Introduction: What is an Algorithm, Algorithm Specification, Pseudo code Conventions, Recursive Algorithm, Performance Analysis, Space complexity, Time complexity, Amortized Complexity, Asymptotic Notation. Practical Complexities, Performance Measurement.

ALGORITHM

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.

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:

1. How to devise algorithms.
   - Creating an algorithm is an art which may never be fully automated.
   - This area of study concentrates on various design techniques that have proven to be useful in that they have often yielded good algorithms.
   - It becomes easier to devise new and useful algorithm, by mastering these design strategies.
2. How to validate algorithms.
   - After devising an algorithm, it is necessary to show that it computes the correct answer for all possible legal inputs. This process is referred as *algorithm validation.*
   - The purpose of this validation is to assure us that this algorithm works correctly in the programming language it will be written in.
3. How to analyze algorithms.
   - This field of study is called *analysis of algorithms.*
   - Execution of an algorithm requires computer's central processing unit (CPU) to perform operations and its memory to hold the program and data.
   - Analysis of algorithms or performance analysis refers to task of determining how much computing time and storage an algorithm requires.
   - This is a challenging area and requires great mathematical skill.
4. How to test algorithm.
   - Testing an algorithm requires two phases
   - Debugging : It is the process of executing programs on sample data sets to determine whether faulty results occur and, of so, to correct them.
   - Profiling : It is the process of executing a correct program on data sets and measuring the time and space it takes to compute the results.

ALGORITHM SPECIFICATION

Algorithm can be described in many ways.

- **Natural language like English**: When this way is chosen, we should ensure that resulting instructions are definite.
- **Graphic representations called flowchart**: This method will work well when the algorithm is small& simple.
- **Pseudo-code Method**: In this method, we should typically describe algorithms as program, which resembles language like C. The advantage of pseudo code over flowchart is that it is very much similar to the final program code. It requires less time and space to develop, and we can write it in our own way as there are no fixed rules.

Pseudo-Code Conventions:

1. Comments begin with // and continue until the end of line.
2. Blocks are indicated with matching braces **{** and **}**. A compound statement can be represented as a block. The body of the procedure also forms a block. The statements are delimited by **;** .
3. An identifier begins with a letter. The data type of variables, or a variable is local or global, depends on the context. we assume simple data types such as integer, float, char, boolean, and so on. Compound data types can be formed with **records**. Here is an example

```
node = record
       {    datatype_1    data1;
            datatype_2    data2;
            :
            :
            datatype_n    datan;
            node          *link;
       }
```

   link is a pointer to the record type *node*. The data items of a record can be accessed with -> and a period. For example if *p* points to a record of type node, *p->data1* stands for the value of the first field in the record. If *q* is a record of type node, *q.data1* denote its first field.

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

   <variable> := < expression >;

5. There are two Boolean values **true** and **false**. In order to produce these values, the logical operators **and, or** and **not** and the relational operators **<, ≤, = , ≠, ≥,** and **>** are used.
6. Elements of the multidimensional arrays are accessed using [and]. For example, if A is a two dimensional array, the $(i, j)^{th}$ element of the array is denoted as A[i, j]. Array indices start at zero.
7. The general forms of looping statements are for, while and repeat-until**.**

   The general form of **while** loop is:

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

As long as <*condition*> is **true**, the statements get executed. The loop is exited whenever the condition becomes **false**.

The general form of for loop is

for *variable := value1* **to** *value2* **step** *st* **do**
{
<*statement 1*>
:
<*statement n*>
}

Here *value1*, *value2*, and *step* are arithmetic expressions. Step value can be either positive or negative. Default value of step is +1.

The general form of repeat-until statement is :

repeat
<*statement 1*>
:
<*statement n*>
until <*condition*>

The statements are executed as long as <*condition*> is false.

The **break** instruction can be used to exit out of loop.
The **return** statement is used to exit from function.

8. A conditional statement has the form

if <*condition*> **then** <*statement*>
if <*condition*> **then** <*statement1*> **else** <*statement2*>

The **case** statement takes the form

case
{
: <*condition 1*> ***:*** < *statement 1* >
:
: <*condition n*> ***:*** < *statement n* >
:else > ***:*** < *statement n+1* >
}

It is interpreted as if <*condition1*> is **true**, <*statement1*> gets executed and exit the case statement. if <*condition1*> is **false**, <*condition2*> is evaluated if it is **true**, <*statement2*> gets executed and exit the case statement. And so on. If none of the conditions are true , <*statement n+1*> is executed. **else** clause is optional.

9. Input and output are done using the instructions **read** and **write**. No format is used to specify the size of input or output quantities.

10. There is only one type of procedure: **Algorithm**. An algorithm consists of a heading and body. The heading takes the form

Algorithm Name (<*parameter list*>)

Where *Name* is the name of the procedure and (<*parameter list*>) is the procedure parameters. The body of the procedure consist of one or more statements enclosed within braces **{** and **}**. Simple variables to the procedure are passed by value. Arrays and records are passed by reference.

Examples:

1. Algorithm for *finding the maximum of n given numbers.* In this algorithm (named Max), A & n are procedure parameters. Result & i are Local variables.

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

2. Algorithm for *Selection Sort* : - we can define this as
From those elements that are currently unsorted, find the smallest and place it next in the sorted list.

```
1    Algorithm SelectionSort( a, n )
2    // sort the array a[ 1 : n ] into nondecreasing 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] ) then j := k;
9                    t := a[i] ; a[i] := a[j]; a[j] := t;
10           }
11   }
```

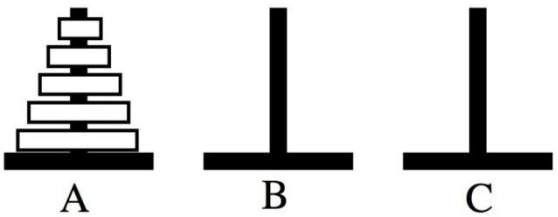
Recursive Algorithms:

- 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 mechanisms are extremely powerful. They can express complex process very clearly.
- The following 2 examples show how to develop a recursive algorithm.

Example 1 : Towers of Hanoi:

It consists of three towers, and a number of disks arranged in ascending order of size. The objective of the problem is to move the entire stack to another tower, obeying the following simple rules:

1. Only one disk can be moved at a time.
2. Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack i.e. a disk can only be moved if it is the uppermost disk on a stack.
3. No disk may be placed on top of a smaller disk.



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 scan be placed on top of it.

```
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  }
```

This algorithm is invoked by ***TowersOfHanoi(n, A, B, C)*.**

Example 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 elements *(a,b,c,d).* The Answer can be constructed by writing
  - *a* followed by all the permutations of *(b,c,d)*
  - *b* followed by all the permutations of *(a,c,d)*
  - *c* followed by all the permutations of *(a,b,d)*
  - *d* followed by all the permutations of *(a,b,c)*

```
1   Algorithm perm(a,k,n)
2   {
3       if (k = n) then write (a[1:n]);        // output permutation
4       else                                   //a[k:n] has more than one permutation
5           // Generate this recursively.
6           for i := k to n do
7           {
8               t:=a[k];  a[k]:=a[i]; a[i]:=t;
9               perm(a, k+1, n);               //all permutation of a[k+1:n]
10              t:=a[k]; a[k]:=a[i]; a[i]:=t;
11          }
12  }
```

This algorithm is invoked by ***perm(a, 1, n)***

EXERCISES

1. Define algorithm and List out the criteria's of an algorithm. (features / properties)
2. What are the four distinct areas of study of algorithm?
3. Define debugging and profiling.
4. Describe pseudo code conventions for specifying algorithms. **
5. The factorial function n! has value 1 when n <=1 and value n * (n – 1)! when n>1. Write both recursive and iterative algorithm to computer n!.
6. What is pseudo-code? Explain with an example.
7. Distinguish between algorithm and pseudo code.
8. Devise an algorithm that inputs three integers and outputs them in nondecreasing order.
9. Present an algorithm that searches an unsorted array a[1:n] for the element x. If x occurs, then return a position in the array; else return zero.
10. The Fibonacci numbers are defined as $f_0 = 0$, $f_1 = 1$, and $f_i = f_{i-1} + f_{i-2}$ for $i > 1$. Write both recursive and iterative algorithms to compute f_i.
11. Devise an algorithm that sorts a collection of n>=1 elements of arbitrary type.
12. Write a recursive algorithm to solve Towers of Hanoi problem with an example.

PERFORMANCE ANALYSIS

Computing time and storage requirement are the criteria for judging algorithms that have direct relationship to performance.

Performance evaluation can be divided into two major phases

1. a priori estimates (performance analysis) and
2. a posteriori testing (performance measurement).

Space Complexity:

Space Complexity of an algorithm is the amount of memory it needs to run to completion.

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

1. A **fixed part**, that is independent of the characteristics of inputs and outputs. This part includes space for instructions(code), space for simple variables, & fixed size component variables, space for constants etc.
2. A **variable part**, which consists of space needed by component variables whose size, is dependent on the particular problem instance being solved, space for reference variables and recursion stack space etc.

The Space requirement *S(P)* of an algorithm *P* may be written as

$$S(P) = c + S_P(\text{instance characteristics})$$ where *'c'* is a constant.

When analyzing the space complexity of an algorithm first we estimate $S_P(\text{instance characteristics})$. For any given problem, we need to determine which instance characteristics to use to measure the space requirements.

Example 1:

```
1  Algorithm abc(a, b, c)
2  {
3     return a+b+b*c + (a+b –c) / ( a+b) + 4.0;
4  }
```

It is characterized by values of *a, b, c*. If we assure that one word is needed to store the values of each *a, b, c* , *result* and also we see S_P*(instance characteristics)=0* as space needed by *abc* is independent of instance characteristics; So 4 words of space is needed by this algorithm.

Example 2:

```
1  Algorithm sum(a, n)
2  {
3     s := 0.0;
4     for i := 1 to n do
5           s := s+a[i];
6     return s;
7  }
```

This algorithm is characterized by *'n'* (number of elements to be summed). The space required for *'n'* is 1 word. The array *a[]* of float values require atleast *'n'* words.
So, we obtain

$$S_{sum}(n) \geq n+3$$

(*n* words for *a*, 1 word for each of *n, i, s*)

Example 3:

```
1  Algorithm Rsum(a, n)
2  {
3     if ( n ≤ 0 ) then  return 0.0;
4     else  return Rsum(a, n-1) + a[n];
5  }
```

Instances of this algorithm are characterized by *n*. The recursion stack space includes space for formal variables, local variables, return address (1-word). In the above algorithm each call to *Rsum* requires 3 words (for *n*, return address, pointer to *a[]*). since the depth of the recursion is *n+1*, the recursion stack space needed is $\geq 3(n+1)$

Time Complexity :

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

- The time *T(P)* taken by a program *P* is the sum of the *compile time* and *Run time*. Compile time does not depend on instance characteristics and a compiled program will be run several times without recompilation, so we concern with just run time of the program. Run time is denoted by T_P***(instance characteristics)***.
- The time complexity of an algorithm is given by the number of steps taken by the algorithm to compute the function. The number of steps is computed as a function of some subset of number of inputs and outputs and magnitudes of inputs and outputs.

- A program step is loosely defined as a syntactically or semantically meaningful segment of a program that has an execution time that is independent of the instance characteristics.
- The number of steps any problem statement is assigned depends on the kind of statement.
  For example, comments → 0 steps.
  Assignment statements → 1 step
  Iterative statements such as for, while & repeat-until→ 1 step for control part of the statement.

- Number of steps needed by a program to solve a particular problem is determined in two ways.

Method 1 : (Step count calculation using a variable)

- In this method, a new global variable count with initial value '0' is introduced into the program.
- Statements to increment count are also added into the program.
- Each time a statement in the original program is executed, count is incremented by the step count of the statement.

Example:

```
1   Algorithm Sum (a, n)
2   {
3      s:=0.0;
4      count :=count + 1;  //count is global; initially 0
5      for i := 1 to n do
6      {
7         count:=count+1;  //for for
8         s:=s+a[i]; count:=count+1; //for assignment
9      }
10     count :=count + 1;  //for last time of for
11    count :=count + 1;  //for the return
12    return s;
13  }
```

The change in the value of *count* by the time this program terminates is the number of steps executed by the algorithm.

The value of count is increment by ***2n*** in the for loop.

At the time of termination the value of count is ***2n+3***.

So invocation of Sum executes a total of **2n+3** steps.

Example 2:

```
1   Algorithm RSum(a, n)
2   {
3      count:=count+1; //for the if conditional
4      if ( n ≤ 0) then
5      {
6            count:=count+1;  //for the return
7            return 0.0;
8      }
9      else
10     {
11           count:=count+1; //for addition, function call, return
12           return RSum(a,n-1)+a[n];
13     }
14  }
```

Let $t_{RSum}(n)$ be the increase in the value of *count* when the algorithm terminates.

When n=0, $t_{RSum}(0) = 2$
When n>0, *count* increases by 2 plus $t_{RSum}(n-1)$

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

For example,

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

- These Recursive formulas are referred to as *recurrence relations*

To solve it, use repeated substitutions

$$\begin{aligned} t_{RSum}(n) &= 2 + t_{RSum}(n-1) \\ &= 2 + 2 + t_{RSum}(n-2) \\ &= 2(2) + t_{RSum}(n-2) \\ &: \\ &= n(2) + t_{RSum}(0) \\ &= 2n + 2, \qquad n \geq 0 \end{aligned}$$

So the step count of *RSum* is *2n+2*.
This step count is telling *the run time for a program with the change in instance characteristics*.

Method 2 : (Step count calculation by building a table)

In this method, the step count is determined by building a table. We list total number of steps contributed by each statement in the table. The table is built in this order.

1. Determine the number of steps per execution (s/e) of the statement and the total number of times (frequency) each statement is executed.
   (*The s/e of a statement is the amount by which the count changes as a result of the execution of that statement.)*
2. The total contribution of the statement is obtained by combining these two quantities.
3. Step count of the algorithm is obtained by adding the contribution of all statements.

Example 1:

	Statement	s/e	Frequency	Total Steps
1	**Algorithm** sum(*a, n*)	*0*	_	*0*
2	{	*0*	_	*0*
3	*s* := 0.0;	*1*	*1*	*1*
4	**for** *i* := 1 **to** *n* **do**	*1*	*n + 1*	*n + 1*
5	*s := s+a[i];*	*1*	*n*	*n*
6	**return** *s;*	*1*	*1*	*1*
7	}	*0*	-	*0*
	Total:			*2n + 3*

Example 2:

	Statement	s/e	Frequency		Total Steps	
			n=0	n>0	n=0	n>0
1	**Algorithm** Rsum(*a, n*)	*0*				
2	{	*0*				
3	**if** ($n \leq 0$) **then**	*1*	*1*	*1*	*1*	*1*
4	**return** 0.0;	*1*	*1*	*0*	*1*	*0*
5	**else**	*0*				
6	**return** Rsum(*a, n-1*) + *a[n];*	*1+x*	*0*	*1*	*0*	*1+x*
7	}	*0*				
	Total:				*2*	*2 + x*

$x = t_{RSum}(n-1)$

Example 3:

	Statement	s/e	Frequency	Total Steps
1	**Algorithm** add(*a, b, c, m, n*)	*0*	_	*0*
2	{	*0*	_	*0*
3	**for** *i* := 1 **to** *m* **do**	*1*	*m+1*	*m+1*
4	**for** *j* := 1 **to** *n* **do**	*1*	*m(n + 1)*	*m(n + 1)*
5	*c[i,j] := a[i,j]+b[i,j];*	*1*	*mn*	*mn*
6	}	*0*	-	*0*
	Total:			*2mn+2m+1*

Exercise :

1. Define time complexity and space complexity.
2. What is space complexity? Illustrate with an example for fixed and variable part in space complexity.
3. Explain the method of determining the complexity of procedure by the step count approach. Illustrate with an example.

4. Implement iterative function for sum of array elements and find the time complexity use the increment count method.
5. Using step count method, analyze the time complexity when 2 m X n matrix added.
6. Write an algorithm for linear search and analyze the algorithm for its time complexity.
7. Give the algorithm for matrix multiplication and find the time complexity of the algorithm using step-count method.
8. Determine the frequency counts for all statements in the following algorithm segment.

```
i : = 1
while ( i <= n) do
{
      x := x + 1
      i := i + 1
}
```

9. Write a recursive algorithm to find the sum of first n integers and Derive its time complexity.
10. Implement an algorithm to generate Fibonacci number sequence and determine the time complexity of the algorithm using the frequency method.
11. What is the time complexity of the following function.

```
int fun() {
      for ( int i = 1; i<=n; i++ )
      {
            for ( int j = 1; j < n; j += i)
            {
                  sum = sum + i * j;
            }
      }
      return(sum);
}
```

12. Give the algorithm for transpose of a matrix m X n and determine the time complexity of the algorithm by frequency-count method.
13. Give the algorithm for matrix addition and find the time complexity of the algorithm using frequency-count method
14. Explain recursive function analysis with an example

Amortized Analysis

- Amortized Analysis not just considers one operation, but a sequence of operations on a given data structure. It averages cost over a sequence of operations.
- In an amortization scheme we charge some of the actual cost of an operation to other operations. This reduces the charged cost of some operations and increases that of others. The amortized cost of an operation is the total cost charged to it.
- The cost transferring (amortization) scheme is required to be such that the sum of the amortized costs of the operations is greater than or equal to the sum of their actual costs.
- The only requirement of amortized complexity is that sum of the amortized complexities of all operations in any sequence of operations be greater than or equal to their sum of the actual complexities.

That is $$\sum_{1 \le i \le n} amortized(i) \ge \sum_{1 \le i \le n} actual(i) \quad \text{--(1)}$$

- Where *amortized(i)* and *actual(i)*, respectively denote the amortized and actual complexities of the i^{th} operation in a sequence of *n* operations.
- For this reason, we may use the sum of the amortized complexities as an upper bound on the complexity of any sequence of operations.

- Amortized cost of an operation is viewed as the amount you charge the operation rather than the amount the operation costs. You can charge an operation any amount you wish so long as the amount charged to all operations in the sequence is at least equal to the actual cost of the operation sequence.
- Relative to the actual and amortized costs of each operation in a sequence of n operations, we define a potential function P(i) as below

$$P(i) = amortized(i) - actual(i) + P(i-1) \quad \text{-- (2)}$$

- That is, the i^{th} operation causes the potential function to change by the difference between the amortized and actual costs of that operation.

- If we sum the equation (2) for $1 \le i \le n$, we get

$$\sum_{1 \le i \le n} P(i) = \sum_{1 \le i \le n} (amortized(i) - actual(i) + P(i-1))$$

or

$$\sum_{1 \le i \le n} (P(i) - P(i-1)) = \sum_{1 \le i \le n} (amortized(i) - actual(i))$$

or

$$P(n) - P(0) = \sum_{1 \le i \le n} (amortized(i) - actual(i))$$

From equation (1), if follows that $P(n) - P(0) \ge 0$ -- (3)

Under the assumption P(0)=0, the potential P(i) is the amount by which the first *i* operations have been overcharged.
Generally, when we analyze the complexity of a sequence of *n* operations, *n* can be any nonnegative integer. Therefore equation (3) can hold for all nonnegative integers.

There are **three** popular methods to arrive at amortized costs for operations.
1. **Aggregate method**: In this we determine the upper bound [*UpperBoundOnSumOfActualCosts(n)*]for sum of the actual costs of the n operations.
The amortized cost of each operation is set equal to *UpperBoundOnSumOfActualCosts(n) / n*

2. **Accounting Method**: In this we assign amortized costs to the operations (by guessing), compute the *p(i)*s and show that $p(n) - p(0) \ge 0$.

3. **Potential method:** we start with a potential function that satisfies $p(n) - p(0) \ge 0$. And compute the amortized complexities using

$$P(i) = amortized(i) - actual(i) + p(i-1)$$

Asymptotic notations:

- The main idea of asymptotic analysis is to have a measure of efficiency of algorithms that doesn't depend on machine specific constants, and doesn't require algorithms to be implemented and time taken by programs to be compared.
- Asymptotic notations are mathematical tools to represent time complexity of algorithms for asymptotic analysis.

The following asymptotic notations are mostly used to represent time complexity of algorithms.

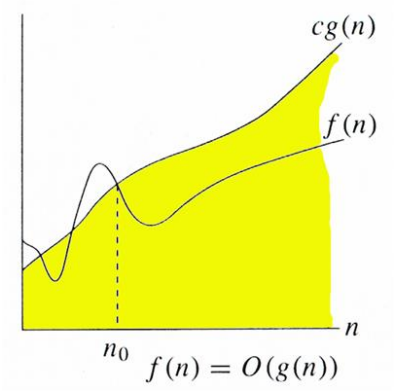
[Big-Oh] O-notation:

- The Big O notation defines an upper bound of an algorithm; it bounds a function only from above.
- Big-Oh notation is used widely to characterize running time and space bounds in terms of some parameter *n*, which varies from problem to problem.
- Constant factors and lower order terms are not included in the big-Oh notation.
- For example, consider the case of Insertion Sort. It takes linear time in best case and quadratic time in worst case. We can safely say that the time complexity of Insertion sort is $O(n^2)$. Note that $O(n^2)$ also covers linear time.

Def: The function $f(n)=O(g(n))$ iff there exists positive constants c and n_0 such that

$f(n) \leq c*g(n)$ for all n, $n \geq n_0$.

Figure shows that, for all values n at and to the right of n_0, the value of function $f(n)$ is on or below $c.g(n)$.



Example : $7n-2$ is $O(n)$
Proof: By the big-Oh definition, we need to find a real constant $c>0$ and an integer constant $n_0 \geq 1$ such that $7n-2 \leq cn$ for every integer $n \geq n_0$.
Possible choice is $c=7$ and $n_0=1$.

Example: $20n^3+10n \log n+5$ is $O(n^3)$
Proof: $20n^3+10n \log n+5 \leq 35n^3$, for $n \geq 1$
In fact, any polynomial $a_k n^k + a_{k-1}n^{k-1} + \ldots + a_0$ will always be $O(n^k)$.

Here is a list of functions that are commonly encountered when analyzing algorithms. The slower growing functions are listed first. k is some arbitrary constant.

Notation	Name
O(1)	Constant
O(log n)	Logarithmic
O(n)	Linear
O(n log n)	Polylogarithmic
O(n^2)	Quadratic
O(n^3)	Cubic
O(n^k) $(k \geq 1)$	Polynomial
O(k^n) $(k>1)$	exponential

Instead of always applying the big-Oh definition directly to obtain a big-Oh characterization, we can use the following rules to simplify notation.

Theorem: Let $d(n)$, $e(n)$, $f(n)$ and $g(n)$ be functions mapping nonnegative integers to nonnegative reals. Then

1. If $d(n)$ is $O(f(n))$, then $kd(n)$ is $O(f(n))$, for any constant $k > 0$.
2. If $d(n)$ is $O(f(n))$ and $e(n)$ is $O(g(n))$, then $d(n)+e(n)$ is $O(f(n)+g(n))$.

3. If $d(n)$ is $O(f(n))$ and $e(n)$ is $O(g(n))$, then $d(n).e(n)$ is $O(f(n).g(n))$.
4. If $d(n)$ is $O(f(n))$ and $f(n)$ is $O(g(n))$, then $d(n)$ is $O(g(n))$.
5. If $f(n)$ is a polynomial of degree d (that is, $f(n)=a_0 + a_1n + ...+ a_dn^d$), then $f(n)$ is $O(n^d)$.
6. n^x is $O(a^n)$ for any fixed $x>0$ and $a>1$.
7. $\log n^x$ is $O(\log n)$ for any fixed $x>0$.
8. $\log^x n$ is $O(n^y)$ for any fixed constants $x>0$ and $y>0$.

Example : $2n^3 + 4n^2 \log n$ is $O(n^3)$
Proof:

- $\log n$ is $O(n)$
- $4n^2 \log n$ is $O(4n^3)$
- $2n^3 + 4n^2 \log n$ is $O(2n^3 + 4n^3)$
- $2n^3 + 4n^3$ is $O(n^3)$
- $2n^3 + 4n^2 \log n$ is $O(n^3)$

[Omega] Ω-notation:

- Ω-notation provides an asymptotic lower bound.

Def: The function $f(n)=\Omega(g(n))$ iff there exists positive constants c and n_0 such that

$f(n) \geq c*g(n)$ for all n, $n \geq n_0$

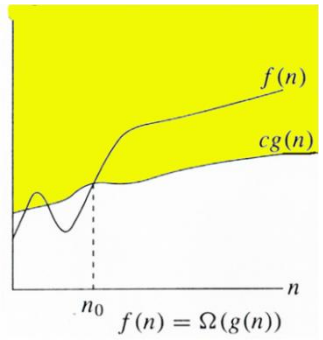


Figure shows the intuition behind Ω-notation for all values n at or to the right of n_0, the value of $f(n)$ is on or above $c.g(n)$.

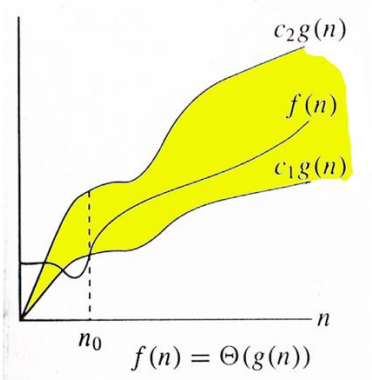
- If the running time of an algorithm is $\Omega(g(n))$, then the meaning is, "the running time on that input is atleast a constant times $g(n)$, for sufficiently large n".
- It says that, $\Omega(n)$ gives a lower-bound on the best-case running time of an algorithm.
- Eg: Best case running time of insertion sort is $\Omega(n)$.

Example: 3n+2 is $\Omega(n)$.
Proof: $3n+2 \geq 3n$ for $n \geq 1$ ($c=3$, $n0=1$)

[Theta] Θ - notation :

Def: The function $f(n)=\Theta(g(n))$ iff there exists positive constants c_1, c_2 and n_0 such that
$c1*g(n) \leq f(n) \leq c2*g(n)$ for all n, $n \geq n_0$

Following figure gives an intuitive picture of functions $f(n)$ and $g(n)$, where $f(n)=\Theta(g(n))$.



- For all values of n at and to the right of n_0, the value of $f(n)$ lies at or above $c1.g(n)$ and at or below $c2.g(n)$.

- For all $n \geq n_0$, the function $f(n)$ is equal to $g(n)$ to within constant factors.
- We can say that $g(n)$ is an asymptotically tight bound for $f(n)$.

Example :

$$3n + 2 = \Theta(n)$$

find c_1, c_2, n_0 such that

$$\left.\begin{aligned} 3n + 2 &\geq c_1 . g(n) \\ 3n + 2 &\leq c_2 . g(n) \end{aligned}\right\} \quad \forall n \geq n_0$$

$c_1 = 3, c_2 = 4, n_0 = 2$ satisfies the above.

$\therefore 3n + 2 = \Theta(n)$

[Little-oh] o-notation :

- O-notation provides asymptotic upper bound. It may or may not be asymptotically tight.
- The bound $2n^2 = O(n^2)$ is asymptotically tight, but bound $2n = O(n^2)$ is not.

- o-notation is used to denote an upper bound that is not asymptotically tight.

Def: The function $f(n)=o(g(n))$ iff there exists a constant $n_0>0$ such that $0 \leq f(n) \leq c.g(n)$ for all $n>n_0$ and for any positive constant $c>0$.

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

Example: The function $3n+2 = o(n^2)$

Since $\lim_{n\to\infty} \frac{3n+2}{n^2} = 0$

[little-omega] ω - notation :

- ω - notation is used to denote a lower bound that is not asymptotically tight.
- By analogy, ω - notation to Ω-*notation* as *o-notation* to *O-notation*.

Def: The function $f(n) = \omega(g(n))$ iff there exists a constant $n_0>0$ such that $0 \leq c.g(n) \leq f(n)$ for all $n \geq n_0$ and for any positive constant $c>0$.

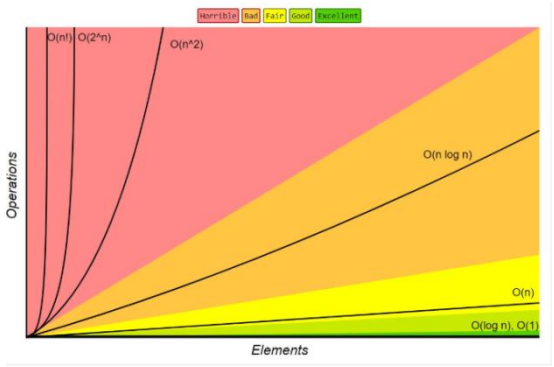
The function $f(n) = \omega(g(n))$ iff $\lim_{n\to\infty} \frac{g(n)}{f(n)} = 0$

Practical complexities

- Time complexity of an algorithm is generally some function of the instance characteristics.
- This function is useful
  - in determining how the time requirements vary as the instance characteristics change.
  - in comparing two algorithm P and Q which perform the same task,
- Assume that algorithm P has complexity $\Theta(n)$ and algorithm Q has complexity $\Theta(n^2)$. we can assert that algorithm P is faster than algorithm Q for sufficiently large n.

Following table shows how various functions grow with n.

n	log n	n	n log n	n2	n3	2n
1	0	1	0	1	1	2
2	1	2	2	4	8	4
4	2	4	8	16	64	16
8	3	8	24	64	512	256
16	4	16	64	256	4096	65536
32	5	32	160	1024	32768	4294967296
64	6	64	384	4096	262144	1.84467E+19
128	7	128	896	16384	2097152	3.40282E+38
256	8	256	2048	65536	16777216	1.15792E+77
512	9	512	4608	262144	134217728	1.3408E+154



It is very clear from the table that the function 2^n grows very rapidly with n.

eg:
Let a computer can execute 1 billion steps per second
And n = 40
If algorithm needs 2^n steps for execution, then the number of steps needed = $1.1 * 10^{12}$ which takes 18.3 minutes.

Important Questions :

1. Write about three popular methods to arrive at amortized costs for operations with example.
2. What is amortized analysis and Explain with and example. ****
3. What is amortized analysis of algorithms and how is it different from asymptotic analysis.
4. What are asymptotic notations? And give its properties.
5. Give the definition and graphical representation of asymptotic notations.
6. Describe and define any three asymptotic notations.
7. Give the Big – O notation definition and briefly discuss with suitable example.
8. Define Little Oh notation with example.
9. Write about big oh notation and also discuss its properties.
10. Define Omega notation
11. Explain Omega and Theta notations.
12. Compare Big-oh notation and Little-oh notation. Illustrate with an example.
13. Differentiate between Bigoh and omega notation with example.
14. Show that the following equalities are incorrect with suitable notations.

$$10\,n^2 + 9 = O(n)$$
$$n^2 \log n = \theta(n^2)$$
$$n^2 / \log n = \theta(n^2)$$
$$n^3 2^n + 5\,n^2 3^n = O(n^3 2^n)$$

15. Describe best case, average case and worst case efficiency of an algorithm.
16. Find big-oh and little-oh notation for $f(n) = 7\,n^3 + 50\,n^2 + 200$
17. What are different mathematical notations used for algorithm analysis
18. Prove the theorem if $f(n) = a_m n^m + \ldots + a_1 n + a_0$, where $f(n) = O(n^m)$.
19. Prove the theorem if $f(n) = a_m n^m + \ldots + a_1 n + a_0$, where $f(n) = \theta(n^m)$.

Divide and conquer: General method, Defective chess board, Binary search, Finding the maximum and minimum, Merge Sort, Quick sort, performance measurement, Randomized Sorting algorithms.

DIVIDE AND CONQUER

General Method:

- Given a function to compute on *'n'* inputs, the divide and conquer strategy
  - Splits the input into *k* subsets, $1<k\leq n$, it yields *k* subproblems.
  - These subproblems must be solved.
  - A method must be found to combine subsolutions into a solution of the whole.
- The Divide and Conquer strategy is reapplied, if the subproblems are large.
- Often these subproblems are of the same type of the original problem. For this reason the divide-and-conquer principle is expressed by a *recursive algorithm*.
- Splitting the problem into subproblems is continued until the subproblems become small enough to be solved without splitting.

Control Abstraction:

- Control abstraction is a procedure whose flow of control is clear but whose primary operations are specified by other procedures whose precise meanings are left undefined.
- Control abstraction that mirrors the way an algorithm is based on divide-and-conquer is shown below.

```
1   Algorithm DAndC(P)
2   {
3       if Small(P) then return S(P);
4       else
5       {
6           divide P into smaller instances P1, P2, . . Pk    k≥1
7           Apply DAndC to each of these problems;
8           return combine(DAndC(P1), DAndC(P2) . . . DAndC(Pk));
9       }
10  }
```

- Initially algorithm is invoked as *DAndC(P)*, where *P* is the problem to be solved.
- *Small(P)* is a boolean-valued function, it determines whether the input size is small enough that the answer can be computed without splitting.
- *S(P)* is invoked if *Small(P)* returns *true*.
- If *Small(P)* returns false, then the problem *P* is divided into subproblems $P_1, P_2, \ldots P_k$. These subproblems are solved by recursive application of *DAndC*.
- Combine function combines the solutions of the *k* subproblems to determine the solution to problem *P*.
- If the size of *P* is *n*, and the sizes of *k* subproblmes are $n_1, n_2, \ldots n_k$, then computing time of the *DAndC* is described by the recurrence relation.

$$T(n)=\begin{cases} g(n) & n \text{ small} \\ T(n_1)+T(n_2)+.....+T(n_k)+f(n) & otherwise \end{cases} \qquad ----(1)$$

Where

T(n) – time for DAndC on any input of size n.
g(n) – time to compute answer directly for small inputs.
f(n) – time for dividing P into subproblems and combing solutions of subproblems.

DAndC strategy produces subproblems of type original problem, Therefore it is convenient to write these algorithms using recursion.

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

$$T(n)=\begin{cases} T(1) & n=1 \\ a.T\left(\frac{n}{b}\right)+f(n) & n>1 \end{cases} \qquad \text{----(2)}$$

Where *a* and *b* are known as constants.

To solve this recurrence relation, we assume that *T(1)* is known, *n* is a power of *b* (i.e., $n=b^k$, $\log_b n=k$)
Substitution method of solving Recurrence Relation repeatedly makes substitution for each occurrence of the function in right hand side until all such occurrences disappear.

Example: consider the case in which *a=2* and *b=2*, *T(1)=2* and *f(n)=n*.

$$T(n)=\begin{cases} T(1) & n=1 \\ a.T\left(\frac{n}{b}\right)+f(n) & n>1 \end{cases}$$

$$T(n)=2.T\left(\frac{n}{2}\right)+n$$

$$=2\left[2.T\left(\frac{n}{4}\right)+\frac{n}{2}\right]+n$$

$$=4.T\left(\frac{n}{4}\right)+2.n$$

$$=4\left[2.T\left(\frac{n}{8}\right)+\frac{n}{4}\right]+2n$$

$$=8.T\left(\frac{n}{8}\right)+3.n$$

:

In general $=2^i.T\left(\frac{n}{2^i}\right)+i.n$ for any $\log_2 n \geq i \geq 1$

In particular $T(n)=2^{\log_2 n}.T\left(\frac{n}{2^{\log_2 n}}\right)+n.\log_2 n$

$$T(n)=n.T(1)+n.\log_2 n$$

$$T(n)=n.\log_2 n+2n$$

Solving recurrence Relation (2) using substitution method, we get

$$T(n)=a^{\log_b n}.T(1)+\log_b n.f(n)$$

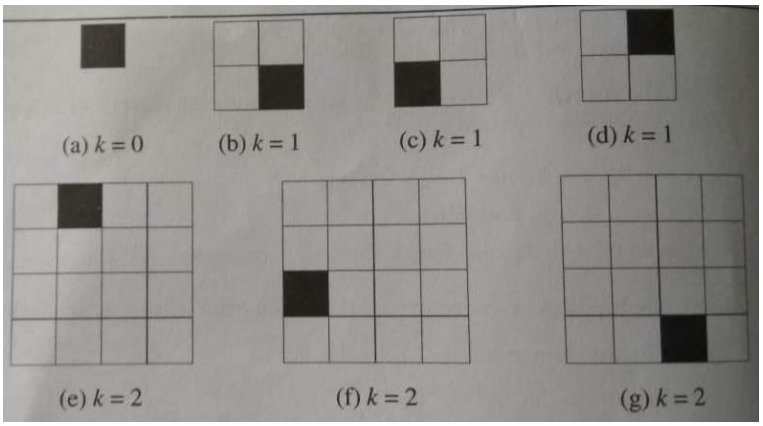
$$=n^{\log_b a}.T(1)+\log_b n.f(n)$$

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

where $u(n)=\sum_{j=1}^{k} h(b^j)$ and $h(n)=\frac{f(n)}{n^{\log_b a}}$

Defective Chessboard Problem :

- A defective chessboard is a $2^k \; X \; 2^k$ board of squares with exactly one defective square.
- Possible defective chessboards for $k \leq 2$ are shown below.



- Shaded square is defective.

- When k = 0, the size of the chess board is 1 X 1 and there is only one possible defective chessboard.
- When k = 1, the size of the chess board is 2 X 2, and there can be 4 possible defective chessboards.
- Therefore, for any k, there are exactly 2^{2k} defective chessboards.

Triomino :

- A triomino is an L shaped object that can cover three squares of a chessboard.
- A triomino has four orientations. Following figure shows triominoes with different orientations.

Defective chessboard problem:

- In this problem, we are required to tile a defective chessboard using triominoes.

Constraints:

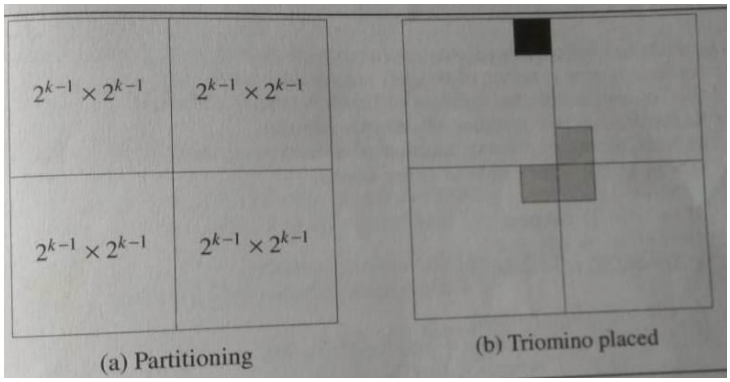
- Two triominoes may not overlap in this tiling.
- Triominoes should not cover defective square.
- Triominoes must cover all other squares.

With the above constaints, number of triominoes required to tile = $\frac{2^{2k}-1}{3}$

If k=0, number of triominoes = 0
If k=1, number of triominoes = 1
If k=2, number of triominoes = 5

Solution :

- Divide and Conquer leads to an elegant solution to this problem.
- The method suggests reducing the problem of tiling a $2^k \; X \; 2^k$ defective chessboard to tiling a smaller defective chessboard.
- A $2^k \; X \; 2^k$ chessboard can be partitioned into four $2^{k-1} \; X \; 2^{k-1}$ chessboards.



- Only one board has a defective square. To convert the remaining three boards into defective chessboards, we place triomino at the corner formed by these three.
- This partitioning technique is recursively used to tile the entire 2^k *X* 2^k chessboard.
- This recursion terminates when the chessboard size has been reduced to 1 X 1

Pseudocode for this strategy to solve Defective chessboard problem :

```
Algorithm TileBoard(topRow, topCol, dRow, dCol, size)
//topRow, topCol are top-left corner of the board
//dRow, dCol are row and column numbers of defective square.
//size is length of one side of the chessboard.
{
   if ( size = 1) return;
   tileToUse := tile++;
   quadSize = size / 2;

   //tile top-left quadrant
   if (dRow < topRow + quadSize && dCol < topCol + quadSize) then
        //defect is in this quadrant
        TileBoard(topRow, topCol, dRow, dCol, quadSize);
   else
   {
        //no defect, place a tile in bottom-right corner
        board[topRow + quadSize -1][topCol + quadSize – 1] := tileToUse;
        TileBoard(topRow,topCol, topRow+quadSize – 1, topCol+quadSize-1, quadSize);
   }

   //code for remaining three quadrands is similar
}
```

- Above pseudocode uses two global variables
  - board is a two-dimensional array that represents the chessboard. board[0][0] represents the top-left corner.
  - tile, with initial value 1, gives the index of the next tile to use.
- This algorithm is invoked with the call ***TileBoard(0,0,dRow,dCol,size)***
  Where
  - Size = 2^k
  - dRow and dCol are row and column index of the defective square.

Time Complexity:

Let ***t(k)*** denote the time taken by TileBoard to tile a defective chessboard.
When k=0, a constant amount of time is spent. Let the constant be ***d***.
When k>0, recursive calls are made. These calls take ***4.t(k – 1)*** time.
This can be represented by the following recurrence equation.

$$t(k) = \begin{cases} d & K = 0 \\ 4t(k-1) + c & K > 0 \end{cases}$$

By solving this using the substitution method, we obtain

$$t(k) = \theta(4^k) = \theta(number\ of\ tiles\ needed)$$

Binary Search:

- Let a_i be a list of elements that are sorted in nondecreasing order.
- Here i is in the range of 1 to n, $1 \le i \le n$.
- Binary search is a problem of determining whether an element x is present in the list or not.
  - If x is present in the list, then we need to determine value j such that $a_j = x$.
  - If x is not in the list, then j is to be set to zero.

- Let $P = (n, a_{low}, \ldots, a_{high}, x)$ denote an instance of search problem. Here
  - n is number of elements in the list.
  - $a_{low}, \ldots, a_{high}$ list of elements
  - x is the element searched for.

- Divide-and-conquer can be used to solve this problem.
  - Let *Small(P)* be true if $n=1$.
  - If $x=a_i$ then *Small(P)* will take the value of i otherwise *Small(P)* will take the value 0
    ⇨ $g(1) = \Theta(1)$.

- If P has more than one element, it can be divided into subproblems, as follows.
  Pick and index q in the range *low* to *high* and compare x with a_q. There are three possibilities.

  1. $\boldsymbol{x = a_q}$: The problem P is immediately solved in this case.
  2. $\boldsymbol{x < a_q}$: x has to be searched in the sublist in this case.
     The sublist is $a_{low}, \ldots, a_{q-1}$.
     Therefore, P reduces to $P = (n, a_{low}, \ldots, a_{q-1}, x)$
  3. $\boldsymbol{x > a_q}$: In this case also x has to be searched in the sublist
     Therefore P reduces to $P = (n, a_{q+1}, \ldots, a_{high}, x)$.

- Dividing the problem P into subproblem takes $\Theta(1)$ time.
- After a comparison with $\boldsymbol{a_q}$, the remaining problem instance can be solved by using divide-and-conquer scheme again.
- If q is choosen that $\boldsymbol{a_q}$ is the middle element, then that search algorithm is called **Binary Serach**.

  i.e., $q = \left\lfloor \frac{(low + high)}{2} \right\rfloor$ 8074816703

- There is no need of combining answers in binary search, because answer of the subproblem is also the answer of the original problem *P*.

Algorithm for recursive binary Search:

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

- It is initially invoked as ***BinSearch(a,1,n,x)***

Algorithm for non-recursive binary search

```
1   Algorithm BinSearch(a,n,x)
2   {
3       low=1, high=n;
4       while (low ≤ high)
5       {
6           mid:=(low+high)/2;
7           if (x=a[mid]) then return mid;
8           else if (x<a[mid]) then high:=mid-1;
9           else low:=mid+1;
10      }
11      return 0;
12  }
```

Non-recursive binary search algorithm has three inputs: *a*, *n*, *x*. The while loop continues processing as long as there are elements left to check. A zero is returned if *x* is not present in the list, or *j* is returned if $a_j=x$.

Example: Consider a list with 14 entries. i.e., n=14.
Place them in a[1:14]

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

The variables low, high, mid need to be traced to simulate this algorithm.

x=151	low	high	mid
	1	14	7
	8	14	11
	12	14	13
	14	14	14
			found

x= -14	low	high	mid
	1	14	7
	1	6	3
	1	2	1
	2	2	2
	2	1	Not found

Space Required for Binary Search :
For binary search algorithm, storage is required for – *'n'* elements of the array, variables *low, high, mid, x*. i.e., *n+4* locations.

Time required for Binary Search :

- The three possibilities that are needed to be considered are best, average and worst cases.
- To determine time for this algorithm, concentrate on comparisons between *x* and the elements in *a[]*.
- Comparisons between *x* and the elements in *a[]* are referred to as element comparisons.
- To test all successful searches, *x* must take on *'n'* values in *a[]*.
- To test all unsuccessful searches, *x* need to take *(n+1)* comparisons.

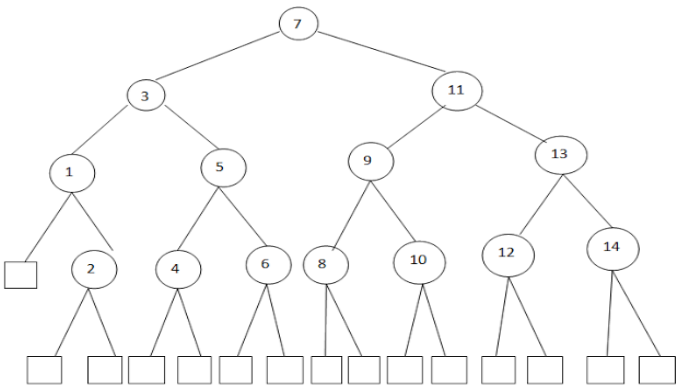
Example: The number of element comparisons needed to find each of the 14 elements is

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

- Average Comparisons for successful search : $\frac{45}{14} \approx 3.21$
- There are (14+1=15) possible ways that an unsuccessful search may terminate.
- If *x<a[1]*, then the algorithm requires 3 element comparisons to determine that x is not present. For remaining cases, the algorithm requires 4 element comparisons.
- Average number of element comparisons for unsuccessful search = $\left(\frac{3+14*4}{15}\right) = \frac{59}{15} \approx 3.93$

But we prefer a formula for *n* elements. A good way to derive a formula is to consider sequence of mid values that are produced by all possible values of *x*. A *Binary Decision Tree* used to describe this. Each tree in this node is the value of mid.

Example : if n=14, a Binary Decision Tree that traces the way in which these values are produced is shown below.



- The first comparison is x with a[7]. If x<a[7], then the next comparison is with a[3]; If x>a[7], then the next comparison is with a[11].
- Each path through the tree represents a sequence of comparisons in the binary search method.
- If x is present then the algorithm will end at one of the circular nodes that lists the index into the array where x was found.

- If x is not present, the algorithm will terminate at one of the square nodes.
- Circular nodes are called *internal nodes*, and square nodes are referred to as *external nodes*.

If n is in the range $[2^{k-1}, 2^k]$ then binary search makes atmost *k* element comparisons for a successful search and either *k-1* or *k* comparisons for an unsuccessful search.
i.e., the time for a successful search is *O(log n)* and for an unsuccessful search is $\Theta(\log n)$

Average, worst case time for binary search:

- From the BDT, it is clear that the distance of a node from the root is one less than its level.
- **Internal Path length (I) :** sum of the distances of all internal nodes from the root.
- **External Path length (E) :** sum of the distances of all external nodes from the root.

- By mathematical induction, we can say that "for any binary tree with n internal nodes, E and I are related by the formula"

$$E=I+2n.$$

- Let $A_s(n)$ be the average number of comparisons in a successful search, and $A_u(n)$ be the average number of comparisons in an unsuccessful search.
- The number of comparisons needed to find an element represented by an internal node is one more than the distance of this node from the root.

$$\text{Hence } A_s(n) = 1 + \frac{I}{n}$$

Since every binary tree with n internal nodes has n+1 external nodes, it follows that

$$A_u(n) = \frac{E}{(n+1)}$$

Using these three formulas,

$$\begin{aligned} A_s(n) &= 1 + \frac{I}{n} \\ &= 1 + \frac{\text{E - 2n}}{\text{n}} \\ &= 1 + \frac{A_u(n).(n+1) - 2n}{n} \\ &= 1 + \frac{A_u(n).n + A_u(n)}{n} - 2 \\ &= \left(1 + \frac{1}{n}\right) A_u(n) - 1 \end{aligned}$$

From this, we see that $A_s(n)$ and $A_u(n)$ are directly related.
From BDT, we conclude that average and worst case comparisons for Binary Search are same within a constant time.

Best-case : For a successful search only one element comparison is needed and for unsuccessful search log n element comparisons are needed in best case.

In conclusion, the computing times of binary search are

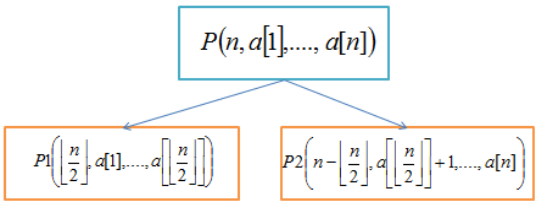
Successful Searches			Unsuccessful Searches
$\Theta(1)$	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$
Best	Average	Worst	Best, average, worst

Finding the Maximum and Minimum:

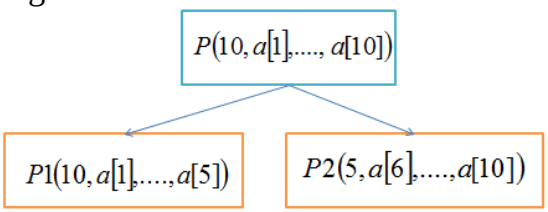
Problem : Find the maximum and minimum items in a set of 'n' elements.

Solution by using Divide and Conquer approach:

- Let P = (n, a[i], …, a[j]) denote an instance of the problem.
- Where n is the number of elements in the list and a[i],..a[j] denote the list in which we want to find minimum and maximum.
- Let **Small(P)** be true when n≤2
- If n = 1, the maximum and minimum are a[i]
- If n = 2, the problem can be solved by doing one element comparison.
- Otherwise, P has to be divided into smaller instances.
- Like,



Eg:



- After dividing P into smaller subproblems, we can solve them by recursively invoking the same divide and conquer algorithm.

Combining the solutions :

- Let P is the problem and P1 and P2 are its subproblems, then
  - MAX(P) is larger of MAX(P1) and MAX(P2) and
  - MIN(P) is smaller of MIN(P1) and MIN(P2)

Algorithm : to find maximum and minimum recursively.

```
Algorithm MaxMin(i, j, max, min)
// a[1 : n] is a global array. Parameters i and j are integers, 1 ≤ i ≤ j ≤ n.
//sets the max and min to the smallest and largest values.
{
        if (i = j) then max := min := a[i];
        else if ( i = j – 1) then
        {
                if (a[i] < a[j]) then
                {
                        max := a[j];  min := a[i];
                }
                else
                {
                        max := a[i]; min := a[j];
                }
        }
        else
        {
        mid := (i + j) / 2;
```

```
        MaxMin(i, mid, max, min);
        MaxMin(mid + 1, j, max1, min1);
        if (max < max1) then max := max1;
        if (min > min1) then min := min1;
        }
}
```

The procedure is initially invoked by the statement

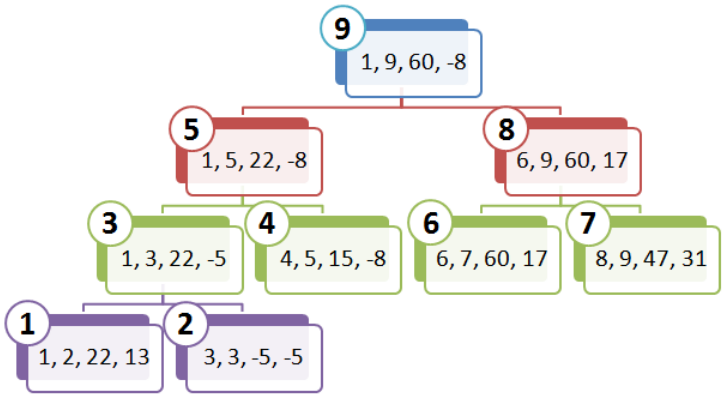
MaxMin(1,n,x,y)

Simulation

n = 9

a	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
	22	13	-5	-8	15	60	17	31	47

Tree of recursive calls



- Root node contains 1 and 9 as values of i, j corresponding to initial call to MaxMin.
- The produced two new calls, where i, j values are 1, 5 and 6, 9 respectively.
- From the tree, maximum depth of recursion is four.
- Circled numbers represent the orders in which Max & Min assigned values.

Analysis:

Computing Time : What is number of element comparsions needed?

$$T(n) = \begin{cases} 0 & n = 1 \\ 1 & n = 2 \\ 2.T\left(\frac{n}{2}\right) + 2 & n > 2 \end{cases}$$

Solve this recurrence equation using substitution method. $T(n) = \frac{3n}{2} - 2$

- It is the best, average, worst case number of comparisons when ‘n’ is power of 2.
- Number of comparsions in a straight method of maximum and minimum is 2n -2 . i.e., this algorithm saves 25% of comparisons.

Storage:

- MaxMin is worse than the straight forward algorithm because it requires stack space for [i,j, max, min, max1, min1]
- For ‘n’ elements, there will be log n + 1 levels of recursion and we need to save seven values for each recursive call.

Merge Sort:

- Merge Sort is a sorting algorithm with the nice property that its worst case complexity is O(n log n).
- Given a sequence of *'n'* elements *a[1],…,a[n]* the general idea is to imagine them split into two sets *a[1],.....,a[* $\left\lfloor \frac{n}{2} \right\rfloor$ *]* and *a[* $\left\lfloor \frac{n}{2} \right\rfloor$ *+1],....a[n]*.
- Each set is individually sorted, and the resulting sorted sequences are merged to produce a single sorted sequence of 'n' elements.

- It is an ideal example of divide-and-conquer strategy, in which
  - Splitting is into two equal sized sets
  - Combining operation is merging of two sorted sets into one.
- *MergeSort* algorithm describes this process using recursion and *Merge* algorithm merges two sorted sets.
- *'n'* elements should be placed in *a[1:n]* before executing *MergeSort*. Then *MergeSort(1,n)* causes the keys to be rearranged into nondecreasing order in *a*.

Algorithm for MergeSort:

```
1   Algorithm MergeSort(low, high)
2   //a[low:high] is a global array to be sorted. Small(P) is true if there is only one element
3   {
4      if (low < high ) then   //if there are more than one element
5      {
6         mid := ⌊(low+high)/2⌋;        //Divide P into sub probelms
7         //Solve sub problems
8         MergeSort(low, mid);
9         MergeSort(mid+1, high);
10        Merge(low, mid, high);
11     }
12  }
```

Algorithm for merging two sorted subarrays.

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

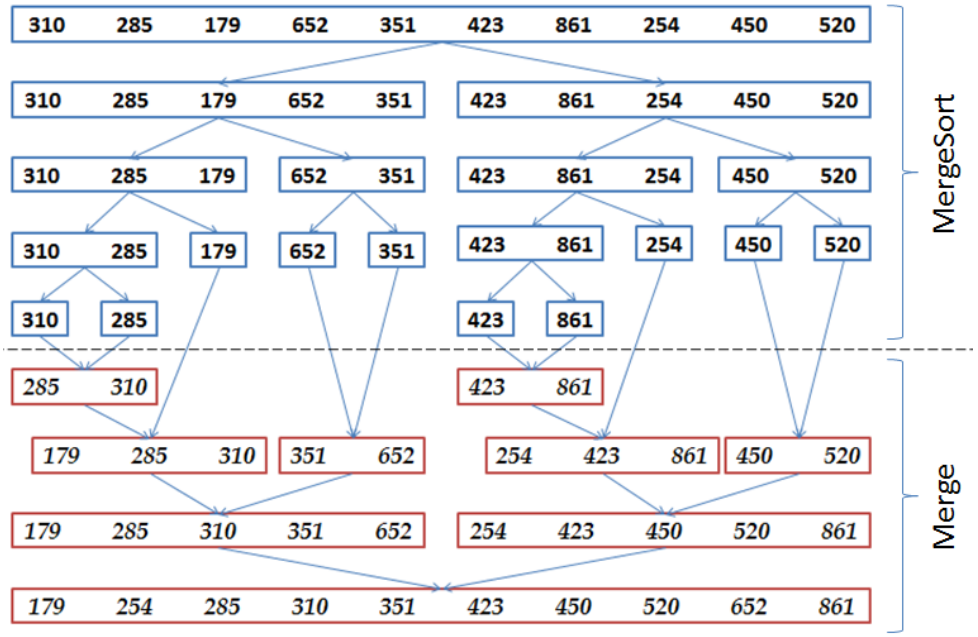
```
17        if (h<mid) then
18           for k := j to high do
19           {
20              b[i]:=a[k]; i:=i+1;
21           }
22        else
23           for k:=h to mid do
24           {
25              b[i]:=a[k]; i:=i+1;
26           }
27        for k:=low to high do
28           a[k]:=b[k];
29     }
```

Example :

Consider an array of 10 elements.

$$a[1:10] = (310, 285, 179, 652, 351, 423, 861, 254, 450, 520)$$

Algorithm MergeSort begins by splitting *a[]* until they become one-element subarrays. Now merging begins. This division and merging is shown in the below figure.



- Following figure is a tree that represents the sequence of recursive calls that are produced by **MergeSort** when it is applied to *10* elements.
- The pair of values in each node is the values of the parameters *low* and *high*.

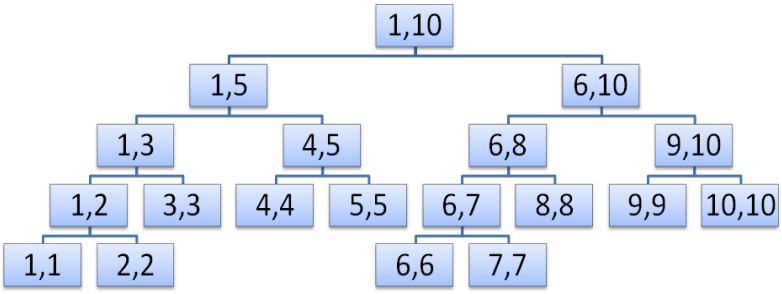


Figure: Tree of calls of MergeSort(1,10)

- Following figure is a tree representing the calls to procedure Merge. For example, the node containing 1, 2, and 3 represents the merging of a[1:2] with a[3].

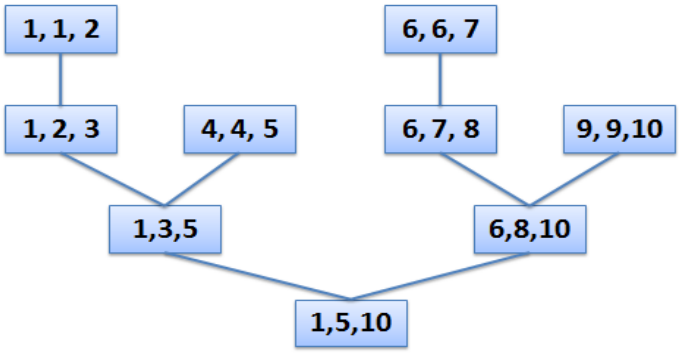


Figure: Tree of calls of Merge

If the time for the merging operation is proportional to *n*,

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

$$T(n) = \begin{cases} a & n = 1 \\ a.T\left(\frac{n}{2}\right) + c.n & n > 1 \end{cases}$$

Where *a* and *c* are constants.

We can solve this equation by successive substitutions.
Assume n is a power of *2*, $n=2^k$, i.e., *log n = k.*

$$T(n) = 2.T\left(\frac{n}{2}\right) + c.n$$

$$= 2\left[2.T\left(\frac{n}{4}\right) + c.\frac{n}{2}\right] + cn.$$

$$= 2^2.T\left(\frac{n}{4}\right) + 2cn$$

$$= 2^2\left[2.T\left(\frac{n}{8}\right) + c.\frac{n}{4}\right] + 2cn$$

$$= 2^3.T\left(\frac{n}{8}\right) + 3cn$$

after k substitutions

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

$$= an + cn.\log n$$

It is easy to see that

$$if\ 2^k < n \le 2^{k+1},\ \text{then}\ T(n) \le T(2^{k+1})$$

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

Quick Sort:

- In Quick sort, the division into two subarrays is made in such a way that, the sorted subarrays do not need to be merged later.
- This is achieved by rearranging the elements in *a[1:n]* such that *a[i]≤a[j]* for all *i* between *1* and *m* and all *j* between *m+1* and *n* for some *m, 1≤m≤n.*
- Thus, the elements in *a[1:m]* and *a[m+1:n]* can be independently sorted. No merge is needed.
- The rearrangement of the elements is accomplished
  - By picking some element of *a[]*, say *t=a[s]*.
  - And then reordering the elements so that all elements appearing before *t* in *a[1:n]* are less than or equal to *t* and all elements appearing after *t* are greater than or equal to *t.*
  - The rearranging is referred to *partitioning*.

Following algorithm accomplishes partitioning of elements of a[m:p]. It is assumed that a[m] is the partitioning element.

```
1   Algorithm Partition(a, m, p)
2   {
3       pivot:=a[m], i:=m+1, j:=p;
4       while (i<j) do
5       {
6           while (a[i]≤pivot and i≤j) do
7               i:=i+1;
8           while (a[j]≥pivot and j≥i) do
9               j:=j-1;
10          if (i<j) then
11              interchange(a, i, j);
12      }
13      interchange(a, m, j);
14      return j;
15  }
```

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

```
1   Algorithm interchange(a, i, j)
2   {
3       temp:=a[i];
4       a[i]:=a[j];
5       a[j]:=temp;
6   }
```

Example: working of partition.

Consider the following array of 9 elements.
The partition function is initially invoked as **Partition**(*a, 1, 9*).
The element a[1] i.e., 65 is the partitioning element

a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]	i	j	
65	70	75	80	85	60	55	50	45	2	9	i<j, swap a[i],a[j]
65	45	75	80	85	60	55	50	70	3	8	i<j, swap a[i],a[j]
65	45	50	80	85	60	55	75	70	4	7	
65	45	50	55	85	60	80	75	70	5	6	
65	45	50	55	60	85	80	75	70	6	5	i>j, swap pivot with a[j]
60	45	50	55	**65**	85	80	75	70			Now elements are partitioned about pivot i.e., 65

Now the elements are partitioned about pivot element and the remaining elements are unsorted.

- Using this method of partitioning, we can directly devise a divide and conquer method for completely sorting *n* elements.
- Two sets S1 and S2 are produced after calling partition. Each set can be sorted independently by reusing the function partition.

Following algorithm describes the complete process.

```
1    Algorithm QuickSort(low, high)
2    {
3       if (low<high) then          //if there are more than one element.
4       {
5            j:=partition(a,low,high);      //Partitioning into subproblems
6            // Solve the subproblems
7            QuickSort(a, low, j-1);
8            QuickSort(a, j+1,high);
9       }
10   }
```

Analysis:

- In quicksort, the pivot element we chose divides the array into 2 parts.
  - One of size *k*.
  - Other of size *n-k*.
- Both these parts still need to be sorted.
- This gives us the following relation.

$$T(n) = T(k)+T(n-k)+c.n$$

Where T(n) refers to the time taken by the algorithm to sort n elements.

Worst-case Analysis:

Worst case happens when pivot is the least element in the array.

Then we have *k=1* and *n-k=n-1*

$$\Rightarrow T(n) = T(1) + T(n-1) + c.n$$
$$= T(1) + [\ T(1) + T(n-2) + c.(n-1)\ ] + c.n$$
$$= T(n-2)+2.T(1)+c.(n-1+n)$$
$$= [\ T(1) + T(n-3) + c.(n-2)\ ] + 2.T(1)+c.(n-1+n)$$
$$= T(n-3)+3.T(1)+c.(n-2+n-1+n)$$
$$:$$

Continuing likewise till ith step

$$= T(n-i)+i.T(1)+c.(n-i-1+\ \ldots +n-2+n-1+n$$
$$= T(n-i)+i.T(1)+c.\sum_{j=0}^{i-1}(n-j)$$

This recurrence can go until i=n-1. Substitute i=n-1

$$T(n) = T(1) + (n-1).T(1) + c.\sum_{j=0}^{n-2}(n-j)$$

$$= n.T(1) + c.\sum_{j=0}^{n-2}(n-j)$$

$$= O(n^2)$$

Best-case Analysis:

Best case of Quick Sort occurs when pivot we pick divide the array into two equal parts, in every step.

$$\therefore k = \frac{n}{2}, \quad n-k = \frac{n}{2}, \quad \text{for array of size n}$$

$$\text{We have } T(n) = T(k) + T(n-k) + c.n$$

$$= 2.T\left(\frac{n}{2}\right) + c.n$$

Solving this gives ***O(n log n).***

Randomized Quick Sort:

- Algorithm Quick Sort has an average time of *O(n log n)* and worst case of $O(n^2)$ on *'n'* elements.
- It does not make use of any additional memory like Merge Sort.
- Quick Sort can be modified by using randomizer, so that its performance will be improved.
- While sorting the array *a[p:q]*, pick a random element (from *a[p]...a[q])* as the partition element.
- The randomized algorithm works on any input and runs in an expected *O(n log n)* time, where the expectation is over the space of all possible outcomes of the randomizer.
- The code of randomized quick sort is given below. It is a *Las Vegas algorithm* since it always outputs the correct answer.

```
1    Algorithm RQuickSort(p, q)
2    {
3       if (p<q) then
4       {
5            if ((q-p)>5) then
6                 interchange(a, Random() mod (q-p+1)+p,p);
7            j := partition(a, p, q+1);
8            RQuickSort(p, j-1);
9            RQuickSort(j+1, q);
10      }
11   }
```

- Reason for invoking randomizer only if *(q – p)>5* is
  - Every call to randomizer Random takes a certain amount of time.
  - If there are only a few elements to sort, the time taken by the randomizer may be comparable to the rest of the computation.

Greedy method: General method, applications- knapsack problem, spanning trees, Job sequencing with deadlines, Minimum cost spanning trees, Prim's Algorithm, Kruskal's Algorithms, An optimal randomized algorithm, Optimal Merge Patterns, Single source shortest path problem.

General Method:

- Greedy method is the most straight forward design technique. It is used to solve optimization problems, and can be applied to wide variety of problems.
- These problems have *n* inputs and require us to obtain a subset that satisfies some constraints.
- Any subset that satisfies these constraints is called a feasible solution. A feasible solution that either maximizes or minimizes a given objective function need to be find. A feasible solution that does this is called an optimal solution.
- An algorithm that works in stages can be devised by using greedy method. At each stage, a decision is made whether a particular input is optimal solution or not. This is done by considering the inputs in an order determined by some selection procedure.
- The selection procedure is based on some optimization measure. This measure may be objective function. This version of greedy technique is called subset paradigm.

The subset paradigm technique is described in the following algorithm abstractly.

```
Algorithm Greedy(a,n) //a[1::n] contains the n inputs.
{
        Solution := ϕ ;//Initialize solution
        for i:=1 to n do
        {
                x:=select(a);
                if Feasible(Solution, x) then
                        Solution:=Union(Solution, x);
        }
        return Solution;
}
```

- The function *select* selects and removes an input from *a[]* and the input value is assigned to *x*.
- The function *Feasible* determines whether *x* can be included into the solution or not.
- The function *union* combines *x* with the *solution* and updates.
- The above algorithm gives an abstract view of greedy technique. The functions select, feasible and union need to be properly implemented for solving a particular problem.

Ordering Paradigm: In this paradigm, each decision is made using an optimization criterion that can be computed using decisions already made.

Example 1: Change making:
Pay 67 rupees with fewest numbers of notes.
Construct the solution in stages.
At each stage a note is added, to increase the total amount.
To assure feasibility of the solution, the selected note should not cause the total amount added so far to exceed the desired amount.
The first note selected is 50 rupees. A 50 rupee note or 20 rupee note cannot be selected as a second note. A 10 rupee note is selected as a second note. A 5 rupee note and 2 rupee note are selected as the remaining change. (67=50+10+5+2)

Knapsack problem:

- We are given n objects and a knapsack. Object i has a weight w_i and the knapsack has a capacity m.
- If a fraction x_i, $0 \le x_i \le 1$, of object i is placed into the knapsack, then a profit of $p_i x_i$ is earned.
- The objective is to obtain a filling of the knapsack that maximizes the total profit earned.
- As the knapsack capacity is m, the total weight of all the chosen objects to be at most m.

Formally, the problem can be stated as

Maximize $\sum_{1 \le i \le n} p_i x_i$ ---- (1)

Subject to $\sum_{1 \le i \le n} w_i x_i \le m$ ---- (2)

and $0 \le x_i \le 1, \quad 1 \le i \le n$ ---- (3)

The profits and weights are positive numbers.

A <u>feasible solution</u> is any set $(x_1, \ldots\ldots, x_n)$, which satisfies (2) and (3).

An <u>optimal solution</u> is a feasible solution which maximizes $\sum_{1 \le i \le n} p_i x_i$.

Example: consider the following instances of the knapsack problem

$n=3, m=20, (p_1, p_2, p_3)=(25,24,15)$ and $(w_1, w_2, w_3)=(18,15,10)$

Some feasible solutions are

	(x_1, x_2, x_3)	$\sum w_i x_i$	$\sum p_i x_i$
1	$\left(\frac{1}{2}, \frac{1}{3}, \frac{1}{4}\right)$	9 + 5 + 2.5 = 16.5	12.5 + 8 + 3.75 = 24.25
2	$\left(1, \frac{2}{15}, 0\right)$	18 + 2 + 0 = 20	25 + 3.2 + 0 = 28.2
3	$\left(0, \frac{2}{3}, 1\right)$	0 + 10 + 10 = 20	0 + 16 +15= 31
4	$\left(0, 1, \frac{1}{2}\right)$	0 + 15 + 5 = 20	0 + 24 + 7.5 = 31.5

Out of these 4 feasible solutions, solution 4 yields the maximum profit. Therefore this solution is optimal for the given problem instance.

***In case the sum of all the weights is $\le m$, then $x_i=1, 1 \le i \le n$ is an optimal solution.

The knapsack problem calls for selecting a subset of objects and hence fits the subset paradigm. In addition, this problem also involves the selection of x_i for each object.

The greedy strategies to obtain feasible solutions are

<u>Strategy 1:</u>

- Fill the knapsack by including next the object with largest profit. If the object doesn't fit, then a fraction of it is included to fill the knapsack.

- Thus each time an object is included into the knapsack, we obtain the largest possible increase in profit value. If we follow this method for the above example, solution 2 is formed. But this solution is suboptimal.

  ∴ This greedy method did not yield an optimal solution.

Strategy 2:

- Fill the knapsack by including next the object with least weight.
- If we follow this method in the above example, solution 3 results. This too is suboptimal.

Strategy 3:

- This strategy strives to achieve a balance between the rate at which profit increases and the rate at which capacity is used.
- At each step, we include that object which has maximum profit per unit of capacity used. i.e., the objects are considered in order of the ratio $\frac{p_i}{w_i}$. Solution 4 results, if we follow this approach.

Following algorithm(GreedyKnapsack) obtains solutions corresponding to this strategy. The objects have to be sorted in decreasing order of $\frac{p_i}{w_i}$.

```
Algorithm GreedyKnapsack(m, n)
{
        for i:=1 to m do x[i]:=0.0;
        U:=m;
        for i:=1 to n do
        {
                if (w[i] > U) then break;
                x[i]:=1.0;
                U:=U-w[i];
        }
        if ( i ≤ n ) then x[i]:=U/w[i];
}
```

If time to sort the objects is discarded, then this strategy requires ***O(n)*** time.

Exercise :

1. Find an optimal solution to the knapsack instance n=7, m=15, $(p_1,\ldots,p_7) = (10, 5, 15, 7, 6, 18, 3)$ and $(w_1,\ldots,w_7) = (2, 3, 5, 7, 1, 4, 1)$

Job sequencing with deadlines:

- The problem is the number of jobs, their profit and deadlines will be given and we have to find a sequence of job, which will be completed within its deadline, and it should yield a maximum profit.

Points To remember:

- Each job i is associated with a deadline $d_i \geq 0$ and a profit $p_i > 0$.
- The profit p_i is earned if and only if the job is completed by its deadline.
- To complete a job, one has to process the job on a machine for one unit of time.
- Only one machine is available for processing jobs.
- A feasible solution for this problem is a subset J of jobs such that each job in this subset can be completed by their deadline.
- The value of the feasible solution J is the sum of the profits of the jobs in J or $\sum_{i \in J} p_i$.

- An optimal solution is a feasible solution with maximum value.
- This problem fits the subset paradigm, since it involves identification of a subset.

Since one job can be processed in a single machine. The other job has to be in its waiting state until the job is completed and the machine becomes free.

So the waiting time and the processing time should be less than or equal to the dead line of the job.

Example : Let $n = 4$, $(p_1,p_2,p_3,p_4) = (100,10,15,27)$, $(d_1,d_2,d_3,d_4) = (2,1,2,1)$

Consider the jobs in the nonincreasing order of profits subject to the constraint that the resulting job sequence J is a feasible solution.

Sr.No.	Feasible Solution	Processing Sequence	Profit value
(i)	(1,2)	(2,1)	110
(ii)	(1,3)	(1,3) or (3,1)	115
(iii)	(1,4)	(4,1)	127 is the optimal one
(iv)	(2,3)	(2,3)	25
(v)	(3,4)	(4,3)	42
(vi)	(1)	(1)	100
(vii)	(2)	(2)	10
(viii)	(3)	(3)	15
(ix)	(4)	(4)	27

To formulate the greedy algorithm for an optimal solution, we must formulate an optimization measure to determine how the next job is chosen.

- Choose the objective function $\sum_{i \in J} p_i$ as the optimization measure.
- The next job to include is the one that increases $\sum_{i \in J} p_i$ the most, subject to the constraint that the resulting J is a feasible solution.
- This requires us to consider the jobs in the nonincreasing order of p_i's .
- Add one by one job to J and check whether it is feasible or not.
- To determine whether the given J is feasible solution, check only permutation in which jobs are ordered in nondecreasing order of deadlines.

Solution by using greedy approach

Job considered	Action	J	Assigned slots	Profit
1	Accept	{1}	(1,2)	100
4	Accept	{1,4}	(0,1)(1,2)	127
3	Reject	{1,4}	(0,1)(1,2)	127
2	Reject	{1,4}	(0,1)(1,2)	127

The optimal solution is J = {1, 4}

With a profit of 127.

Ex 2: Let n = 5, $(p_1, \ldots, p_5) = (20, 15, 10, 5, 1)$ and $(d_1, \ldots, d_5) = (2, 2, 1, 3, 3)$

Job considered	Action	J	Assigned slots	Profit
1	Accept	{1}	(1,2)	20
2	Accept	{1,2}	(0,1)(1,2)	35
3	Reject	{1,2}	(0,1)(1,2)	35
4	Accept	{1,2,4}	(0,1)(1,2)(2,3)	40
5	Reject	{1,2,4}	(0,1)(1,2)(2,3)	40

The optimal solution is J = {1, 2, 4}
With a profit of 40.

Ex 3: Let n = 7, $(p_1, \ldots, p_7) = (3, 5, 20, 18, 1, 6, 30)$ and $(d_1, \ldots, d_7) = (1, 3, 4, 3, 2, 1, 2)$

Job considered	Action	J	Assigned slots	Profit

The optimal solution is J =
With a profit of _____

A high level description of the greedy algorithm for Job Sequencing with deadlines problem is shown in the algorithm. This algorithm constructs an optimal set *J* of jobs that can be processed by their due times. The selected jobs can be processed in the order given by the following theorem.

" Let J be a set of k jobs and $\sigma = (i_1, i_2, \ldots i_k)$ be a permutation of jobs in J such that $di_1 \leq di_2 \leq \ldots \leq di_k$. Then J is a feasible solution iff the jobs in J can be processed in the order σ without violating any deadline."

```
Algorithm GreedyJob(d, J, n)
// J is a set of Jobs that can be completed by their deadlines.
{
        J := { 1 } ;
        for i =2 to n do
        {
                if ( all jobs in J U { i } can be completed by their deadlines ) then J := J U { i };
        }
}
```

To write complete algorithm,

- We can use an array *d[1:n]* to store the deadlines of the jobs in the order of their p-values.
- The set J itself can be represented by a one-dimensional array *J[1:k]* such that *J[r]*, $1 \le r \le k$ are the jobs in *J* and $d[J[1]] \le d[J[2]] \le \ldots\ldots \le d[J[k]]$.
- To test weather $J \cup \{i\}$ is feasible, we have just to insert *i* into *J* preserving the deadline ordering and then verify that $d[J[1]] \le r,\ 1 \le r \le k+1$.
- The insertion of *i* into *J* is simplified by the use of a fictitious *job 0* with *d[0]=0, J[0]=0*.

The algorithm that results from the above discussion is

Algorithm JS(*d, J, n*)

```
//The jobs are ordered such that p[1]>p[2] ...>p[n]
//J[i] is the i^th job in the optimal solution also at termination d[J[ i]] ≤ d[ J[i+1]],1 < i < k
{
   d[0]:= J[0]:=0;
   J[1]:=1;
   k=1;
   for i :=2 to n do
   {    // consider jobs in nonincreasing order of P[i]; find the position for i and check feasibility insertion
        r:=k;
        while ( (d[J[r]] > d[i]) and (d[J[r]] ≠ r) ) do r := r – 1;
        if ( (d[J[r]] ≤ d[i]) and (d[i] > r)) then
        {
            for q:=k to (r+1) step –1 do J [q+1]:=J[q];
            J[r+1]:=i; k:=k+1;
        }
   }
   return k;
}
```

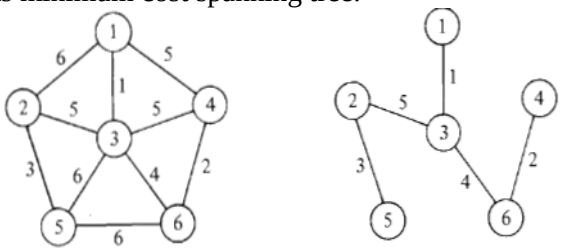
Complexity:

- Complexity of Job Sequencing can be measured in terms of two parameters. They are *n*, the number of jobs and S, the number of jobs included in the solution.
- The while loop iterates for at most *k* times. Each iteration takes $\Theta(1)$.
- If the if condition is true then the time taken to insert job *I* is $\Theta(k-r)$.
- Hence, the total time for each iteration of for loop is time $\Theta(k)$.
- Loop iterates for $(n-1)$ times.
- If *s* is the final value of *k*, then the time needed by algorithm JS is – $\Theta(sn)$
- Since sn, the worst case time, as a function of n alone is - $\Theta(n^2)$

Minimum-cost spanning trees:

- Let ***G=(V, E)*** be an undirected connected graph. A subgraph $t=(V,E')$ of **G** is a spanning tree of **G** iff ***t*** is a tree.
- In practical situations, the edges have weights assigned to them. The cost of spanning trees is the sum of the costs of the edges in that tree.
- The spanning tree of **G** represents all feasible choices. A minimum-cost spanning tree is a spanning tree with minimum cost.

- Figure shows a graph and its minimum cost spanning tree.



- Since the identification of minimum-cost spanning tree involves the selection of a subset of the edges, this problems fits the subset paradigm.
- A greedy method to obtain a minimum-cost spanning tree builds this tree edge by edge.
- The next edge to include is chosen according to some optimization criterion.
- The simplest such criterion is to choose an edge that result in a minimum increase in the sum of the costs of the edges so far included.

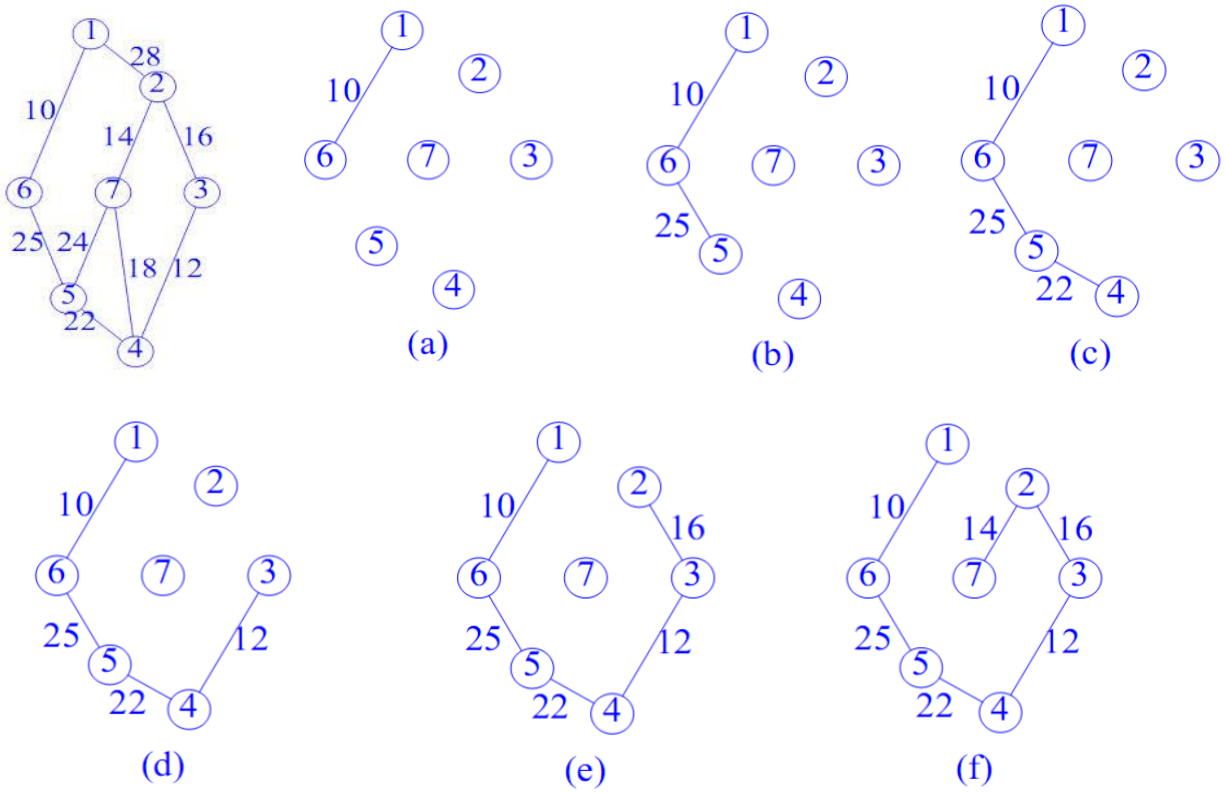
There are two possible ways to interpret this criterion. Their respective algorithms are

1. Prim's algorithm
2. Kruskal's algorithm

Prim's Algorithm:

- The set of edges selected by this algorithm should form a tree.
- Start from an arbitrary vertex and store it in A.
- Thus, if A is the set of edges selected so far, then A forms a tree.
- The next edge (u, v) to be included in A is a minimum-cost edge not in A with the property that $AU\{(u,v)\}$ is also a tree.

Following figures shows the working of prim's method on a graph.



Pseudocode algorithm to find a minimum cost spanning tree:

```
Algorithm Prim (E, cost, n, t)
//E is the set of edges in G. cost[1:n, 1:n] is the cost adjacency matrix.
//A MST is computed and stored as a set of edges in the array t[1:n-1, 1:2]. The final cost is returned.
{
    Let (k, l) be an edge with mincost in E;
    mincost := cost[k, l];
    t[1,1]:=k, t[1,2]:=l;
    for i:=1 to n do
        if cost[i,l] < cost[i,k] then near[i]:=l;
        else near[i]:=k;
    near[k]:=near[l]:=0;
    for i:=2 to n-1 do
    {
        let j be an index such that near[j]≠0 and cost[i,near[j]] is minimum;
        t[i,1]:=j; t[i,2]:=near[j];
        mincost:=mincost+cost[j,near[j]];
        near[j]:=0;
        for k:=1 to n do
            if (near[k]≠0 and cost[k,near[k]]>cost[k,j]) then near[k]:=j;
    }
    return mincost;
}
```

- This algorithm will start with a tree that includes only a minimum cost edge of *G*. Then edges are added to this tree one by one.
- The next edge (*i*, *j*) to be added is such that
  - *i* is a vertex already included in the tree.
  - *j* is a vertex not yet included.
  - The cost of (*i*, *j*) is minimum among all edges (*k*, *l*) such that vertex *k* is in the tree and vertex *l* is not in the tree.
  - To determine this edge (*i*, *j*) efficiently, we associate with each vertex j not yet included in the tree a value *near[j]*.
  - For all vertices *j* that are already in the tree, set *near[j]=0.*
  - The next edge to include is defined by the vertex *j* such that *near[j]≠0* and *cost[j, near[j]]* is minimum.

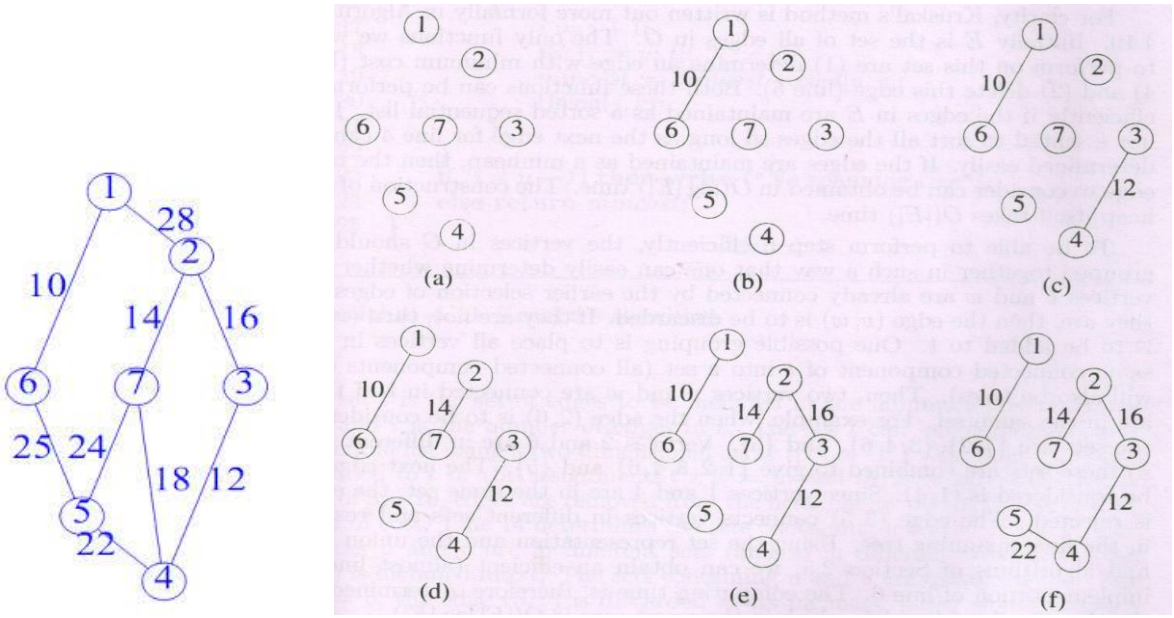
Running Time:

- Selecting edge with minimum cost : O(|E|)
- Initialization of *near[]* takes : $\Theta(n)$
- For loop runs for all the vertices i.e., *n* times and it includes updation of *near[]* which runs for *n* times. Cost is : $O(n^2)$
- Hence, prim runs in $O(n^2)$ time.

Kruskal's algorithm:

- In this algorithm, the edges of the graph are considered in nondecreasing order of cost.
- The set of *t* edges selected for the spanning tree be such that it is possible to complete *t* into a tree. Thus *t* may not be a tree at all stages in the algorithm.
- The set of edges selected are generally a forest since the set of edges *t* can be completed into a tree iff there are no cycles in *t*.
- This interpretation also results in a Minimum Spanning Tree.

Example: following figure shows stages in kruskal's algorithm. It begins with no edges selected.



Consider the graph shown above.

- We begin with no edges selected. Figure (a) shows the graph with no edges selected.
- First edge considered is (1,6). It is included in the spanning tree and it yields the graph in figure (b)
- Next, the edge (3,4) is considered and included in the tree and it shown in figure (c).
- Next, the edge (2,7) is considered and included in the tree and it shown in figure (d).
- Next, the edge (2,3) is considered and included in the tree and it shown in figure (e).
- Of the edges not yet considered, (7, 4) has the least cost. It is considered next. Its inclusion in the tree results in a cycle, so this edge is discarded.
- Next, the edge (5,4) is considered and included in the tree and it shown in figure (f).
- As edge (5,7) forms a cycle, addition of edge (5,6) completes the MST.

Kruskal's algorithm can be formally written as

```
t := Ø;
while( ( t has less than n-1 edges ) and ( E ≠ Ø ) ) do
{
   Choose an edge (v, w) from E of lowest cost;
   Delete (v, w) from E;
   If ((v, w) does not create a cycle in t) then
         Add (v, w) to t;
   else discard (v, w);
}
```

While implementing this, the steps

1. Determining minimum edge and deleting the edge can be performed efficiently by constructing min-heap with edge costs and taking root of it. This takes O(|E|) time for heap and O(log|E|) for selecting next edge and reheap.
2. The selected edge will form a cycle in *t* or not can be identified with find operation. And if it is not forming a cycle then adding this to the tree can be performed using union operation, by representing nodes as sets. Which are the operations of linear complexity So the total algorithm will take an O(|E| log(|E|). Where |E| is the edges of G.

Pseudo code for kruskal's algorithm:

```
Algorithm kruskal(E,cost,n,t)
//E→set of edges in G has 'n' vertices.
//cost[u,v]→cost of edge (u,v). t→set of edges in minimum cost spanning tree
// the minimum cost is returned.
{
        Construct a heap out of the edge costs;
        for i:=1 to n do parent[i]:= -1;
        i:=0; mincost:=0.0;
        while((i<n-1)and (heap not empty)) do
        {
                delete a minimum cost edge (u,v) from the heap;
                j:=find(u); k:=find(v);
                if ( j ≠ k) then
                {
                        i:=i+1;
                        t[i,1]:=u; t[i,2]:=v;
                        mincost=mincost+cost[u,v];
                        union(j,k);
                }
        }
        if ( i ≠ n-1) then write("No spanning tree"); else return minimum cost;
}
```

Analysis :
The time complexity of minimum cost spanning tree algorithm in worst case is O(|E|log|E|), where E is the edge set of G.

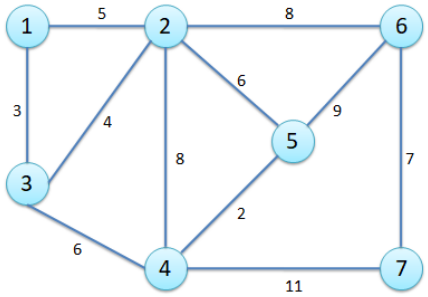
Optimal Randomized Algorithm :

Any algorithm for finding the minimum-cost spanning tree of a given graph G(V, E) will take $\Omega(|V| + |E|)$ time in the worst case.
Reason : since it has to examine each node and each edge at least once before determining the correct answer.

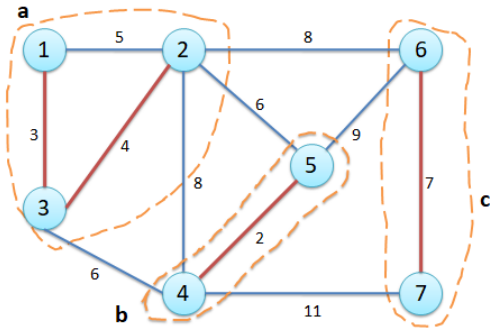
An optimal randomized algorithm which takes O(|V| + |E|) can be devised as follows

1. Randomly sample *m* edges from G
2. Let G^1 be the induced sub graph; i.e., G^1 has V as its node set and the sampled edges in the edge set. The subgraph G^1 need not be connected. Recursively find Minimum Cost Spanning Tree for each component of G^1. Let F be the resultant minimum cost spanning forest of G^1.
3. Using F eliminate certain edges (heavy edges) of G that cannot possibly in a Minimum Cost Spanning Tree. Let G^{11} be the graph that results from G after elimination of F-heavy edges.
4. Recursively find Minimum Cost Spanning Tree for G^{11}.

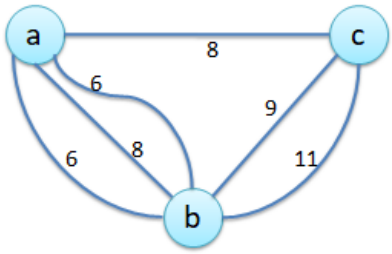
Example :
Given input graph G is

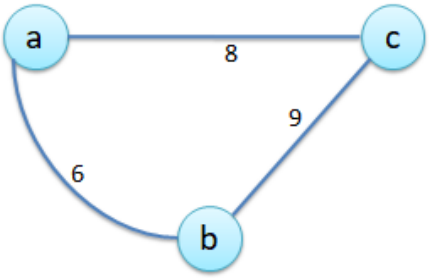


Randomly sampled 4 edges (1,3) , (2,3), (4,5) and (6,7)
Selected edges are forming three trees (forest). They should be minimum spanning trees.

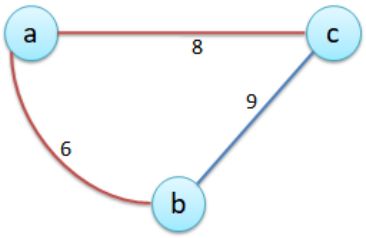


Using F translate the given Graph G into G1 and remove the heavy edges.

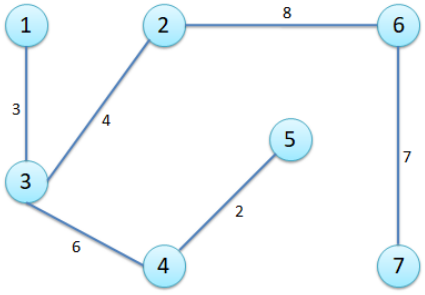




Apply the same process recursively on obtained graph



The resultant minimum spanning tree is

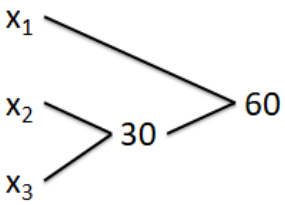


Optimal Merge Patterns

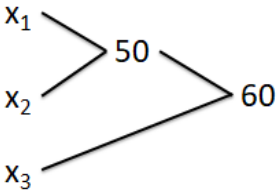
Problem : Determining an optimal way (one requiring the fewest comparisons) to pair-wise merge n sorted files.

- Merging two sorted files having *n* and *m* records to obtain one sorted file takes *O(n+m)* time.
- When more than two sorted files are to be merged together, the merge can be accomplished by repeatedly merging sorted files in pairs.
- Given n sorted files, there are many ways in which to pairwise merge them into a single sorted file.
- Different pairings require different computing time.

Eg: Consider three files x1, x2, x3 with record lengths 30, 20, 10.



This merge pattern is taking 90 comparisons



This merge pattern is taking 110 comparisons.

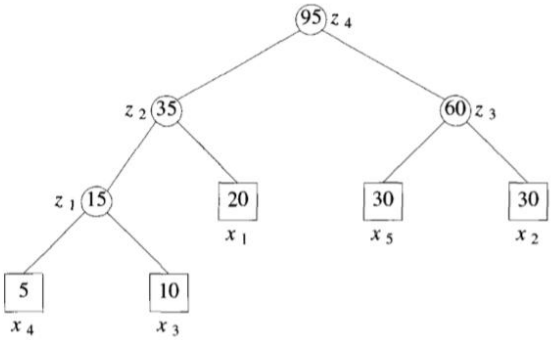
Greedy method to obtain an optimal merge pattern :

Selection criterion: since merging *n*-record and *m*-record files requires *n+m* record moves, the obvious choice is at each step merge the two smallest size files together.

Eg: no.of files = 5
(x1, x2, x3, x4, x5) sizes (20, 30, 10, 5, 30)
Merge x3 and x4 to get z1 ➔ |z1| = 15
Merge z1 and x1 to get z2 ➔ |z2| = 35
Merge x2 and x5 to get z3 ➔ |z3| = 60
Merge z2 and z3 to get z4 ➔ |z4| = 95

The total number of record moves = 205

It can be represented as a binary merge tree.



- The leaf nodes are drawn as squares and represent the given five files. These nodes are called external nodes.
- Remaining nodes are drawn as circles and are called internal nodes.
- Each internal node has exactly two children.
- The number in each node is the length (no.of records) of the file represented by that node.

- The external node x4 is at a distance of 3 from the root node z4. i.e., the records of file x4 are moved three times, once to get z1, once to get z2 and finally to get z4.
- Total number of record moves for the binary merge tree is $\sum_{i=1}^{n} d_i\, q_i$
  - d*i* is the distance from the root to the external node for file xi
  - q_i is the length of x_i
- This sum is the weighted external path length of the tree.

Algorithm to generate a 2-way merge tree :

```
node = record {
        node    *lchild, *rchild;
        integer weight;
}
Algorithm Tree(n)
{
        //list is a global list of n single node binary trees
        for i := 1 to n-1 do
        {
                pt := new node;
                pt->lchild := Least(list);
                pt->rchild := Least(list);
                pt->weight := pt->lchild->weight + pt->rchild->weight;
                insert(list, pt);
        }
        return (Least(list))
}
```

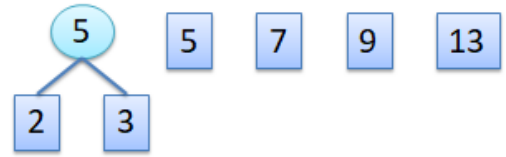
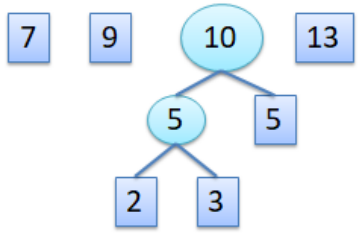
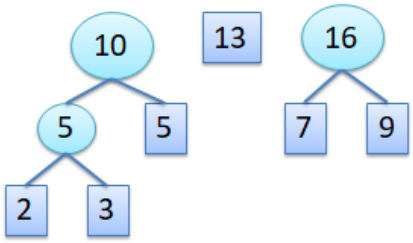
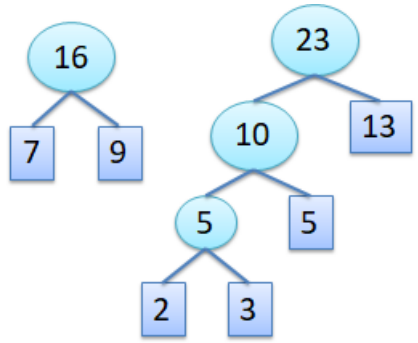
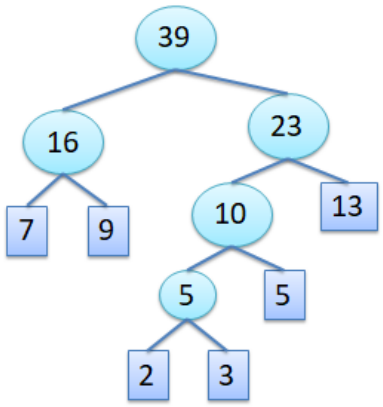
- Input to this algorithm is list of n trees. Each node in a tree has 3 fields lchild, rchild, weight.
- Initially, each tree in list has exactly one node.
- Least(list) function finds a tree in list whose root has least weight.
- Insert() function is used to insert a node into the list.

Analysis :

- Main for loop is executed *(n – 1)* times
- If list is kept in nondecreasing order according to weight value in the roots, then **Least(list)** requires O(1) time
- **Insert(list, t)** can be done in O(n) time.
- Hence, the total time taken is $O(n^2)$

Eg: Trace the algorithm for 6 files with lengths 2, 3, 5, 7, 9, 13

Iteration	List
Initially	2 3 5 7 9 13

1	
2	
3	
4	
5	

Optimal merge pattern

1. Merge files whose lengths are 2 and 3
2. Merge files whose lengths are 5 and 5
3. Merge files whose lengths are 7 and 9
4. Merge files whose lengths are 10 and 13
5. Merge files whose lengths are 16 and 23

Exercise :

Find an optimal binary merge pattern for ten files whose lengths are
28, 32, 12, 5, 84, 53, 91, 35, 3, 11

Single Source Shortest Paths

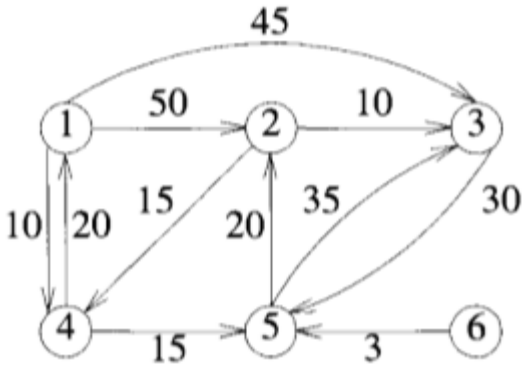
- **Problem** : Given a directed graph G = (V, E), a weighting function *cost* for the edges of G, and a source vertex v_0, The problem is to determine the shortest paths from v_0 to all the remaining vertices of G.
- It is assumed that all the weights are positive.
- The shortest path between v_0 and other node v is an ordering among a subset of the edges. Hence this problem fits the ordering paradigm.
- A multistage solution must be conceived to formulate a greedy-based algorithm to generate shortest paths.
- Here the optimization measure is, each individual path must be of minimum length.
- The greedy way to generate the shortest paths from v_0 to the remaining vertices is to generate these paths in increasing order of the path length.
- First, a shortest path to the nearest vertex is generated, and then a shortest path to the second nearest vertex is generated, and so on.
- For eg : nearest vertex to (v0 = 1) is 4 cost[1, 4] = 10
  Second nearest vertex to (v0 = 1) is 5 distance = 25 path 1, 4, 5 is generated.
- Inorder to generate the shortest paths in this order, we need to determine
  1. The next vertex to which a shortest path must be generated &
  2. A shortest path to this vertex.
- Greedy algorithm to generate shortest paths is

```
Algorithm ShortestPaths(v, cost, dist, n)
{
// dist[j], 1≤j ≤n, is set to the length of the shortest path from vertex v to vertex j in a digraph G with
// n  vertices. dist[v] is set to zero. G is represented by its cost adjacency matrix cost[l :n,1:n].

        for i := 1 to n do
        {
                S[i] := false;
                dist[i] := cost[v, i];
        }
        S[v] := true;
        for num := 1 to n-1 do
        {
                choose u from among those vertices not in S such that dist[u] is minimum;
                S[u] := false;
                for (each w adjacent to u with S[w]=false) do
                        //update the distances
                        if (dist[w] > dist[u] + cost[u,w]) then
                                dist[w] := dist[u] = cost[u, w];
        }
}
```

- Let S denote the set of vertices (including v_0) to which the shortest paths have already been generated.
- For w, not in S, let dist[w] be the length of the shortest path starting from v_0, going through only those vertices that are in S, and ending at w.

Example : Use algorithm ShortestPaths to obtain in non-decreasing order the lengths of the shortest paths from vertex 1to all remaining vertices in the digraph.



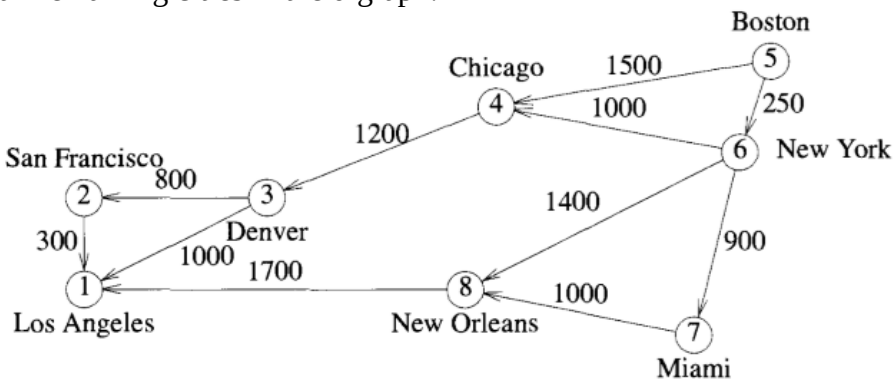
$V_0 = 1$

$$\text{Cost matrix} = \begin{bmatrix} 0 & 50 & 45 & 10 & \infty & \infty \\ \infty & 0 & 13 & 15 & \infty & \infty \\ \infty & \infty & 0 & \infty & 30 & \infty \\ 20 & \infty & \infty & 0 & 15 & \infty \\ \infty & 20 & 35 & \infty & 0 & \infty \\ \infty & \infty & \infty & \infty & 3 & 0 \end{bmatrix}$$

Trace of the algorithm

Iteration	S	Vertex selected	Distance					
			[1]	[2]	[3]	[4]	[5]	[6]
Initially	--	--	0	50	45	10	∞	∞
1	{1}	4	0	50	45	10	25	∞
2	{1, 4}	5	0	45	45	10	25	∞
3	{1,4,5}	2	0	45	45	10	25	∞
4	{1,2,4,5}	3	0	45	45	10	25	∞
5	{1,2,3,4,5}	6	0	45	45	10	25	∞
6	{1,2,3,4,5,6}							

Example 2: Use algorithm ShortestPaths to obtain in non-decreasing order the lengths of the shortest paths from city Boston to all remaining cities in the digraph.



Length-adjacency matrix

	1	2	3	4	5	6	7	8
1	0							
2	300	0						
3	100	800	0					
4			1200	0				
5				1500	0	250		
6				1000		0	900	1400
7							0	1000
8	1700							0

Tracing

Iteration	S	Vertex selected	Distance LA [1]	SF [2]	DEN [3]	CHI [4]	BOST [5]	NY [6]	MIA [7]	NO [8]
Initial	--	----	+∞	+∞	+∞	1500	0	250	+∞	+∞
1	{5}	6	+∞	+∞	+∞	1250	0	250	1150	1650
2	{5,6}	7	+∞	+∞	+∞	1250	0	250	1150	1650
3	{5,6,7}	4	+∞	+∞	2450	1250	0	250	1150	1650
4	{5,6,7,4}	8	3350	+∞	2450	1250	0	250	1150	1650
5	{5,6,7,4,8}	3	3350	3250	2450	1250	0	250	1150	1650
6	{5,6,7,4,8,3}	2	3350	3250	2450	1250	0	250	1150	1650
	{5,6,7,4,8,3,2}									