( n \)’. Assuming that one unit of time equals one millisecond. The size of the problems that can be solved by each of these five algorithms is:

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Time complexity</th>
<th>Maximum problem size</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>1 second</td>
</tr>
<tr>
<td>A1</td>
<td>( n )</td>
<td>1000</td>
</tr>
<tr>
<td>A2</td>
<td>( n \log n )</td>
<td>140</td>
</tr>
<tr>
<td>A3</td>
<td>( n^2 )</td>
<td>31</td>
</tr>
<tr>
<td>A4</td>
<td>( n^3 )</td>
<td>10</td>
</tr>
<tr>
<td>A5</td>
<td>( 2^n )</td>
<td>9</td>
</tr>
</tbody>
</table>

The speed of computations has increased so much over last thirty years and it might seem that efficiency in algorithm is no longer important. But, paradoxically, efficiency matters more today than ever before. The reason why this is so is that our ambition has grown with our computing power. Virtually all applications of computing simulation of physical data are demanding more speed.

The faster the computer run, the more need are efficient algorithms to take advantage of their power. As the computer becomes faster and we can handle larger problems, it is the complexity of an algorithm that determines the increase in problem size that can be achieved with an increase in computer speed.

Suppose the next generation of computers is ten times faster than the current generation, from the table we can see the increase in size of the problem.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Time Complexity</th>
<th>Maximum problem size before speed up</th>
<th>Maximum problem size after speed up</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1</td>
<td>( n )</td>
<td>S1</td>
<td>10 S1</td>
</tr>
<tr>
<td>A2</td>
<td>( n \log n )</td>
<td>S2</td>
<td>□10 S2 for large S2</td>
</tr>
<tr>
<td>A3</td>
<td>( n^2 )</td>
<td>S3</td>
<td>3.16 S3</td>
</tr>
<tr>
<td>A4</td>
<td>( n^3 )</td>
<td>S4</td>
<td>2.15 S4</td>
</tr>
<tr>
<td>A5</td>
<td>( 2^n )</td>
<td>S5</td>
<td>S5 + 3.3</td>
</tr>
</tbody>
</table>

Instead of an increase in speed consider the effect of using a more efficient algorithm. By looking into the following table it is clear that if minute as a basis for comparison, by replacing algorithm A4 with A3, we can solve a problem six times larger; by replacing A4 with A2 we can solve a problem 125 times larger. These results are for more impressive than the two fold improvement obtained by a ten fold increase in speed. If an hour is used as the basis of comparison, the differences are even more significant.
We therefore conclude that the asymptotic complexity of an algorithm is an important measure of the goodness of an algorithm.

The Running time of a program

When solving a problem we are faced with a choice among algorithms. The basis for this can be any one of the following:

We would like an algorithm that is easy to understand, code and debug.

We would like an algorithm that makes efficient use of the computer’s resources, especially, one that runs as fast as possible.

Measuring the running time of a program

The running time of a program depends on factors such as:

The input to the program.

The quality of code generated by the compiler used to create the object program.

The nature and speed of the instructions on the machine used to execute the program, and

The time complexity of the algorithm underlying the program.

The running time depends not on the exact input but only the size of the input. For many programs, the running time is really a function of the particular input, and not just of the input size. In that case we define T(n) to be the worst case running time, i.e. the maximum overall input of size ‘n’, of the running time on that input. We also consider Tavg(n) the average, over all input of size ‘n’ of the running time on that input. In practice, the average running time is often much harder to determine than the worst case running time. Thus, we will use worst–case running time as the principal measure of time complexity.

Seeing the remarks (2) and (3) we cannot express the running time T(n) in standard time units such as seconds. Rather we can only make remarks like the running time of such and such algorithm is proportional to n2. The constant of proportionality will remain un-specified, since it depends so heavily on the compiler, the machine and other factors.

Asymptotic Analysis of Algorithms:

Our approach is based on the asymptotic complexity measure. This means that we don’t try to count the exact number of steps of a program, but how that number grows with the size of the input to the program. That gives us a measure that will work for different operating systems, compilers and CPUs. The asymptotic complexity is written using big-O notation.

Rules for using big-O:

The most important property is that big-O gives an upper bound only. If an algorithm is O(n2), it doesn’t have to take n2 steps (or a constant multiple of n2). But it can’t
take more than \(n^2\). So any algorithm that is \(O(n)\), is also an \(O(n^2)\) algorithm. If this seems confusing, think of big-O as being like "\(<\". Any number that is \(< n\) is also \(< n^2\).

Ignoring constant factors: \(O(c f(n)) = O(f(n))\), where \(c\) is a constant; e.g. \(O(20 \, n^3) = O(n^3)\)

Ignoring smaller terms: If \(a < b\) then \(O(a+b) = O(b)\), for example \(O(n^2+n) = O(n^2)\)

Upper bound only: If \(a < b\) then an \(O(a)\) algorithm is also an \(O(b)\) algorithm. For example, an \(O(n)\) algorithm is also an \(O(n^2)\) algorithm (but not vice versa).

\(n\) and \(\log n\) are "bigger" than any constant, from an asymptotic view (that means for large enough \(n\)). So if \(k\) is a constant, an \(O(n + k)\) algorithm is also \(O(n)\), by ignoring smaller terms. Similarly, an \(O(\log n + k)\) algorithm is also \(O(\log n)\).

Another consequence of the last item is that an \(O(n \log n + n)\) algorithm, which is \(O(n(\log n + 1))\), can be simplified to \(O(n \log n)\).

Calculating the running time of a program:

Let us now look into how big-O bounds can be computed for some common algorithms.

**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 and goto step 3 otherwise perform final computations and goto step 1

![Diagram](image)

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

**Iteration v/s Recursion:**

**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 0,1,2,3,5,8,13,…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:**

- Mathematical functions such as factorial and Fibonacci series generation can be easily implemented using recursion than iteration.
- In iterative techniques looping of statement is very much necessary.

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

Iteration is more of a bottom up approach. It begins with what is known and from this constructs the solution step by step. The iterative function obviously uses time that is $O(n)$ whereas recursive function has an exponential time complexity. It is always true that recursion can be replaced by iteration and stacks. It is also true that stack can be replaced by a recursive program with no stack.

![Fig 2.6](image)

**SOLVING RECURRENCES:**

- The indispensable last step when analyzing an algorithm is often to solve a recurrence equation.
- With a little experience and intention, most recurrence can be solved by intelligent guesswork.
- However, there exists a powerful technique that can be used to solve certain classes of recurrence almost automatically.
- This is a main topic of this section the technique of the characteristic equation.

1. **Intelligent guess work:**

This approach generally proceeds in 4 stages.

1. Calculate the first few values of the recurrence
2. Look for regularity.
3. Guess a suitable general form.
4. And finally prove by mathematical induction (perhaps constructive induction).

Then this form is correct.
Consider the following recurrence,

\[
T(n) = \begin{cases} 
0 & \text{if } n=0 \\
3T(n/2)+n & \text{otherwise}
\end{cases}
\]

- First step is to replace \( n \div 2 \) by \( n/2 \)
- It is tempting to restrict ‘n’ to being ever since in that case \( n/2 = n/2 \), but recursively dividing an even no. by 2, may produce an odd no. larger than 1.
- Therefore, it is a better idea to restrict ‘n’ to being an exact power of 2.
- First, we tabulate the value of the recurrence on the first few powers of 2.

<table>
<thead>
<tr>
<th>( n )</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>32</th>
</tr>
</thead>
<tbody>
<tr>
<td>( T(n) )</td>
<td>1</td>
<td>5</td>
<td>19</td>
<td>65</td>
<td>211</td>
<td>665</td>
</tr>
</tbody>
</table>

* For instance, \( T(16) = 3 \times T(8) +16 = 3 \times 65 +16 = 211. \)

* Instead of writing \( T(2) = 5 \), it is more useful to write \( T(2) = 3 \times 1 +2. \)

Then,

\[
T(A) = 3 \times T(2) +4 \\
= 3 \times (3 \times 1 +2) +4 \\
= (3^2 \times 1) + (3 \times 2) +4
\]

* We continue in this way, writing ‘n’ as an explicit power of 2.

<table>
<thead>
<tr>
<th>( n )</th>
<th>( T(n) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>( 3 \times 1 +2 )</td>
</tr>
<tr>
<td>( 2^2 )</td>
<td>( 3^2 \times 1 + 3 \times 2 + 2^2 )</td>
</tr>
<tr>
<td>( 2^3 )</td>
<td>( 3^3 \times 1 + 3^2 \times 2 + 3 \times 2^2 + 2^3 )</td>
</tr>
<tr>
<td>( 2^4 )</td>
<td>( 3^4 \times 1 + 3^3 \times 2 + 3^2 \times 2^2 + 3 \times 2^3 + 2^4 )</td>
</tr>
<tr>
<td>( 2^5 )</td>
<td>( 3^5 \times 1 + 3^4 \times 2 + 3^3 \times 2^2 + 3^2 \times 2^3 + 3 \times 2^4 + 2^5 )</td>
</tr>
</tbody>
</table>

- The pattern is now obvious.
T(2^k) = 3^{k+1}2^0 + 3^{k-1}2^1 + 3^{k-2}2^2 + \ldots + 3^12^{k-1} + 3^{0}2^k.

= \sum 3^{k-i}2^i

= 3^k \sum (2/3)^i

= 3^k \left[ \left( \frac{1 - (2/3)^{k+1}}{1 - (2/3)} \right) \right]

= 3^{k+1} - 2^{k+1}

**Proposition: (Geometric Series)**

Let \( S_n \) be the sum of the first \( n \) terms of the geometric series \( a, ar, ar^2 \ldots \) Then

\[ S_n = \frac{a(1-r^n)}{1-r}, \text{ except in the special case when } r = 1; \text{ when } S_n = a_n. \]

\[ = 3^k \left[ \left( \frac{1 - (2/3)^{k+1}}{1 - (2/3)} \right) \right] \]

\[ = 3^k \left[ \left( \frac{3^{k+1} - 2^{k+1}}{3^{k+1}} \right) \right] \]

\[ = 3^k \left[ \left( \frac{3^{k+1} - 2^{k+1}}{3^{k+1}} \right)/\left( \frac{3 - 2}{3} \right) \right] \]

\[ = 3^k \left[ \left( \frac{3^{k+1} - 2^{k+1}}{3^{k+1}} \right)/\left( \frac{3}{3} \right) \right] \]

\[ = 3^k \left[ \left( \frac{3^{k+1} - 2^{k+1}}{3^{k+1}} \right)/\left( \frac{3}{3} \right) \right] \]

\[ = 3^{k+1} - 2^{k+1} \]

* It is easy to check this formula against our earlier tabulation.

\[ \text{EG : 2} \]

\[ t_n = \begin{cases} 0 & n=0 \\ 5 & n=1 \\ 3t_{n-1} + 4t_{n-2}, \text{ otherwise} \end{cases} \]

\[ t_n = 3t_{n-1} - 4t_{n-2} = 0 \quad \rightarrow \text{General function} \]

Characteristics Polynomial, \( x^2 - 3x - 4 = 0 \)
\( (x - 4)(x + 1) = 0 \)

Roots \( r_1 = 4, r_2 = -1 \)
General Solution, \( f_n = C_1 r_1^n + C_2 r_2^n \) \( \Rightarrow (A) \)

\[
\begin{align*}
  n=0 & \Rightarrow C_1 + C_2 = 0 \quad \Rightarrow (1) \\
  n=1 & \Rightarrow C_1 r_1 + C_2 r_2 = 5 \quad \Rightarrow (2)
\end{align*}
\]

Eqn 1 \( \Rightarrow C_1 = -C_2 \)

Sub \( C_1 \) value in Eqn (2)

\[-C_2 r_1 + C_2 r_2 = 5 \]

\[C_2(r_2 - r_1) = 5\]

\[C_2 = \frac{5}{r_2 - r_1} \]

\[= \frac{-1 + 4}{-1 + 4} = -1\]

\[C_2 = -1, \quad C_1 = 1\]

Sub \( C_1, C_2, r_1 \) & \( r_2 \) value in equation \( \Rightarrow (A) \)

\[
\begin{align*}
  f_0 &= 1. 4^0 + (-1) . (-1)^0 \\
  f_n &= 4^n + 1^n
\end{align*}
\]

2. **Homogenous Recurrences**:

* We begin our study of the technique of the characteristic equation with the resolution of homogenous linear recurrences with constant co-efficient, i.e the recurrences of the form,

\[
a_0 t_n + a_1 t_{n-1} + \ldots + a_k t_{n-k} = 0
\]

where the \( t_i \) are the values we are looking for.

* The values of \( t_i \) on ‘K’ values of \( i \) (Usually \( 0 \leq i \leq k-1 \) (or) \( 0 \leq i \leq k \)) are needed to determine the sequence.

* The initial condition will be considered later.

* The equation typically has infinitely many solution.

* The recurrence is,

  \( \Rightarrow \) linear because it does not contain terms of the form \( t_n \), \( t_{n-1} \) \( t_{n-2} \), and soon.

  \( \Rightarrow \) homogeneous because the linear combination of the \( t_n \) is equal to zero.

  \( \Rightarrow \) With constant co-efficient because the \( a_i \) are constants
Consider for instance our non familiar recurrence for the Fibonacci sequence,
\[ f_n = f_{n-1} + f_{n-2} \]

This recurrence easily fits the mould of equation after obvious rewriting.
\[ f_n - f_{n-1} - f_{n-2} = 0 \]

Therefore, the fibonacci sequence corresponds to a homogenous linear recurrence with constant co-efficient with \( k=2, a_0=1, a_1=a_2 = -1 \).

In other words, if \( f_n \) & \( g_n \) satisfy equation.

\[ \sum_{i=0}^{k} a_i f_{n-i} = 0 \ & \text{similarly for } g_n \& f_n \]

We set \( t_n = C f_n + d g_n \) for arbitrary constants \( C \& d \), then \( t_n \) is also a solution to equation.

This is true because,
\[
\begin{align*}
  a_0 t_n + a_1 t_{n-1} + \ldots + a_k t_{n-k} & = a_0 (C f_n + d g_n) + a_1 (C f_{n-1} + d g_{n-1}) + \ldots + a_k (C f_{n-k} + d g_{n-k}) \\
  & = C(a_0 f_n + a_1 f_{n-1} + \ldots + a_k f_{n-k}) + d(a_0 g_n + a_1 g_{n-1} + \ldots + a_k g_{n-k}) \\
  & = C * 0 + d * 0 \\
  & = 0.
\end{align*}
\]

1) (Fibonacci) Consider the recurrence.

\[ f_n = \begin{cases} 
  n & \text{if } n=0 \text{ or } n=1 \\
  f_{n-1} + f_{n-2} & \text{otherwise}
\end{cases} \]

We rewrite the recurrence as,
\[ f_n - f_{n-1} - f_{n-2} = 0. \]

The characteristic polynomial is,
\[ x^2 - x - 1 = 0. \]

The roots are,
\[
\begin{align*}
  x & = \frac{-(-1) \pm \sqrt{((-1)^2 + 4)}}{2} \\
  & = \frac{1 \pm \sqrt{1 + 4}}{2}
\end{align*}
\]
\[
2 \\
1 \pm \sqrt{5} \\
= \frac{1 + \sqrt{5}}{2} \quad \text{and} \quad \frac{1 - \sqrt{5}}{2}
\]

The general solution is,
\[
f_n = C_1 r_1^n + C_2 r_2^n
\]

when \( n=0, \) \( f_0 = C_1 + C_2 = 0 \)
when \( n=1, \) \( f_1 = C_1 r_1 + C_2 r_2 = 1 \)

\[
\begin{align*}
C_1 + C_2 &= 0 \quad \rightarrow \ (1) \\
C_1 r_1 + C_2 r_2 &= 1 \quad \rightarrow \ (2)
\end{align*}
\]

From equation (1)
\[
C_1 = -C_2
\]

Substitute \( C_1 \) in equation(2)
\[
-C_2 r_1 + C_2 r_2 = 1 \\
C_2 [r_2 - r_1] = 1
\]

Substitute \( r_1 \) and \( r_2 \) values
\[
\frac{1 - \sqrt{5}}{2} \quad \frac{1 - \sqrt{5}}{2} = 1
\]
\[
\frac{2}{1 - \sqrt{5} - 1 - \sqrt{5}} = 1
\]
\[
\frac{2}{2} = 1
\]
\[
-C_2 * 2\sqrt{5} \\
\frac{-\sqrt{5}C_2}{2} = 1
\]

\[
C_1 = \frac{1}{\sqrt{5}} \quad C_2 = -\frac{1}{\sqrt{5}}
\]

Thus,
\[
1 \quad 1 + \sqrt{5} \quad n \quad -1 \quad 1 - \sqrt{5} \quad n
\]
\[ f_n = \frac{\sqrt{5}}{2} \left( \frac{1 + \sqrt{5}}{2} \right)^n - \frac{1}{2} \frac{\sqrt{5}}{2} \left( \frac{1 - \sqrt{5}}{2} \right)^n \]

3. **Inhomogeneous recurrence**:

* The solution of a linear recurrence with constant co-efficient becomes more difficult when the recurrence is not homogeneous, that is when the linear combination is not equal to zero.
* Consider the following recurrence
  \[ a_0 t_n + a_1 t_{n-1} + \ldots + a_k t_{n-k} = b^n p(n) \]
* The left hand side is the same as before, (homogeneous) but on the right-hand side we have \( b^n p(n) \), where,
  \( \rightarrow b \) is a constant
  \( \rightarrow p(n) \) is a polynomial in ‘n’ of degree ‘d’.

**Example(1)**:

Consider the recurrence,
\[ t_n - 2t_{n-1} = 3^n \rightarrow (A) \]

In this case, \( b=3 \), \( p(n) = 1 \), degree = 0.

The characteristic polynomial is,
\[ (x - 2)(x - 3) = 0 \]

The roots are, \( r_1 = 2, r_2 = 3 \)

The general solution,
\[ t_n = C_1 r_1^n + C_2 r_2^n \]
\[ t_n = C_1 2^n + C_2 3^n \rightarrow (1) \]

when \( n=0 \), \( C_1 + C_2 = t_0 \rightarrow (2) \)
when \( n=1 \), \( 2C_1 + 3C_2 = t_1 \rightarrow (3) \)

sub \( n=1 \) in eqn (A)
\[ t_1 - 2t_0 = 3 \]
\[ t_1 = 3 + 2t_0 \]

substitute \( t_1 \) in eqn(3),
\[ (2) \times 2 \rightarrow 2C_1 + 2C_2 = 2t_0 \]
\[ 2C_1 + 3C_2 = (3 + 2t_0) \]
\(-C_2 = -3 \Rightarrow C_2 = 3\)

Substituting \(C_2 = 3\) in eqn (2)
\[
C_1 + C_2 = t_0 \\
C_1 + 3 = t_0 \\
C_1 = t_0 - 3
\]

Therefore \(t_n = (t_0 - 3)2^n + 3 \cdot 3^n\)
\[
= \text{Max}[O((t_0 - 3)2^n], O[3.3^n]] \\
= \text{Max}[O(2^n), O(3^n)] \text{ constants} \\
= O[3^n]
\]

Example : (2)
\[
t_n - 2t_{n-1} = (n + 5)3^n, \ n \geq 1 \rightarrow (A)
\]
This is Inhomogeneous

In this case, \(b=3\), \(p(n) = n+5\), degree = 1

So, the characteristic polynomial is,
\[
(x-2)(x-3)^2 = 0
\]

The roots are,
\[
r_1 = 2, \ r_2 = 3, \ r_3 = 3
\]

The general equation,
\[
t_n = C_1r_1^n + C_2 r_2^n + C_3nr_3^n \rightarrow (1)
\]
when \(n=0\), \(t_0 = C_1 + C_2 \rightarrow (2)
\]
when \(n=1\), \(t_1 = 2C_1 + 3C_2 + 3C_3 \rightarrow (3)

Substituting \(n=1\) in eqn(A),
\[
t_1 - 2t_0 = 6 \cdot 3 \Rightarrow (4)
\]
\[
t_1 - 2t_0 = 18 \\
t_1 = 18 + 2t_0
\]

Substituting \(t_1\) value in eqn(3)
\[
2C_1 + 3C_2 + 3C_3 = 18 + 2t_0 \Rightarrow (4)
\]
\[
C_1 + C_2 + C_3 = t_0 \Rightarrow (2)
\]

Sub. \(n=2\) in eqn(1)
\[
4C_1 + 9C_2 + 18C_3 = t_2 \Rightarrow (5)
\]

Sub \(n=2\) in eqn(A)
\[ t_2 - 2t_1 = 7.9 \]
\[ t_2 = 63 + 2t_1 \]
\[ = 63 + 2[18 + 2t_0] \]
\[ t_2 = 63 + 36 + 4t_0 \]
\[ t_2 = 99 + 4t_0 \]

sub. \( t_2 \) value in eqn(3),
\[ 4C_1 + 9C_2 + 18C_3 = 99 + 4t_0 \rightarrow (5) \]

solve eqn (2),(4) & (5)

\[ n=0, \quad C_1 + C_2 = t_0 \quad \rightarrow (2) \]
\[ n=1, \quad 2C_1 + 3C_2 + 3C_3 = 18 + 2t_0 \rightarrow (4) \]
\[ n=2, \quad 4C_1 + 9C_2 + 18C_3 = 99 + 4t_0 \rightarrow (5) \]

\[ (4) \times 6 \rightarrow 12C_1 + 18C_2 + 18C_3 = 108 + 2t_0 \rightarrow (4) \]
\[ (5) \rightarrow 4C_1 + 9C_2 + 18C_3 = 99 + 4t_0 \rightarrow (5) \]

\[ \frac{8C_1 + 9C_2}{8C_1 + 9C_2} = 9 + 8t_0 \rightarrow (6) \]

\[ (2) \times 8 \rightarrow 8C_1 + 8C_2 = 8t_0 \quad \rightarrow (2) \]
\[ (6) \rightarrow 8C_1 + 9C_2 = 9 + 8t_0 \rightarrow (6) \]

\[ -C_2 = -9 \]
\[ C_2 = 9 \]

Sub, \( C_2 = 9 \) in eqn(2)
\[ C_1 + C_2 = t_0 \]
\[ C_1 + 9 = t_0 \]
\[ C_1 = t_0 - 9 \]

Sub \( C_1 \) & \( C_2 \) in eqn (4)
\[ 2C_1 + 3C_2 + 3C_3 = 18 + 2t_0 \]
\[ 2(t_0 - 9) + 3(9) + 3C_3 = 18 + 2t_0 \]
\[ 2t_0 - 18 + 27 + 3C_3 = 18 + 2t_0 \]
\[ 2t_0 + 9 + 3C_3 = 18 + 2t_0 \]
\[ 3C_3 = 18 - 9 + 2t_0 - 2t_0 \]
\[ 3C_3 = 9 \]
\[ C_3 = 9/3 \]
\[ C_3 = 3 \]

Sub. \( C_1, C_2, C_3, r_1, r_2, r_3 \) values in eqn (1)
\[ t_n = C_12^n + C_23^n + C_3n.3^n \]
\[ = (t_0 - 9)2^n + 9.3^n + 3n.3^n \]
\[ = \text{Max}[O[(t_0 - 9), 2^n], O[9.3^n], O[3.n.3^n]] \]
\[= \text{Max}[O(2^n), O(3^n), O(n3^n)]\]

\[t_n = O(n3^n)\]

**Example: (3)**

Consider the recurrence,

\[\begin{align*}
    t_n &= 1 & \text{if } n=0 \\
    &= 4t_{n-1} - 2^n & \text{otherwise}
\end{align*}\]

\[t_n - 4t_{n-1} = -2^n \rightarrow (A)\]

In this case, \(c=2, p(n) = -1, \text{degree} = 0\)

\((x-4)(x-2) = 0\)

The roots are, \(r_1 = 4, r_2 = 2\)

The general solution,

\[t_n = C_14^n + C_22^n \rightarrow (1)\]

when \(n=0\), in (1) \(\Rightarrow C_1 + C_2 = 1 \rightarrow (2)\)

when \(n=1\), in (1) \(\Rightarrow 4C_1 + 2C_2 = t_1 \rightarrow (3)\)

sub \(n=1\) in (A),

\[t_n - 4t_{n-1} = -2^n\]

\[t_1 - 4t_0 = -2\]

\[t_1 = 4t_0 - 2 \text{ [since } t_0 = 1\]

\[t_1 = 2\]

sub \(t_1\) value in eqn (3)

\[4C_1 + 2C_2 = 4t_0 - 2 \rightarrow (3)\]

\((2) \times 4 \Rightarrow 4C_1 + 4C_2 = 4\)

\[\begin{align*}
    -2C_2 &= 4t_0 - 6 \\
    &= 4(1) - 6 \\
    &= -2 \\
    C_2 &= 1
\end{align*}\]

\[\begin{align*}
    -2C_2 &= 4t_0 - 6 \\
    2C_2 &= 6 - 4t_0 \\
    C_2 &= 3 - 2t_0 \\
    3 - 2(1) &= 1 \\
    C_2 &= 1
\end{align*}\]
Sub. \( C_2 \) value in eqn(2),

\[
\begin{align*}
C_1 + C_2 &= 1 \\
C_1 + (3-2t_0) &= 1 \\
C_1 + 3 - 2t_0 &= 1 \\
C_1 &= 1 - 3 + 2t_0 \\
C_1 &= 2t_0 - 2 \\
&= 2(1) - 2 = 0 \\
C_1 &= 0
\end{align*}
\]

Sub \( C_1 \) & \( C_2 \) value in eqn (1)

\[
\begin{align*}
t_n &= C_1 4^n + C_2 2^n \\
&= \text{Max}[O(2t_0 - 2).4^n, O(3 - 2t_0).2^n] \\
&= O(2^n)
\end{align*}
\]

Example : (4)

\[
\begin{align*}
t_n = 0 & \text{ if } n=0 \\
2t_{n-1} + n + 2^n & \text{ otherwise}
\end{align*}
\]

\[
t_n - 2t_{n-1} = n + 2^n \rightarrow (A)
\]

There are two polynomials.

For \( n; b=1, p(n), \text{degree} = 1 \)

For \( 2n; b=2, p(n) = 1, \text{degree} = 0 \)

The characteristic polynomial is,

\[
(x-2)(x-1)^2(x-2) = 0
\]

The roots are, \( r_1 = 2, r_2 = 2, r_3 = 1, r_4 = 1 \).

So, the general solution,

\[
t_n = C_1 r_1^n + C_2 n r_2^n + C_3 r_3^n + C_4 n r_4^n
\]

sub \( r_1, r_2, r_3 \) in the above eqn

\[
t_n = 2^n C_1 + 2^n C_2 n + C_3 . 1^n + C_4 . n . 1^n \rightarrow (1)
\]

sub. \( n=0 \rightarrow C_1 + C_3 = 0 \rightarrow (2) \)

sub. \( n=1 \rightarrow 2C_1 + 2C_2 + C_3 + C_4 = t_1 \rightarrow (3) \)

sub. \( n=1 \) in eqn (A)

\[
\begin{align*}
t_n - 2t_{n-1} &= n + 2^n \\
t_1 - 2t_0 &= 1 + 2 \\
t_1 - 2t_0 &= 3 \\
t_1 &= 3 \text{ [since } t_0 = 0]\end{align*}
\]
sub. n=2 in eqn (1)
\[ 2^2 C_1 + 2. 2^2 C_2 + C_3 + 2 C_4 = t_2 \]
\[ 4 C_1 + 8 C_2 + C_3 + 2 C_4 = t_2 \]

sub n=2 in eqn (A)
\[ t_2 - 2t_1 = 2 + 2^2 \]
\[ t_2 - 2t_1 = 2 + 4 \]
\[ t_2 - 2t_1 = 6 \]
\[ t_2 = 6 + 2t_1 \]
\[ t_2 = 6 + 2.3 \]
\[ t_2 = 6 + 6 \]
\[ t_2 = 12 \]

\[ 4 C_1 + 8 C_2 + C_3 + 2 C_4 = 12 \rightarrow (4) \]

sub n=3 in eqn (!)
\[ 2^3 C_1 + 2. 2^3 C_2 + C_3 + 3 C_4 = t_3 \]
\[ 3 C_1 + 24 C_2 + C_3 + 3 C_4 = t_3 \]

sub n=3 in eqn (A)
\[ t_3 - 2t_2 = 3 + 2^3 \]
\[ t_3 - 2t_2 = 3 + 8 \]
\[ t_3 - 2t_2 = 11 \]
\[ t_3 - 2t_2 = 11 \]
\[ t_3 = 11 + 24 \]
\[ t_3 = 35 \]

\[ 8 C_1 + 24 C_2 + C_3 + 3 C_4 = 35 \rightarrow (5) \]

n=0, solve;
\[ C_1 + C_3 = 0 \rightarrow (2) \]
\[ n=1, (2), (3), (4)&(5) \]
\[ 2 C_1 + 2 C_2 + C_3 + C_4 = 3 \rightarrow (3) \]
\[ n=2, \]
\[ 4 C_1 + 8 C_2 + C_3 + 2 C_4 = 12 \rightarrow (4) \]
\[ n=3, \]
\[ 8 C_1 + 24 C_2 + C_3 + 3 C_4 = 35 \rightarrow (5) \]

\[ -4 C_1 - 16 C_2 - C_4 = -23 \rightarrow (6) \]

solve: (2) & (3)
(2) \[ C_1 + C_3 = 0 \]
(3) \[ 2 C_1 + C_3 + 2 C_2 + C_4 = 3 \]

\[ -C_1 - 2 C_2 - C_4 = -3 \rightarrow (7) \]

solve(6) & (7)
(6) \[ -4 C_1 - 16 C_2 - C_4 = -23 \]
4. Change of variables:
* It is sometimes possible to solve more complicated recurrences by making a change of variable.
* In the following example, we write \( T(n) \) for the term of a general recurrences, and \( t_i \) for the term of a new recurrence obtained from the first by a change of variable.

Example: (1)
Consider the recurrence,

\[
T(n) = \begin{cases} 
1 & \text{if } n=1 \\
3T(n/2) + n & \text{if } \text{‘}n\text{’ is a power of 2, } n>1
\end{cases}
\]

Reconsider the recurrence we solved by intelligent guesswork in the previous section, but only for the case when ‘\( n \)’ is a power of 2

\[
T(n) = \\
3T(n/2) + n
\]

* We replace ‘\( n \)’ by \( 2^i \).
* This is achieved by introducing new recurrence \( t_i \), define by \( t_i = T(2^i) \)
* This transformation is useful because \( n/2 \) becomes \( (2^i)/2 = 2^{i-1} \)
* In other words, our original recurrence in which \( T(n) \) is defined as a function of \( T(n/2) \) given way to one in which \( t_i \) is defined as a function of \( t_{i-1} \), precisely the type of recurrence we have learned to solve.

\[
t_i = T(2^i) = 3T(2^{i-1}) + 2^i \\
t_i = 3t_{i-1} + 2^i \\
t_i - 3t_{i-1} = 2^i \rightarrow (A)
\]

In this case,
\( b = 2, p(n) = 1, \) degree = 0

So, the characteristic equation,

\( (x - 3)(x - 2) = 0 \)

The roots are, \( r_1 = 3, r_2 = 2. \)

The general equation,

\[
t_n = C_1 r_1^i + C_2 r_2^i
\]

sub. \( r_1 \) & \( r_2 \): \( t_n = 3^n C_1 + C_2 2^n \)
\[ t_n = C_1 3^i + C_2 2^i \]

We use the fact that, \( T(2^i) = t_i \) & thus \( T(n) = \log n \) when \( n = 2^i \) to obtain,
\[ T(n) = C_1 \cdot \log_2 n + C_2 \cdot 2 \log_2 n \]
\[ T(n) = C_1 \cdot n^{\log_2 3} + C_2 \cdot n \quad [i = \log n] \]

When ‘n’ is a power of 2, which is sufficient to conclude that,

\[ T(n) = O(n^{\log 3}) \quad \text{‘n’ is a power of} \ 2 \]

**Example: (2)**

Consider the recurrence,
\[ T(n) = 4T(n/2) + n^2 \rightarrow (A) \]

Where ‘n’ is a power of 2, \( n \geq 2 \).
\[ t_i = T(2^i) = 4T(2^{i-1}) + (2^i)^2 \]
\[ t_i = 4t_{i-1} + 2^i \]
\[ \Rightarrow t_i - 4t_{i-1} = 2^i \]

In this eqn,
\[ b = 4, \ P(n) = 1, \ \text{degree} = 0 \]

The characteristic polynomial,
\[ (x - 4)(x - 4) = 0 \]

The roots are, \( r_1 = 4, \ r_2 = 4 \).

So, the general equation,
\[ t_i = C_1 4^i + C_2 4^i \cdot i \quad [\text{since} \ i = \log n] \]
\[ = C_1 \cdot 4^{\log n} + C_2 \cdot 4^{\log n} \cdot \log n \quad [\text{since} \ 2^i = n] \]
\[ = C_1 \cdot n^{\log 4} + C_2 \cdot n^{\log 4} \cdot \log 1 \]

\[ T(n) = O(n^{\log 4}) \quad \text{‘n’ is the power of} \ 2. \]

**EXAMPLE : 3**

\[ T(n) = 2T(n/2) + \log n \]

When ‘n’ is a power of 2, \( n \geq 2 \)
\[ t_i = T(2^i) = 2T(2^{i/2}) + 2^i \cdot i \quad [\text{since} \ 2^i = n; \ i = \log n] \]
\[ t_i - 2t_{i-1} = i \cdot 2^i \]

In this case,
\[ b = 2, \ P(n) = i, \ \text{degree} = 1 \]
\[ (x - 2)(x - 2)^2 = 0 \]
The roots are, \( r_1 = 2, r_2 = 2, r_3 = 2 \)

The general solution is,
\[
t_n = C_1 2^i + C_2 \cdot 2^i \cdot i + C_3 \cdot 2^i
\]
\[= nC_1 + nC_2 + nC_3 (\log n^2n)\]

\( t_n = O(n \log^2 n) \)

**Example: 4**

\[
T(n) = \begin{cases} 2, & n = 1 \\ 5T(n/4) + Cn^2, & n > 1 \end{cases}
\]

\[
t_i = T(4^i) = 5T(4^{i-1}) + C(4^i)^2
= 5T(4^{i-1}) + C \cdot 16^i
= 5t_{i-1} + C \cdot 16^i
\]

\[t_i - 5t_{i-1} = C \cdot 16^i\]

In this case,
\[b = 16, P(n) = 1, \text{degree} = 0\]

The characteristic eqn,
\[(x - 5)(x - 16) = 0\]

The roots are, \( r_1 = 5, r_2 = 16 \)

The general solution,
\[
t_i = C_1 \cdot 5^i + C_2 \cdot 16^i
= C_1 \cdot 5^i + C_2 \cdot (4^2)^i
\]

\( t_n = O(n^2) \)

**EXAMPLE: 5**

\[
T(n) = \begin{cases} 2, & n = 1 \\ T(n/2) + Cn, & n > 1 \end{cases}
\]

\[
T(n) = T(n/2) + Cn
= T(2^{i/2}) + C \cdot 2^i
= T(2^{i-1}) + C \cdot 2^i
\]

\[t_i = t_{i-1} + C \cdot 2^i
\]

\[t_i - t_{i-1} = C \cdot 2^i\]
In this case, \( b=2, \ P(n) =1, \) degree =0

So, the characteristic polynomial,

\[(x -1)(x - 2) = 0\]

The roots are, \( r_1 = 1, \ r_2 = 2 \)

\[t_i = C_1 \cdot 1^i + c_2 \cdot 2^i\]

\[
= C_1 \cdot 1^{\log_2 n} + C_2 \cdot n
\]

\[
= C_1 \cdot n^{\log_2 1} + C_2 \cdot n
\]

\[t_n = O(n)\]

EXAMPLE: 6

\[
T(n) = \begin{cases} 1 & n = 1 \\ 3T(n/2) + n & n \text{ is a power of } 2 \end{cases}
\]

\[t_i = T(2^i) = 3T(2^{i/2}) + 2^i \]

\[
= 3T(2^{i-1}) + 2^i
\]

\[t_i = 3t_{i-1} + 2^i\]

So, \( b = 2, \ P(n) =1, \) degree =0

\[(x - 3)(x - 2) = 0\]

The roots are, \( r_1 = 3, \ r_2 = 2 \)

\[t_i = C_1 \cdot 3^i + C_2 \cdot 2^i\]

\[
= C_1 \cdot n^{\log_3 3} + C_2 \cdot n^{\log_2 2} = 1
\]

\[
= C_1 \cdot n^{\log_3 3} + C_2 \cdot n
\]

\[t_n = O(n^{\log_3 3})\]

EXAMPLE: 7

\[T(n) = 2T(n/2) + n \cdot \log n\]

\[t_i = T(2^i) = 2T(2^{i/2}) + 2^i \cdot i\]

\[
= 2T(2^{i-1}) + i \cdot 2^i
\]

\[t_i - 2t_{i-1} = i \cdot 2^i\]

\( \Rightarrow b=2, \ P(n) = 1, \) degree = 1

The roots is \( (x - 2)(x - 2) = 0 \)

\[x = 2, 2, 2\]
General solution,

\[ t_n = C_1 2^i + C_2 i 2^i + C_3 2^i i \]

\[ = C_1 2^i + C_2 2^i i + C_3 i^2 2^i \]

\[ = C_1 n + C_2 n \log_2 n + C_3 i^2 n \]

\[ = C_1 n + C_2 n \log_2 n + C_3 (2 \log_2 n) n \]

\[ t_n = O(n \cdot 2 \log_2 n) \]

5. Range Transformation:

* When we make a change of variable, we transform the domain of the recurrence.
* Instead it may be useful to transform the range to obtain a recurrence in a form that we know how to solve.
* Both transformation can be sometimes be used together.

EXAMPLE: 1

Consider the following recurrence, which defines \( T(n) \) where 'n' is the power of 2

\[ T(n) = \begin{cases} 
1/3, & \text{if } n=1 \\
nT^2(n/2), & \text{otherwise} 
\end{cases} \]

The first step is a change of variable,

Let \( t_i \) denote \( T(2^i) \)

\[ t_i = T(2^i) = 2^i T^2(2^{i-1}) \]

\[ = 2^i t_{i-1} \]

* This recurrence is not clear, furthermore the co-efficient \( 2^i \) is not a constant.
* To transform the range, we create another recurrence by using \( u_i \) to denote \( \lg t_i \)

\[ u_i = \lg t_i = i + 2 \lg t_{i-1} \]

\[ = i + 2u_{i-1} \]

\[ \Rightarrow u_i - 2u_{i-1} = i \]

\[ (x - 2)(x - 1)^2 = 0 \]

The roots are, \( x = 2, 1, 1. \)

G.S,

\[ u_i = C_1 2^i + C_2 1^i + C_3 i 1^i \]

Sub. This solution into the recurrence,

For \( u_i \) yields,

\[ i = u_i - 2u_{i-1} \]

\[ = C_1 2^i + C_2 + C_3 i - 2(C_1 2^{i-1} + C_2 + C_3 (i-1)) \]
\[ (2C_3 - C_2) - C_3i. \]

\[ C_3 = -1 \text{ and } C_2 = 2C_3 = -2 \]

\[ u_i = C_1 2^i - i - 2 \]

This gives us the G.S for \( t_i \) & \( T(n) \)

\[ t_i = 2^{u_i} = 2^{C_1 2^i - i - 2} \]
\[ T(n) = t_{\text{lg}n} = 2^{C_1 n - \log n - 2} = 2^{C_1 n} / 4n \]

We use the initial condition \( T(1) = 1/3 \)

To determine \( C_1 \): \( T(1) = 2C_1 / 4 = 1/3 \)

Implies that \( C_1 = \lg(4/3) = 2 - \log 3 \)

The final solution is

\[ T(n) = 2^{2n} / 4n \cdot 3^n \]

1. Newton Raphson method: \( x_2 = x_1 - f(x_1) / f'(x_1) \)

**SEARCHING**

Let us assume that we have a sequential file and we wish to retrieve an element matching with key ‘k’, then, we have to search the entire file from the beginning till the end to check whether the element matching \( k \) is present in the file or not.

There are a number of complex searching algorithms to serve the purpose of searching. The linear search and binary search methods are relatively straightforward methods of searching.

**Sequential search:**

In this method, we start to search from the beginning of the list and examine each element till the end of the list. If the desired element is found we stop the search and return the index of that element. If the item is not found and the list is exhausted the search returns a zero value.

In the worst case the item is not found or the search item is the last (\( n^{th} \)) element. For both situations we must examine all \( n \) elements of the array so the order of magnitude or complexity of the sequential search is \( n \) i.e., \( O(n) \). The execution time for this algorithm is proportional to \( n \) that is the algorithm executes in linear time.

The algorithm for sequential search is as follows,

**Algorithm** : sequential search

**Input** : A, vector of \( n \) elements

K, search element
Output : j – index of k
Method : i=1
While(i<=n)
{ 
  if(A[i]=k)
  {
    write("search successful")
    write(k is at location i)
    exit();
  } 
  else
  i++ 
if end
while end
write (search unsuccessful);
algorithm ends.

Binary search:

Binary search method is also relatively simple method. For this method it is necessary to have the vector in an alphabetical or numerically increasing order. A search for a particular item with X resembles the search for a word in the dictionary. The approximate mid entry is located and its key value is examined. If the mid value is greater than X, then the list is chopped off at the (mid-1)^th location. Now the list gets reduced to half the original list. The middle entry of the left-reduced list is examined in a similar manner. This procedure is repeated until the item is found or the list has no more elements. On the other hand, if the mid value is lesser than X, then the list is chopped off at (mid+1)^th location. The middle entry of the right-reduced list is examined and the procedure is continued until desired key is found or the search interval is exhausted.

The algorithm for binary search is as follows,

Algorithm : binary search
Input : A, vector of n elements
K, search element
Output : low – index of k
Method : low=1,high=n
While(low<=high-1)
{ 
  mid=(low+high)/2
  if(k<a[mid])
    high=mid
  else
    low=mid
if end
}
while end
if(k=A[low])
{
    write("search successful")
    write(k is at location low)
    exit();
}
else
    write (search unsuccessful);
if end;
algorithmd ends.

SORTING

One of the major applications in computer science is the sorting of information in a table. Sorting algorithms arrange items in a set according to a predefined ordering relation. The most common types of data are string information and numerical information. The ordering relation for numeric data simply involves arranging items in sequence from smallest to largest and from largest to smallest, which is called ascending and descending order respectively. The items in a set arranged in non-decreasing order are {7,11,13,16,16,19,23}. The items in a set arranged in descending order is of the form {23,19,16,16,13,11,7} Similarly for string information, {a, abacus, above, be, become, beyond} is in ascending order and { beyond, become, be, above, abacus, a} is in descending order. There are numerous methods available for sorting information. But, not even one of them is best for all applications. Performance of the methods depends on parameters like, size of the data set, degree of relative order already present in the data etc.

Selection sort:

The idea in selection sort is to find the smallest value and place it in an order, then find the next smallest and place in the right order. This process is continued till the entire table is sorted.

Consider the unsorted array,

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>35</td>
<td>18</td>
</tr>
<tr>
<td>8</td>
<td>14</td>
<td>41</td>
</tr>
<tr>
<td>3</td>
<td>39</td>
<td></td>
</tr>
</tbody>
</table>

The resulting array should be

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>8</td>
<td>14</td>
</tr>
<tr>
<td>18</td>
<td>20</td>
<td>35</td>
</tr>
<tr>
<td>39</td>
<td>41</td>
<td></td>
</tr>
</tbody>
</table>
One way to sort the unsorted array would be to perform the following steps:

- Find the smallest element in the unsorted array
- Place the smallest element in position of a[1]

i.e., the smallest element in the unsorted array is 3 so exchange the values of a[1] and a[7]. The array now becomes,

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>35</td>
<td>18</td>
</tr>
</tbody>
</table>
Now find the smallest from a[2] to a[8], i.e., 8 so exchange the values of a[2] and a[4] which results with the array shown below,

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>8</td>
<td>18</td>
</tr>
<tr>
<td>35</td>
<td>14</td>
<td>41</td>
</tr>
<tr>
<td>20</td>
<td>39</td>
<td></td>
</tr>
</tbody>
</table>

Repeat this process until the entire array is sorted. The changes undergone by the array is shown in fig 2.2. The number of moves with this technique is always of the order O(n).

**Bubble sort:**

Bubble Sort is an elementary sorting algorithm. It works by repeatedly exchanging adjacent elements, if necessary. When no exchanges are required, the file is sorted.

**BUBBLESORT (A)**
for \( i \leftarrow 1 \) to length \([A]\) do
\hspace{1cm} for \( j \leftarrow \) length \([A]\) downto \( i+1 \) do
\hspace{1cm} If \( A[A] < A[j-1] \) then
\hspace{1cm} Exchange \( A[j] \leftrightarrow A[j-1] \)

Here the number of comparison made
\[ 1 + 2 + 3 + \ldots + (n-1) = n(n-1)/2 = O(n^2) \]

Another well known sorting method is bubble sort. It differs from the selection sort in that instead of finding the smallest record and then performing an interchange two records are interchanged immediately upon discovering that they are of out of order. When this apporoach is used there are at most \( n-1 \) passes required. During the first pass \( k1 \) and \( k2 \) are compared, and if they are out of order, then records \( R1 \) AND \( R2 \) are interchanged; this process is repeated for records \( R2 \) and \( R3 \), \( R3 \) and \( R4 \) and so on .this method will cause with small keys to bubble up. After the first pass the record with the largest key will be in the nth position. On each successive pass, the records with the next largest key will be placed in the position \( n-1 \), \( n-2 \) …2 respectively, thereby resulting in a sorted table.

After each pass through the table, a check can be made to determine whether any interchanges were made during that pass. If no interchanges occurred then the table must be sorted and no further passes are required.

**Insertion sort :**

Insertion sort is a straight forward method that is useful for small collection of data. The idea here is to obtain the complete solution by inserting an element from the unordered part into the partially ordered solution extending it by one element. Selecting an element from the unordered list could be simple if the first element of that list is selected.

<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>20 35 18 8 14 41 3 39</td>
</tr>
</tbody>
</table>

Initially the whole array is unordered. So select the minimum and put it in place of \( a[1] \) to act as sentinel. Now the array is of the form,

<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>3 35 18 8 14 41 20 39</td>
</tr>
</tbody>
</table>

Now we have one element in the sorted list and the remaining elements are in the unordered set. Select the next element to be inserted. If the selected element is less than the preceding element move the preceding element by one position and insert the smaller element.
In the above array the next element to be inserted is \( x = 35 \), but the preceding element is 3 which is less than \( x \). Hence, take the next element for insertion i.e., 18. 18 is less than 35, so move 35 one position ahead and place 18 at that place. The resulting array will be,

\[
\begin{array}{cccccccc}
3 & 18 & 35 & 8 & 14 & 41 & 20 & 39
\end{array}
\]

Now the element to be inserted is 8. 8 is less than 35 and 8 is also less than 18 so move 35 and 18 one position right and place 8 at \( a[2] \). This process is carried till the sorted array is obtained.

The changes undergone are shown in fig 2.3.

The disadvantages of the insertion sort method is the amount of movement of data. In the worst case, the number of moves is of the order \( O(n^2) \). For lengthy records it is quite time consuming.

**Heaps and Heap sort:**

A heap is a complete binary tree with the property that the value at each node is at least as large as the value at its children.

The definition of a max heap implies that one of the largest elements is at the root of the heap. If the elements are distinct then the root contains the largest item. A max heap can be implemented using an array \( \text{an}[] \).

To insert an element into the heap, one adds it "at the bottom" of the heap and then compares it with its parent, grandparent, great grandparent and so on, until it is less than or equal to one of these values. Algorithm insert describes this process in detail.
Algorithm Insert(a,n)
{
// Insert a[n] into the heap which is stored in a[1:n-1]
I=n;
item=a[n];
while( (I>n) and (a[I!/2] < item)) do
{
a[I] = a[I/2];
I=I/2;
}
a[I]=item;
return (true);
}

The figure shows one example of how insert would insert a new value into an existing heap of five elements. It is clear from the algorithm and the figure that the time for insert can vary. In the best case the new elements are correctly positioned initially and no new values need to be rearranged. In the worst case the number of executions of the while loop is proportional to the number of levels in the heap. Thus if there are n elements in the heap, inserting new elements takes $O(\log n)$ time in the worst case.

To delete the maximum key from the max heap, we use an algorithm called Adjust. Adjust takes as input the array a[ ] and integer I and n. It regards a[1..n] as a complete binary
If the subtrees rooted at $2I$ and $2I+1$ are max heaps, then adjust will rearrange elements of $a[I]$ such that the tree rooted at $I$ is also a max heap. The maximum elements from the max heap $a[1..n]$ can be deleted by deleting the root of the corresponding complete binary tree. The last element of the array, i.e. $a[n]$, is copied to the root, and finally we call Adjust($a, 1, n-1$).

**Algorithm Adjust** ($a, I, n$)

```plaintext
j = 2I;
item = a[I];
while ($j <= n$) do
    { if (($j <= n$) and ($a[j] < a[j+1]$)) then
        j = j + 1;
    // compare left and right child and let $j$ be the right
    // child
    if (item >= a[I]) then break;
        // a position for item is found
    a[i/2] = a[j];
    j = 2I;
    }
    a[j/2] = item;
    }
```

**Algorithm Delmac** ($a, n, x$)

// Delete the maximum from the heap $a[1..n]$ and store it in $x$
```plaintext
if (n = 0) then
    { write('heap is empty');
      return (false);
    }
x = a[1];
a[1] = a[n];
Adjust(a, 1, n-1);
Return(true);
```

Note that the worst case run time of adjust is also proportional to the height of the tree. Therefore if there are $n$ elements in the heap, deleting the maximum can be done in $O(\log n)$ time.

To sort $n$ elements, it suffices to make $n$ insertions followed by $n$ deletions from a heap since insertion and deletion take $O(\log n)$ time each in the worst case this sorting algorithm has a complexity of $O(n \log \ n)$.

**Algorithm sort** ($a, n$)
{ 
  for i=1 to n 
  do 
    Insert(a,i); 
  for i= n to 1 step –1 do 
    { 
      Delmax(a,i, x); a[i]=x; 
    } 
}
Objective Questions.

1. In analysis of algorithm, approximate relationship between the size of the job and the amount of work required to do is expressed by using __________
   (a) Central tendency (b) Differential equation (c) Order of execution (d) Order of magnitude (e) Order of Storage.
   **Ans:** Order of execution

2. Worst case efficiency of binary search is
   (a) log2 n + 1 (b) n (c) N2 (d) 2n (e) log n.
   **Ans:** log2 n + 1

3. For analyzing an algorithm, which is better computing time?
   (a) O (100 Log N) (b) O (N) (c) O (2N) (d) O (N logN) (e) O (N2).
   **Ans:** O (100 Log N)

4. Consider the usual algorithm for determining whether a sequence of parentheses is balanced. What is the maximum number of parentheses that will appear on the stack AT ANY ONE TIME when the algorithm analyzes: ( ))(()())
   (a) 1 (b) 2 (c) 3 (d) 4
   **Ans:** 3

5. Breadth first search __________
   (a) Scans each incident node along with its children. (b) Scans all incident edges before moving to other node. (c) Issame as backtracking (d) Scans all the nodes in random order.
   **Ans:** Scans all incident edges before moving to other node

6. Which method of traversal does not use stack to hold nodes that are waiting to be processed?
   (a) Dept First (b) D-search (c) Breadth first (d) Back-tracking
   **Ans:** Breadth first

7. The Knapsack problem where the objective function is to minimize the profit is _______
   (a) Greedy (b) Dynamic 0 / 1 (c) Back tracking (d) Branch & Bound 0/1
   **Ans:** Branch & Bound 0/1

8. Choose the correct answer for the following statements:
   I. The theory of NP-completeness provides a method of obtaining a polynomial time for NP algorithms.
   II. All NP-complete problem are NP-Hard.
     (a) I is FALSE and II is TRUE (b) I is TRUE and II is FALSE (c) Both are TRUE (d) Both are FALSE
   **Ans:** I is FALSE and II is TRUE

9. If all c(i, j )’s and r(i, j ’s are calculated, then OBST algorithm in worst case takes one of the following time.
   (a) O(n log n) (b) O(n3) (c) O(n2) (d) O(log n) (e) O(n4).
   **Ans:** O(n3)

10. The upper bound on the time complexity of the nondeterministic sorting algorithm is
   (a) O(n) (b) O(n log n) (c) O(1) (d) O( log n) (e) O(n2).
    **Ans:** O(n)
11. The worst case time complexity of the nondeterministic dynamic knapsack algorithm is
(a) O(n log n) (b) O(\log n) (c) O(n^2) (d) O(n) (e) O(1).
   Ans: O(n)

12. Recursive algorithms are based on
   (a) Divide and conquer approach (b) Top-down approach (c) Bottom-up approach (d) Hierarchical approach
   Ans: Bottom-up approach

13. What do you call the selected keys in the quick sort method?
   (a) Outer key (b) Inner Key (c) Partition key (d) Pivot key (e) Recombine key.
   Ans: c

14. How do you determine the cost of a spanning tree?
   (a) By the sum of the costs of the edges of the tree (b) By the sum of the costs of the edges and vertices of the tree
   (c) By the sum of the costs of the vertices of the tree (d) By the sum of the costs of the edges of the graph
   (e) By the sum of the costs of the edges and vertices of the graph.
   Ans: By the sum of the costs of the edges of the tree

15. The time complexity of the normal quick sort, randomized quick sort algorithms in the worst case is
   (a) O(n^2), O(n log n) (b) O(n^2), O(n^2) (c) O(n log n), O(n^2) (d) O(n log n), O(n log n) (e) O(n log n), O(n log n).
   Ans: O(n^2), O(n^2)

16. Let there be an array of length ‘N’, and the selection sort algorithm is used to sort it, how many times a swap function is called to complete the execution?
   (a) N log N times (b) log N times (c) N^2 times (d) N-1 times (e) N times.
   Ans: N-1 times

17. The sorting method which is used for external sort is
   (a) Bubble sort (b) Quick sort (c) Merge sort (d) Radix sort (e) Selection sort.
   Ans: Radix sort

18. The graph colouring algorithm’s time can be bounded by
   (a) O(mn^m) (b) O(nm) (c) O(nm. 2^n) (d) O(nmn).
   Ans: O(nmn).

19. Sorting is not possible by using which of the following methods?
   (a) Insertion (b) Selection (c) Deletion (d) Exchange
   Ans: Deletion

20. What is the type of the algorithm used in solving the 8 Queens problem?
   (a) Backtracking (b) Dynamic (c) Branch and Bound (d) DandC
   Ans: Backtracking
**Important Questions.**

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Solve the recurrence relation ( T(n) = T(1), \ n=1 ) ( T(n) = T(n/2) + c, \ n&gt;1 ) and ( n ) is a power of ( 2 )</td>
<td>CO1</td>
<td>APPLY</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.</td>
<td>Solve the following recurrence relation ( T(n) = 2T(n/2) + 1 ), and ( T(1)=2 )</td>
<td>CO1</td>
<td>APPLY</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.</td>
<td>Solve the following recurrence relation ( T(n) = 7T(n/2) + cn^2 )</td>
<td>CO1</td>
<td>APPLY</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4.</td>
<td>Differentiate Breadth first search and depth first search</td>
<td>CO1</td>
<td>REMEMBER</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>