Asymptotic analysis of an algorithm refers to defining the mathematical boundation/framing of its run-time performance. Using asymptotic analysis, we can very well conclude the best case, average case, and worst case scenario of an algorithm.

Asymptotic analysis is input bound i.e., if there's no input to the algorithm, it is concluded to work in a constant time. Other than the "input" all other factors are considered constant.

Asymptotic analysis refers to computing the running time of any operation in mathematical units of computation. For example, the running time of one operation is computed as *f*(n) and may be for another operation it is computed as *g*(n2). This means the first operation running time will increase linearly with the increase in **n** and the running time of the second operation will increase exponentially when **n** increases. Similarly, the running time of both operations will be nearly the same if **n** is significantly small.

Usually, the time required by an algorithm falls under three types −

* **Best Case** − Minimum time required for program execution.
* **Average Case** − Average time required for program execution.
* **Worst Case** − Maximum time required for program execution.

**Asymptotic Notations/ mathematical notation used in analysis of algo**

Following are the commonly used asymptotic notations to calculate the running time complexity of an algorithm.

* Ο Notation
* Ω Notation
* θ Notation

Big Oh Notation, Ο

|  |  |
| --- | --- |
| The notation Ο(n) is the formal way to express the upper bound of an algorithm's running time. It measures the worst case time complexity or the longest amount of time an algorithm can possibly take to complete.  For example, for a function ***f*(n)** | Big O Notation |

Ο(*f*(n)) = { *g*(n) : there exists c > 0 and n0 such that *f*(n) ≤ c.*g*(n) for all n > n0. }

Omega Notation, Ω

|  |  |
| --- | --- |
| The notation Ω(n) is the formal way to express the lower bound of an algorithm's running time. It measures the best case time complexity or the best amount of time an algorithm can possibly take to complete. | Omega Notation |

For example, for a function ***f*(n)**

Ω(*f*(n)) ≥ { *g*(n) : there exists c > 0 and n0 such that *g*(n) ≤ c.*f*(n) for all n > n0. }

Theta Notation, θ

|  |  |
| --- | --- |
| The notation θ(n) is the formal way to express both the lower bound and the upper bound of an algorithm's running time. It is represented as follows | Theta Notation |

θ(*f*(n)) = { *g*(n) if and only if *g*(n) = Ο(*f*(n)) and *g*(n) = Ω(*f*(n)) for all n > n0. }

Common Asymptotic Notations

Following is a list of some common asymptotic notations −

|  |  |  |
| --- | --- | --- |
| constant | − | Ο(1) |
| logarithmic | − | Ο(log n) |
| linear | − | Ο(n) |
| n log n | − | Ο(n log n) |
| quadratic | − | Ο(n2) |
| cubic | − | Ο(n3) |
| polynomial | − | nΟ(1) |
| exponential | − | 2Ο(n) |

**BASIC DIFFERENCES BETWEEN SPACE COMPLEXITY AND TIME COMPLEXITY**

**SPACE COMPLEXITY:**

The space complexity of an algorithm is the amount of memory it requires to run to completion.

the space needed by a program contains the following components:

1)  Instruction space:

-stores the executable version of programs and is generally fixed.

2) Data space:

It contains:

a)  Space required by constants and simple variables.Its space is fixed.

b)  Space needed by fixed size stucture variables such as array and structures.

c)  dynamically allocated space.This space is usually variable.

3) enviorntal stack:

-Needed to stores information required to reinvoke suspended processes or functions.

the following data is saved on the stack

- return address.

-value of all local variables

-value of all formal parameters in the function..

**TIME COMPLEXITY:**

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

To measure the time complexity we can count all operations performed in an algorithm and if we know the time taken for each operation then we can easily compute the total time taken by the algorithm.This time varies from system to system.

Our intention is to estimate execution time of an algorithm irrespective of the computer on which it will be used. Hence identify the key operation and count such operation performed till the program completes its execution.

The time complexity can be expressd as a function of a key operation performed.

The space and time complexity is usually expressed in the form of function f(n),where n is the input size for a given instance of a problem being solved.

f(n) helps us to predict the rate of growthof complexity that will increase as size of input to the problem increases.

f(1) also helps us to predict complexity of two or more algorithms in order ro find which is more efficient.

# Sorting Methods(internal and external )

The function of sorting or ordering a list of objects according to some linear order is so fundamental that it is ubiquitous in engineering applications in all disciplines. There are two broad categories of sorting methods: **Internal**sorting takes place in the main memory, where we can take advantage of the random access nature of the main memory; **External** sorting is necessary when the number and size of objects are prohibitive to be accommodated in the main memory.

**The Problem:**

* Given records *r*1, *r*2,..., *r*n, with key values *k*1, *k*2,..., *k*n, produce the records in the order

*r*i1, *r*i2,..., *r*in,

such that

*k*i1$\displaystyle \leq$*k*i2$\displaystyle \leq$...$\displaystyle \leq$*k*in 

* The complexity of a sorting algorithm can be measured in terms of

number of algorithm steps to sort *n* records number of comparisons between keys (appropriate when the keys are long character strings)

number of times records must be moved (appropriate when record size is large)

* Any sorting algorithm that uses comparisons of keys needs at least O(n log n) time to accomplish the sorting.

|  |  |
| --- | --- |
| Sorting Methods | |
| **Internal** | **External** |
| **(In memory)** | **Appropriate for secondary storage** |
| quick sort |  |
| heap sort | mergesort |
| bubble sort | radix sort |
| insertion sort | polyphase sort |
| selection sort |  |
| shell sort |  |

External sorting is a term for a class of sorting algorithms that can handle massive amounts of data. External sorting is required when the data being sorted do not fit into the main memory of a computing device (usually RAM) and instead they must reside in the slower external memory (usually a hard drive). External sorting typically uses a hybrid sort-merge strategy. In the sorting phase, chunks of data small enough to fit in main memory are read, sorted, and written out to a temporary file. In the merge phase, the sorted sub-files are combined into a single larger file.

One example of external sorting is the external merge sort algorithm, which sorts chunks that each fit in RAM, then merges the sorted chunks together. We first divide the file into **runs** such that the size of a run is small enough to fit into main memory. Then sort each run in main memory using merge sort sorting algorithm. Finally merge the resulting runs together into successively bigger runs, until the file is sorted.

When all data that needs to be sorted cannot be placed in-memory at a time, the sorting is called [external sorting](http://en.wikipedia.org/wiki/External_sorting). External Sorting is used for massive amount of data. Merge Sort and its variations are typically used for external sorting. Some external storage like hard-disk, CD, etc is used for external storage.  
 When all data is placed in-memory, then sorting is called internal sorting.

**FINDING THE MAXIMUM AND MINIMUM using DIVIDE AND CONQUER Strategy**

*FINDING THE MAXIMUM AND MINIMUM*

The problem is to find the maximum and minimum items in a set of n elements.

*1.*      *Algorithm for straight forward maximum and minimum*

StraightMaxMin(a,n,max,min)

// set max to the maximum and min to the minimum of a[1:n].

{

     max := min := a[1];

     for i := 2 to n do

     {

           if(a[i] > max) then max := a[i];

           if(a[i] > min) then min := a[i];

     }

}

*Analyzing the Straight Forward Method*

                In analyzing the time complexity of this algorithm, we have to concentrate on the number of element comparisons. This algorithm requires 2(n-1) element comparisons in the best, average, and worst cases. An immediate improvement is possible by realizing that the comparison a[i] < min is necessary only when a[i]>max is false.

                Now the Best case occurs when the elements are in increasing order. The number of element comparisons is n-1. The worst case occurs when the element are in decreasing order. In this case number of comparisons is 2(n-1).

*FINDING THE MAXIMUM AND MINIMUM using DIVIDE AND CONQUER Strategy*

·         *What is DIVIDE AND CONQUER Strategy?*

Given a function to compute on n inputs the *divide-and-conquer* strategy suggest splitting the inputs into k distinct subsets, 1 < K ≤ n, yielding k sub problems. These Sub problems must be solved, and then a method must be found to combine sub solutions into a solution of the whole. If the Sub problems are still relatively large, then the *divide-and-conquer* strategy can possibly be reapplied. Often the sub problems resulting from a *divide-and-conquer* design are the same type as the original problem. For those cases the reapplication of the *divide-and-conquer* principle is naturally expressed by a recursive algorithm. Now smaller sub problems of the same kind are generated until eventually sub problems that are small enough to be solved without splitting are produced.

A *divide-and-conquer*algorithm for this problem would proceed as follows:  Let P = (n,a[i],….,a[j]) denote an arbitrary instance of the problem. Here n is the number of elements in the list a[i],….,a[j] and we are interested in finding the maximum and minimum of this list. Let small(P) be true when n ≤ 2. In this case, the maximum and minimum are a[i] if n = 1. If n = 2, the problem can be solved by making one comparison.

If the list has more than two elements, P has to be divided into smaller instances. For example, we might divide P into the two instances P1 = (n/2,a[1],….,a[n/2]) and P2 = (n - n/2,a[n/2 + 1],….,a[n]). After having divided P into two smaller sub problems, we can solve them by recursively invoking the same divide and conquer algorithm.

Now the question is How can we combine the Solutions for P1 and P2 to obtain the solution for P? If MAX(P) and MIN(P) are the maximum and minimum of the elements of P, then MAX(P) is the larger of MAX(P1) and MAX(P2) also MIN(P) is the smaller of MIN(P1) and MIN(P2).

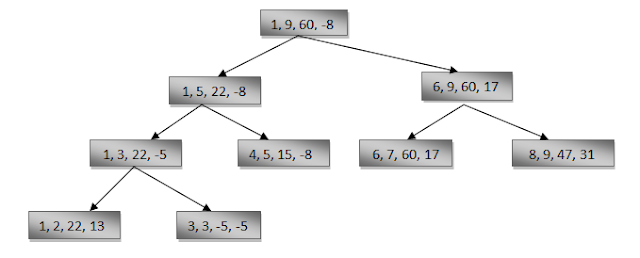
MaxMin is a recursive algorithm that finds the maximum and minimum of the set of elements {a(i),a(i+1),…,a(j)}. The situation of set sizes one (i=j) and two (i=j-1) are handled separately. For sets containing more than two elements, the midpoint is determined and two new sub problems are generated. When the maxima and minima of this sub problems are determined, the two maxima are compared and the two minima are compared to achieve the solution for the entire set.

The procedure is initially invoked by the statement MaxMin(1,n,x,y). for this algorithm each node has four items of information: i, j, max, min. Suppose we simaulate MaxMin on the following nine elements:

a: [1]  [2]  [3]  [4]  [5]  [6]  [7]  [8]  [9]

    22   13   -5   -8   15  60   17   31  47

A good way of keeping track of recursive calls is to build a tree by adding a node each time a new call is made. On the array a[ ] above, the following tree is produced.

[](http://3.bp.blogspot.com/-BntLQuCaTaM/UDtYks9NG0I/AAAAAAAAAI4/c3L99uVf2oE/s1600/Capture.PNG)

            We see that the root node contains 1 and 9 as the values of i and j corresponding to the initial call to MaxMin. This execution produces two new call to MaxMin, where i and j have the values 1, 5 and 6, 9, and thus split the set into two subsets of the same size. From the tree we can immediately see that the maximum depth of recursion is four (including the first call).

*1.*      *Algorithm for  maximum and minimum using divide-and-conquer*

MaxMin(i, j, max, min)

// a[1:n] is a global array. Parameters i and j are integers,   // 1≤i≤j≤n. The effect is to set max and min to the largest and  // smallest values in a[i:j].

{

     if (i=j) then max := min := a[i]; //Small(P)

     else if (i=j-1) then // Another case of Small(P)

          {

                if (a[i] < a[j]) then max := a[j]; min := a[i];

                else max := a[i]; min := a[j];

          }

     else

     {

           // if P is not small, divide P into sub-problems.

           // Find where to split the set.

           mid := ( i + j )/2;

           // Solve the sub-problems.

           MaxMin( i, mid, max, min );

           MaxMin( mid+1, j, max1, min1 );

           // Combine the solutions.

           if (max < max1) then max := max1;

           if (min > min1) then min := min1;

     }

}

*Complexity:*

     Now what is the number of element comparisons needed for MaxMin? If T(n) represents this number, then the resulting recurrence relation is

                           0                                              n=1

T(n) =      1                                              n=2

                T(n/2) + T(n/2) + 2           n>2

     When n is a power of two, n = *2*k

-for some positive integer k, then

  T(n) = 2T(n/2) + 2

           = 2(2T(n/4) + 2) + 2

           = 4T(n/4) + 4 + 2

           .

           .

           .

           = *2*k-1 T(2) + ∑(1≤i≤k-1) *2*k

           = *2*k-1 +*2*k – 2

           = 3n/2 – 2 = O(n)

Note that 3n/2 – 2 is the best, average, worst case number of comparison when n is a power of two.

*Comparisons with Straight Forward Method:*

        Compared with the 2n – 2 comparisons for the Straight Forward method, this is a saving of 25% in comparisons. It can be shown that no algorithm based on comparisons uses less than 3n/2 – 2 comparisons.

 Lets say we have given an array {100,23,10,13,1,109}, you can see max is 109 and min is 1 here so in this case of naive search it will take O(n) time to search min and max .   
  
Best case will be when array is sorted in non-decreasing order e.g.{1,10,13,23,100,109} no. of comparisons will be **1+(n-2)** and when array is sorted in non-increasing order e.g. {109,100,23,13,10,1} no. of comparisons will be **1+2(n-2)**and worst time complexity will be**T(n)=**O(n)O(n)  
  
Now lets see how no. of comparisons decreases in case of divide and conquer or tournament approach, in this approach we divide the array into two parts and compare the max and min of the the two parts to get the maximum and the minimum of the the whole array, this is recursive in nature.see the pseudo code below.  
  
**MaxAndMinUsingDevideAndConquer(array, size)   
  
1.if size= 1       return current element as both max and min  //base condition   
2.else if size= 2       one comparison to determine max and min //base condition  
3.else    /\* if size > 2  find mid and call recursive function \*/  
    recur for max and min of left half       recur for max and min of right half.  
    one comparison determines to max of the two subarray, update max.  
    one comparison determines min of the two subarray, update min.  
4. finally return or print the min/max in whole array**.  
   
Recurrence relation can be written as **T(n)=2(T/2)+2** solving which give you   
**T(n)  =3/2n-2** which is the exact no. of comparisons but still the worst time complexity will be**T(n)=**O(n)O(n) and best case time complexity will be O(1)O(1) when you have only one element in array, which will be candidate for both max and min.

**Problem Statement**

The Max-Min Problem in algorithm analysis is finding the maximum and minimum value in an array.

**Solution**

To find the maximum and minimum numbers in a given array ***numbers[]*** of size **n**, the following algorithm can be used. First we are representing the **naive method** and then we will present **divide and conquer approach**.

**Naïve Method**

Naïve method is a basic method to solve any problem. In this method, the maximum and minimum number can be found separately. To find the maximum and minimum numbers, the following straightforward algorithm can be used.

**Algorithm: Max-Min-Element (numbers[])**

max := numbers[1]

min := numbers[1]

for i = 2 to n do

if numbers[i] > max then

max := numbers[i]

if numbers[i] < min then

min := numbers[i]

return (max, min)

**Analysis**

The number of comparison in Naive method is **2n - 2**.

The number of comparisons can be reduced using the divide and conquer approach. Following is the technique.

**Divide and Conquer Approach**

In this approach, the array is divided into two halves. Then using recursive approach maximum and minimum numbers in each halves are found. Later, return the maximum of two maxima of each half and the minimum of two minima of each half.

In this given problem, the number of elements in an array is y−x+1, where **y** is greater than or equal to **x**.

Max−Min(x,y) will return the maximum and minimum values of an array numbers[x...y],

**Algorithm: Max - Min(x, y)**

if x – y ≤ 1 then

return (max(numbers[x], numbers[y]), min((numbers[x], numbers[y]))

else

(max1, min1):= maxmin(x, ⌊((x + y)/2)⌋)

(max2, min2):= maxmin(⌊((x + y)/2) + 1)⌋,y)

return (max(max1, max2), min(min1, min2))

**Analysis**

Let ***T(n)*** be the number of comparisons made by Max−Min(x,y), where the number of elements n=y−x+1

If ***T(n)*** represents the numbers, then the recurrence relation can be represented as

**T(n) = T(⌊n/2⌋)+T(⌈n/2⌉)+2 for n>2 for n=2**

**1 for n=2**

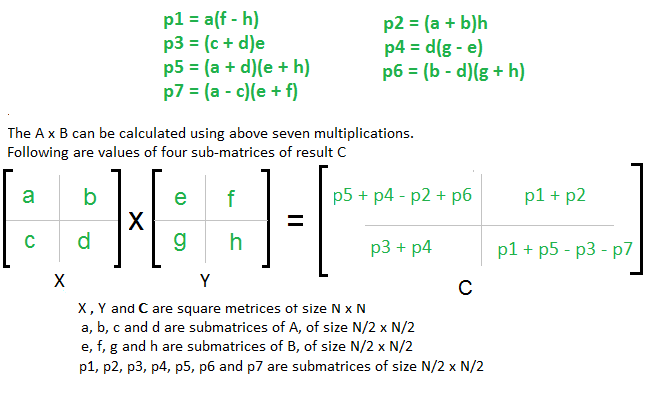
**0 for n=1**

Let us assume that ***n*** is in the form of power of **2**. Hence, **n = 2k** where **k** is height of the recursion tree.

So,

**T(n)=2.T(n/2)+2=2.(2.T(n/4)+2)+2.....=3n/2−2T(n)=2.T(n2)+2=2.(2.T(n4)+2)+2.....=3n2−2**

Compared to Naïve method, in divide and conquer approach, the number of comparisons is less. However, using the asymptotic notation both of the approaches are represented by **O(n)**.

simplest way to remember those :  
[](http://cdncontribute.geeksforgeeks.org/wp-content/uploads/stressen_formula_new_new1.png)

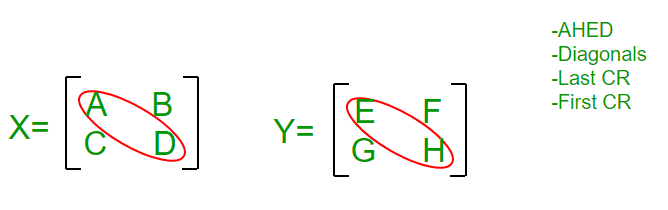
You just need to remember 4 Rules :

* AHED (Learn it as ‘Ahead’)
* Diagnol
* Last CR
* First CR

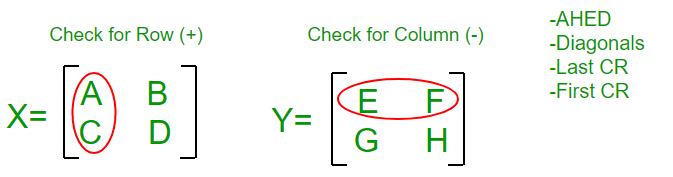
Also, consider X as (Row +) and Y as (Column -) matrix

Follow the Steps :

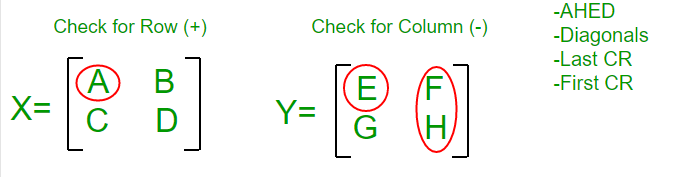
* Write P1 = A; P2 = H; P3 = E; P4 = D
* For P5 we will use Diagnol Rule i.e.  
  (Sum the Diagnol Elements Of Matrix X ) \* (Sum the Diagnol Elements Of Matrix Y ), we get  
  P5 = (A + D)\* (E + H)

[](https://cdncontribute.geeksforgeeks.org/wp-content/uploads/strasen.png)  
P1 = A  
P2= H  
P3= E  
P4= D  
P5= ( A + D ) \* ( E + H )

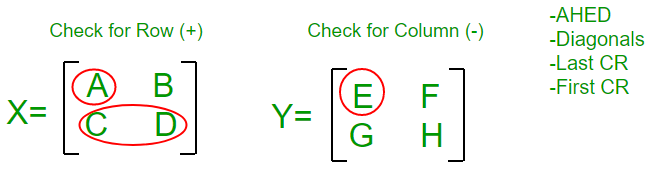
* For P6 we will use Last CR Rule i.e. Last Column of X and Last Row of Y and remember that Row+ and Column- so i.e. (B – D) \* (G + H), we get  
  P6 = (B – D) \* (G + H)
* For P7 we will use First CR Rule i.e. First Column of X and First Row of Y and remember that Row+ and Column- so i.e. (A – C) \* (E + F), we get  
  P6 = (A – C) \* (E + F)

[](https://cdncontribute.geeksforgeeks.org/wp-content/uploads/strasen-1.png)  
P1 = A  
P2= H  
P3= E  
P4= D  
P5= ( A + D ) \* ( E + H )  
P6= ( B – D ) \* ( G + H)  
P7= ( A – C ) \* ( E + F)

* Come Back to P1 : we have A there and it’s adjacent element in Y Matrix is E, since Y is Column Matrix so we select a column in Y such that E won’t come, we find F H Column, so multiply A with (F – H)  
  So, finally P1 = A \* (F – H)

[](https://cdncontribute.geeksforgeeks.org/wp-content/uploads/strasen-2.png)  
P1 = A \* ( F – H)  
P2= H  
P3= E  
P4= D  
P5= ( A + D ) \* ( E + H )  
P6= ( B – D ) \* ( G + H)  
P7= ( A – C ) \* ( E + F)

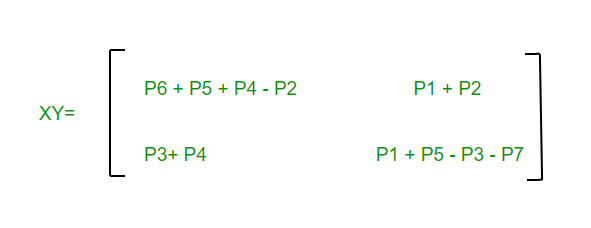
* Come Back to P2 : we have H there and it’s adjacent element in X Matrix is D, since X is Row Matrix so we select a Row in X such that D won’t come, we find A B Column, so multiply H with (A + B)  
  So, finally P2 = H \* (A + B)
* Come Back to P3 : we have E there and it’s adjacent element in X Matrix is A, since X is Row Matrix so we select a Row in X such that A won’t come, we find C D Column, so multiply E with (C + D)  
  So, finally P3 = E \* (C + D)

[](https://cdncontribute.geeksforgeeks.org/wp-content/uploads/strasen-3.png)  
P1= A \* ( F – H )  
P2= H \* ( A + B )  
P3= E \* ( C + D )  
P4= D  
P5= ( A + D ) \* ( E + H )  
P6= ( B – D ) \* ( G + H)  
P7= ( A – C ) \* ( E + F)

* Come Back to P4 : we have D there and it’s adjacent element in Y Matrix is H, since Y is Column Matrix so we select a column in Y such that H won’t come, we find G E Column, so multiply D with (G – E)  
  So, finally P4 = D \* (G – E)

We are done with P1 – P7 equations, so now we move to C1 – C4 equations in Final Matrix C :

* Remember Counting : Write P1 + P2 at C2
* Write P3 + P4 at its diagnol Position i.e. at C3
* Write P4 + P5 + P6 at 1st position and subtract P2 i.e. C1 = P4 + P5 + P6 – P2
* Write odd values at last Position with alternating – and + sign i.e. P1 P3 P5 P7 becomes  
  C4 = P1 – P3 + P5 – P7

[](https://cdncontribute.geeksforgeeks.org/wp-content/uploads/strasen-4.png)

1. The **Job sequencing problem** states as follows:

* There are ‘n’ jobs to be processed on a machine.
* Each job ‘i’ has a deadline di ≥ 0 and profit pi ≥ 0.
* Profit is earned if & only if the job is completed by its deadline.
* The job is completed if it is processed on a machine for unit time.
* Only one machine is available for processing jobs.
* Only one job is processed at a time on the machine.
* A feasible solution is a subset of jobs J such that each job is completed by its deadline.
* An optimal solution is a feasible solution with maximum profit value.

**Example:**

Let n = 4, (p1,p2,p3,p4) = (100,10,15,27), (d1,d2,d3,d4) = (2,1,2,1)

The feasible solution and their values are given below.

|  |  |  |  |
| --- | --- | --- | --- |
| **Sr No.** | **Feasible Solution** | **Processing Time** | **Value** |
| 1. | (1,2) | 2,1 | 110 |
| 2. | (1,3) | 1,3 or 3,1 | 115 |
| 3. | (1,4) | 4,1 | 127 |
| 4. | (2,3) | 2,3 | 25 |
| 5. | (3,4) | 4,3 | 42 |
| 6. | (1) | 1 | 100 |
| 7. | (2) | 2 | 10 |
| 8. | (3) | 3 | 15 |
| 9. | (4) | 4 | 27 |

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

In the example considered before, the non-increasing profit vector is

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **100** | **27** | **15** | **10** |  | **2** | **1** | **2** | **1** |
| p1 | p4 | p3 | p2 |  | d1 | d4 | d3 | d2 |

i. Now we start adding the job with J = 0 and Σ i ej Pi= 0.

ii. Job 1 is added to J as it has the largest profit and thusJ = {1} is a feasible one.

iii. Now job 4 is considered. The solution J = {1, 4} is a feasible one with processing sequence (4, 1).

iv. The next job, Job 3 is considered & it is discarded as J = {1, 3, 4} is not feasible.

v. Finally, job 2 is added into J and it is discarded as J = {1, 2, 4} is not feasible.

vi. Thus we have J = {1, 4} is optimal solution with value 127.’

[Note: If this question is asked for 10 marks then write algorithm also.]

**Algorithm:-**

Algorithm Job\_seq (d, 1, n)

{

d [0] = 0;

JS [1] = 1;

i = 1;

for j= 2 to n do;

{

k = i;

while ((d[JS[k] > d[j]]) and (d[JS[k]] != k)) do

k = k -1;

if ((d[JS[k] < d[j]]) and (d[j] > k)) then

{

For m= i to (k+1) step-1 do

JS [m+1] = JS [m];

JS [K+1] = j;

i = i+1;

}

}

Return i;

}

It is established that the computing time of job sequencing can be reduced from O(n2)O(n2) to O (n).