Divide And Conquer

Divide and conquer is probably the best known general algorithm design technique. It follows a top-down approach. The approach consists of dividing the problem into smaller subproblems hoping that the solutions of the subproblems are easier to find and then combining the partial solutions into the solution of the original problem.

divide and conquer method consists of following phases.

1. Breaking the problem into several sub-problems that are similar to the original problem but smaller in size.
2. Solve the sub-problem recursively (successively and independently), and then.
3. If necessary, the solutions obtained for the subproblems are combined to get a solution to the original problem.

```
1.  Algorithm DAndC (P)
    {
        if Small (P) then return S(P);
        else
        {
          divide P into smaller instances P1, P2, ... Pk,
                                          K ≥ 1;
          Apply DAndC to each of these subproblems;
          return Combine (DAndC(P1), DAndC(P2), ...,
                                  DAndC(Pk));
        }
    }
```

Algorithm: Control Abstraction for D. & C. Approach

* control abstraction mirrors the way an algorithm based on divide and conquer will look.

* By a control abstraction, the divide and conquer strategy is explained as :

→ DAndC is initially invoked as DAndC(p), where p is the problem to be solved.

→ Small(p) is a Boolean-valued function that determines whether the input size is small enough that the answer can be computed without splitting.

→ If this is so, the function S is invoked.

→ Otherwise the problem p is divided into smaller subproblems.

→ These sub problems $P_1, P_2, \cdots, P_k$ are solved by recursive applications of DAndC.

→ Combine is a function that determines the solution to p using the solutions to the k subproblems.

→ If the size of p is n and sizes of the k subproblems are $n_1, n_2, \cdots, n_k$, respectively, then the computing time of DAndC is described by the ~~recurrence~~ recurrence relation.

$$T(n) = \begin{cases} g(n) & n \text{ is small} \\ T(n_1) + T(n_2) + \cdots + T(n_k) + f(n) & \text{otherwise} \end{cases}$$

where $T(n)$ is the time for DAndC on any input of size n, and $g(n)$ is the time to compute the answer directly for small inputs. The function $f(n)$ is the time for dividing p and combining the solutions to subproblems.

The complexity of many divide-and-conquer algorithm is represented by the recurrences of the form.

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

where a and b are known constants. Here we assume that $T(1)$ is known and n is a power of b (i.e, $n=b^k$).

—o—

Example: consider the case in which $a=2$ and $b=2$. Let $T(1)=2$ and $f(n)=n$. we have

$$\begin{aligned} T(n) &= 2T(n/2) + n \\ &= 2[2T(n/4) + n/2] + n \\ &= 4T(n/4) + 2n \\ &= 4[2T(n/8) + n/4] + 2n \\ &= 8T(n/8) + 3n \\ &\vdots \end{aligned}$$

In general we see that

$$T(n) = 2^i T(n/2^i) + in, \quad \text{for any } \log_2 n \geq i \geq 1.$$

In particular that $T(n) = 2^{\log_2 n} T(n/2^{\log_2 n}) + n \log n$.

corresponding to the choice of $i = \log_2 n$.

Thus $T(n) = nT(1) + n\log_2 n$

$$\boxed{T(n) = n\log_2 n + 2n}$$

Binary Search :-

Binary search is an efficient algorithm for searching in a sorted array. Let a_i, $1 \le i \le n$, be an array of elements that are sorted in ascending order. Consider the problem of determining whether a given element x is present in the array. Divide-and-conquer can be used to solve this problem.

* Let small(p) be true if $n=1$. In this case, S(p) will take the value i if $x = a_i$; otherwise it will take the value 0. Then $g(1) = \theta(1)$.
* If p has more than one element, it can be divided into subproblems using middle element m. There are three possibilities.
  - Compare a search key x with the array's middle element A[mid]. If $x = A[mid]$, the algorithm stops.
  - If $x < A[mid]$, x has to be searched for only in the sublist $A[1], \ldots, A[mid-1]$.
  - If $x > A[mid]$, x has to be searched for only in the sublist $A[mid+1], \ldots, A[n]$.
* The same comparision is repeated recursively.
* Binary search is clearly based on a recursive idea, it can be easily implemented as a non recursive algorithm.
* The recursive algorithm for binary search is given as follows.

```
Algorithm BinSrch ( A, low, high, x)
{
    if (low > high) then
        return 0;
    else
    {
        mid = ⌊((low+high)/2)⌋
        if x = a[mid] then return mid;
        else if (x < a[mid]) then
            BinSrch (A, low, mid-1, x);
        else
            BinSrch (A, mid+1, high, x).
    }
}.
```

Algorithm : Algorithm for BinarySearch.

Time Complexity of BinSrch (A, low, high, x).

* For searching problems, the time complexity depends on number of comparisions.
* Assume that T(n) is the time complexity of BinSrch(A, low, high, x).
* For first if statement, the complexity is 1.
* For the else part, either else if is executed or else is executed. That means $n/2$ elements are searched. So, the complexity of this statement is $T(n/2)$.
* Therefore the TimeComplexity of BinSrch(A, low, high, x) is $T(n/2) + 1 = O(\log n)$.

—o—

Example:- Let us consider the elements.

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

place them in a[1:14], search 151 & -14.

Here low = 1, high = 14 (No of elements), x = 151 (searching element)

low > high // which is false

so

low	high	mid
1	14	7
8	14	11
12	14	13
14	14	14

Found.

① mid = $\frac{low+high}{2} = \frac{1+14}{2} = 7$

if x = a[mid]

151 = a[7] // but 151 != 54

else if x < a[mid]

151 < a[7] // 151 < 54 // which is // false

else

151 > 54

then low = mid+1 = 8.
high = 14

② mid = $(8+14)/2 = 11$

if 151 = a[11] // 151 != 125

else if 151 < a[11] // which is false

else 151 > a[11] // 151 > 125

then low = mid+1 = 12.
high = 14

③ mid = $(12+14)/2 = 13$

if 151 = a[13] // 151 != 142

else if 151 < a[13] // which is false

else 151 > 142

Then low = mid+1 = 14
high = 14

④ mid = (14+14)/2 = 14. ⟹ if 151 = a[14] // which is true

So 151 = 151

The searched element is in the 14th position.

Exercise: Search the elements -14 and 9.

Calculation of time complexity:

Assume that n is powers of 2. ie., $n = 2^k$, where k is a non-negative integer. And $T(1) = a$.

$$T(n) = T(n/2) + 1 \quad -1$$

$$T(n/2) = T(n/4) + 1 \quad -2$$

$$T(n/4) = T(n/8) + 1 \quad -3.$$

Substitute equations 2 in 1

$$T(n) = (T(n/4) + 1) + 1$$

$$= T(n/4) + 2 \quad -4.$$

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

Substitute equations 3 in 4

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

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

$$= T(n/2^3) + 3. \quad -5$$

$$\vdots$$

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

Since $2^k = n$, $k = \log_2 n$

$$T(n) = T(n/n) + \log_2 n$$

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

$$= a + \log_2 n$$

$$\boxed{T(n) = O(\log_2 n)}$$

Maxima and Minima problem

* The simplest problem to implement directly divide and conquer strategy is finding maximum and minimum items in a set of n elements.

* The conventional approach of maxima minima is given, below. The time complexity for finding maximum and minimum items is $2(n-1)$ for element comparisions in the best, average and worst cases.

```
Algorithm CMaxMin (a, n, max, min)
{
    max = min = a[1];
    for i = 2 to n
    {
        if (a[i] > max) then max = a[i];
        if (a[i] < min) then min = a[i];
    }
}
```

Algorithm: Conventional Maximum and minimum.

* In the above algorithm, even $a[i] > max$, we compare $a[i] < min$ also. The general improvement in conventional method is.

```
if (a[i] > max) then max = a[i];
else if (a[i] < min) then min = a[i];
```

* Now the best case occurs when the elements are in increasing order, In this case, never goes to else part. The time complexity is $n-1$.

* The worst case occurs when the elements are in decreasing Order. In this case, always goes to the else part. The Time complexity is $2(n-1)$.

* on the average, the time complexity is $3n/2 - 1$.

```
Algorithm MaxMin(i, j, max, min)
{
    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
        {
        // if P is not small, divide p into subproblems
            mid = ⌊(i+j)⌋/2;
        // split problem into two equal subproblems.
            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;
        }
}.
```

Algorithm: Divide and conquer strategy for maximum and minimum.

→ A divide and conquer strategy for maximum and minimum problem would be described as.

* P is a problem with n elements array.
* Let Small (P) be true where $n \leq 2$.

- In this case, the maximum and minimum are a[i] if n=1.
- If n=2, the problem can be solved by making one comparision.

* If the list has more than two elements, p has to be divided into smaller sub problems P_1 and P_2.
* Find maximum and minimum values for the subproblems P_1 and P_2.
* Max(P) is the larger of Max(P_1) and Max(P_2)
* Min(P) is the smaller of Min(P_1) and Min(P_2).
* If T(n) is the number of comparisions of this algorithm, then the recurrence relation is.

$T(n)=0$ for $n=1$

$T(n)=1$ for $n=2$

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

when $n = 2^k$ for some positive integer then

$$T(n) = 2T(n/2) + 2$$
$$= 2^2(2T(n/4)+2) + 2$$
$$= 4T(n/4) + 4 + 2$$
$$= 4(2T(n/8)+2) + 2$$
$$= 2^2T(n/2^2) + 2^2 + 2$$
$$= 2^3T(n/2^3) + 2^3 + 2^2 + 2$$
$$\vdots$$
$$= 2^{k-1}T(n/2^{k-1}) + 2^{k-1} + 2^{k-2} + \cdots + 2$$
$$= n/2\ T(2) + 2^{k-1} \quad \text{since } (x^{n+1}-1)/(x-1) = x^n + x^{n-1} + \cdots + 1$$
$$= n/2 + n - 1$$

$$\boxed{T(n) = 3n/2 - 1}$$

* Note that $3n/2 - 1$ is the best, average and worst-case number of comparisions.
* Compared to $2n-2$ comparisions for the straightforward method there is a saving of 25% on comparisions.

Example: Find the Maximum and minimum of the array

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

a[1]	a[2]	a[3]	a[4]	a[5]
22	13	-5	-8	15

a[6]	a[7]	a[8]	a[9]
60	17	31	47

a[1]	a[2]	a[3]
22	13	-5

a[4]	a[5]
-8	15

a[6]	a[7]
60	17

a[8]	a[9]
31	47

a[1]	a[2]
22	13

a[3]
-5

* The maximum and minimum of a[1] and a[2] is 22 & 13
* The maximum and minimum of a[3] and a[3] is -5
* The maximum and minimum of a[1], a[2] and a[3] is 22 & -5
* The maximum and minimum of a[4] & a[5] is 15 and -8
* The maximum and minimum of a[6] & a[7] is 60 and 17
* Thus the maximum and minimum of a[1], a[2], a[3], a[4] and a[5] is 22 & -8.
* The maximum and minimum of a[6] and a[7] is 60 & 17
* The maximum and minimum of a[8] and a[8] is 47 and 31
* Thus the maximum and minimum of a[6], a[7], a[8] and a[9] is 60 and 17
* Finally we get the maximum and minimum of the given array as 60 and -8

Merge Sort

Mergesort is based on the divide-and-conquer Paradigm. The mergesort algorithm can be described in the following three steps:

(1) Divide step: If given array A has zero or one element, return S; it is already sorted. otherwise divide A into two arrays, A_1 and A_2, each containing half of the elements of A.

- Given a sequence of n elements a[1]... a[n], the idea is to split them into two sets a[1],... a[n/2] and a[n/2]+1, ... a[n].

(2) Recursion step: Recursively sort array A_1 and A_2.

- Each set is individually sorted and the resulting sorted sequences are merged to produce a single sorted sequence of n elements.
- mergesort describes this process using recursion.

(3) Conquer step:- combine the elements back in A by merging the sorted arrays A_1 and A_2 into a sorted sequence.

- A function Merge is used to merge two sorted sets.

```
Algorithm : MergeSort(low, high)
// a[low:high] is a global array to be sorted
// Small(p) is true if there is only one element to sort
{
    if (low < high) then // if there are more than one element
    {
        // divide p into subproblems
            mid = ⌊(low+high)/2⌋;
        //Solve the sub-problems
            Mergesort (low, mid);
            Mergesort (mid+1, high);
```

```
        // combine the solutions
            Merge (low, mid, high);
        }
    }.
```

Algorithm : Mergesort

* Before executing Mergesort, the n elements should be placed in a[1:n]. Then Mergesort(1,n) causes the keys to be rearranged into nondecreasing order in a.

```
Algorithm Merge (low, mid, high)
// a [low : high] is a global array containing two sorted
// subsets in a[low : mid] and in a[mid+1 : high].
{
    h = low ; i = low ; j = mid+1;
    while ((h ≤ mid) and (j ≤ high)) do
    {
        if (a[h] ≤ a[j]) then
        {
            b[i] := a[h]; h := h+1;
        }
        else
        {
            b[i] := a[j]; j := j+1;
        }
        i := i+1;
    }
    if (h > mid) then
        for k := j to high do
        {
            b[i] := a[k]; i := i+1;
        }
        else
            for k := h to mid do
            {
                b[i] := a[k]; i := i+1;
            }
        for k := low to high do a[k] := b[k];
}.
```

Merging two sorted arrays

Example: Sort the following Numbers Using merge sort Algorithm 310 285 179 652 351 423 861 254 450 520.

* Algorithm begins by splitting a[] into 2 subarrays each of size, five a[1:5] and a[6:10].
* The elements in a[1:5] are then split into two subarrays of size three & a[1:3] and two a[4:5].
* The items in a[1:3] are split into subarrays of size two a[1:2] and one a[3:3].
* The two values in a[1:2] are then split into one element subarrays and now merging begins.

- Step1 : 310 | 285 | 179 | 652 , 351 | 423 861 254 450 520.
  Elements a[1] and a[2] are merged.

  step2 : 285 310 | 179 | 652 351 | 423 861 254 450 520.
  a[3] is merged with a[1:2].

  Step 3 : 179 285 310 | 652 351 | 423 861 254 450 520
  Elements a[4] and a[5] are merged.

  Step 4 : 179 285 310 | 351 652 | 423 861 254 450 520
  Elements a[1:3] and a[4:5] are merged.

  Step 5 : 179 285 310 351 652 | 423 861 254 450 520.
  Now the algorithm returns to the first invocation of the merge sort.

  step 6 : 179 285 310 351 652 | 423 | 861 | 254 | 450 520.
  Elements a[6] and a[7] are merged

  Step 7 : 179 285 310 351 652 | 423 861 | 254 | 450 520
  Elements a[6:7] and a[8] are merged

  step8. 179 285 310 351 652 | 254 423 861 | 450 520.

Elements a[6:8] and a[9:10].

Step 9: 179 285 310 351 652 | 254 423 450 520 861

we get two sorted subarrays and the final merge will give the sorted sequence, ie a[1:5] and a[6:10] are merged.

Step 10: (179 254 285 310 351 423 450 520 652 861)

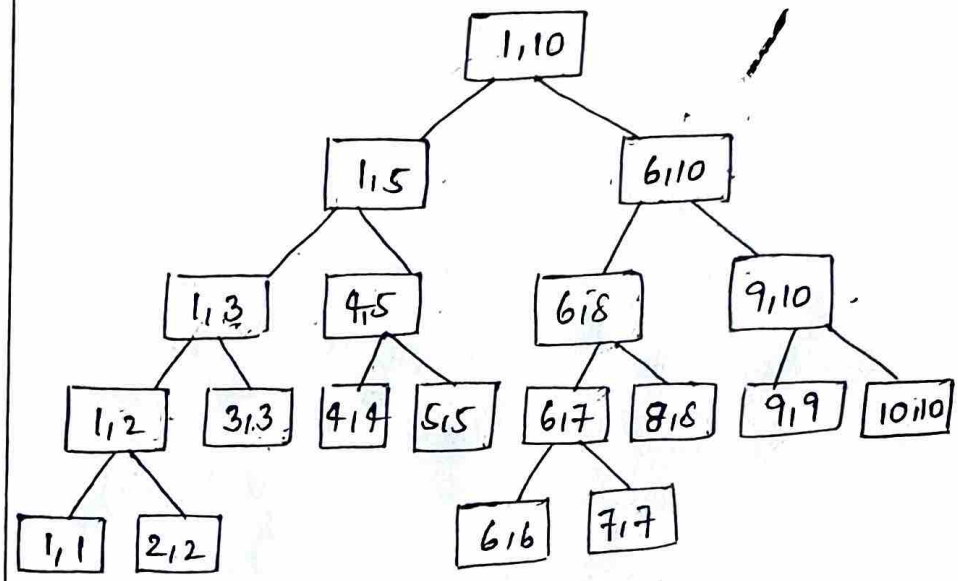


Fig: Tree calls of Mergesort(1,10)

* The above tree represents the sequence of recursive calls that are produced by mergesort algorithm when it applied to 10 elements.
* The splitting continues until sets containing a single elements are produced.

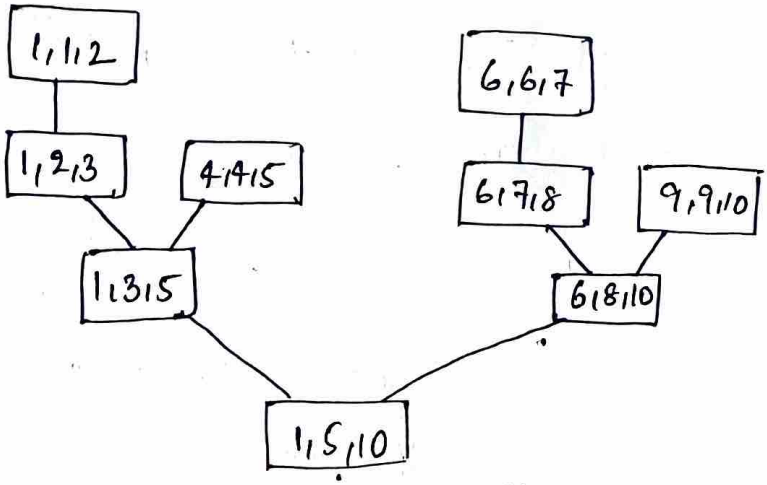


Fig: Tree of merge calls.

* The above figure representing the calls to produce merge by mergesort.

* The node containing 1, 2, and 3 represents the merging of a[1;2] and a[3].

Time Complexity of merge sort:

* Assume that T(n) is the time complexity of merge sort for n elements. merge sort has an average and worst-case performance of O(n log n). If the running time of merge sort for a list of length n is T(n).

* The time complexity of mergesort (low, mid) is $T(n/2)$, because mid = $n/2$.

* The time complexity of mergesort (mid+1, high) is $T(n/2)$.

* The time complexity of Merge (low, mid, high) is n.

Therefore, the Time Complexity of Merge sort is

$$2T(n/2) + cn = O(n \log n)$$

Calculation of Time complexity of mergesort:

Assume that n is power of 2. i.e, $n = 2^k$, $T(1) = a$.

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

$$= 2\left(2T(n/2^2) + cn/2\right) + cn$$

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

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

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

$$= 8T(n/8) + 2*cn/2 + 2cn.$$

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

$$= 2^3 T(n/2^3) + 3cn.$$

$$\vdots$$

$$= 2^k T(n/2^k) + kcn$$

$$T(n) = nT(1) + cn\log_2 n \qquad [\text{since } 2^k = n,\ k = \log_2 n]$$

$$T(n) = na + cn\log n$$

The time complexity of the mergesort is $T(n) = O(n\log_2 n)$

—o—

Quick Sort

Quick sort is used on the principle of divide and conquer. It is efficient compared to other sorting algorithms.

Quicksort works by partitioning a given array $A[m:n]$ into two non-empty subarrays $A[m..j]$ and $A[j+1, \cdots n]$ such that every key in $A[m..j]$ is less than or equal to every key in $A[j+1 \cdots n]$.

Then the two subarrays are sorted by recursive calls to quicksort. The exact position of the partition depends on the given array and index j is computed as a part of the partitioning procedure.

Mainly we have to check three conditions always.

* if $(a[i] <= k)$ then i++
* if $(a[j] <= k)$ then j--
* if $(i<j)$ then swap $a[i]$ and $a[j]$. otherwise
* swap pivot element with $a[j]$.

In this algorithm we are using 2 pointers i and j

* i always moves towards right and j moves towards left until we get partition.
* After the first partition the elements which are less than pivot element are considered to be ~~another~~ one sublist.
* And all the elements which are greater than pivot element are considered to be another sublist.
* Again we have to apply the quick sort procedure recursively to both the sublists.

```
Algorithm Partition (a, m, n)
{
    V := a[m]; i := m; j := p;
    repeat
    {
      repeat
          i := i+1
      until (a[i] ≥ V);
      repeat
          j := j-1;
      until (a[j] ≤ V);
      if (i < j) then Interchange (a, i, j);
    } until i ≥ j;
    a[m] := a[j]; a[j] := V; return j;
}
```

```
Algorithm Interchange (a, i, j)
// Exchange a[i] with a[j].
{
    p := a[i];
    a[i] := a[j]; a[j] := p;
}
```

```
Algorithm Quicksort (p, q)
// sorts the elements a[p], ..., a[q], which reside
// in the global array a[1:n] into ascending order;
// a[n+1] is considered to be defined and must be
//  >= all the elements in a[1:n].
{
```

```
if (p < q) then // If there are more than one element
{
  // divide P into two subproblems.
     j := Partition (a, p, q+1);
          // j is the position of partitioning element
  // Solve the subproblems.
     QuickSort (p, j-1);
     QuickSort (j+1, q);
  // There is no need for combining solutions.
}
}.
```

Algorithm: Sorting by partitioning.

Analysis of QuickSort

* The running time of quicksort depends on whether Partition is balanced or unbalanced, which in turn depends on which elements of an array to be sorted are used for partitioning.
* A very good partition splits an array up into two equal sized arrays.
* A bad partition, on other hand, splits an array up into two arrays of very different sizes.
* The worst partition puts only one element in one array and all the other elements in the other array.
* If the partitioning is balanced, the Quicksort runs asymptotically as fast as mergesort.

* on the otherhand if partitioning is unbalanced, the quicksort runs asymptotically as slow as Insertion sort.

* Let $T(n)$ be the expected time required by quick sort to sort a sequence of n elements.
Clearly $T(0) = T(1) = b$, for some constant b.

* The running time of the partition procedure is $O(n)$ where $n = n - m + 1$ which is the number of keys in the array.

Worst case time complexity of Quick sort:

The worst case occurs when in each recursion step an unbalanced partitioning is produced, namely that one part consists of only one element and the other part consists of the rest of the elements.

Then the recursion depth is $n-1$

$$T(n) = T(n-1) + cn$$
$$= T(n-2) + c(n-1) + cn$$
$$= T(n-3) + c(n-2) + c(n-1) + cn$$
$$\cdots$$
$$= T(1) + c(1+2+3+4+\cdots+n)$$
$$= b + c\left(n(n+1)/2\right)$$
$$= b + c\left((n^2+n)/2\right)$$

$$\boxed{T(n) = O(n^2)}$$

Best case complexity of Quick Sort

The best-case behavior of the quick sort algorithm occurs when in each recursion step the partitioning produces two parts of equal length ie., $n/2$.

* In order to sort n elements, in this case the running time is in $O(n\log(n))$.
* This is because the recursion depth is $\log(n)$ and on each level there are n elements to be treated.
* The recurrence relation is then.

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

Assume that n is power of 2 ie $n^k = 2^k$, $T(1) = a$.

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

$$= 2\left[2T(n/2^2) + cn/2\right] + cn$$

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

$$= 4\left[2T(n/4/2) + cn/4\right] + 2cn$$

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

$$= 2^3 T(n/2^3) + 3cn$$

$$\vdots$$

Do the kth substitution

$$= 2^k T(n/2^k) + kcn$$

$$T(n) = nT(1) + kcn \quad [\text{since } 2^k = n,\ k = \log_2 n]$$

$$= an + \log_2 n\, cn$$

$$= an + cn\log_2 n$$

$$\boxed{T(n) = O(n\log_2 n)}$$

Average case time complexity of Quicksort

Example:- Sort these numbers 26, 5, 37, 1, 61, 11, 59, 15, 48, 19 using quick sort.

Let p = 1, q = 10

If (1 < 10)

{

j = : partition(a, 1, 11);

now the partition procedure will be invoked.

In that

v = a[i] = 26 ; i ← 1 ; j ← 11

i++;

i = 2

a[i] > v

5 > 26 false.

i = i+1

i = 3

a[i] > v

37 > 26 true.

j--;

j = 10.

a[j] < v

19 < 26 true.

so swap a[i] and a[j] // a[3] and a[10].

ie 26 5 19 1 61 11 59 15 48 37.

i = 3.

a[i] > v

19 > 26 false i++

i = 4

a[i] > v

1 > 26 false i++

i = 5

a[i] > v

61 > 26 true

j = 10

a[j] < V

37 < 26 false j--

j = 9

a[j] < V

48 < 26 false j--

j = 8

a[j] < V ⇒ 15 < 26 true so swap a[i] & a[j]

i.e a[5] & a[8]

26 5 19 1 15 11 59 61 48 37

i = 5 j = 8

a[i] > V ⇒ 15 > 26 false. i++

i = 6

a[i] > V ⇒ 11 > 26 false. i++

i = 7

a[i] > V ⇒ 59 > 26 true.

a[j] < V ⇒ 61 < 26 false j--

j = 7

a[j] < V ⇒ 59 < 26 false j--

j = 6

a[j] < V ⇒ 11 < 26 true

but i > j is true.

So swap a[j] and pivot element.

[11 5 19 1 15] 26 [59 61 48 37]

The pivot element is placed in it's exact position. after one partition, and the element present at the left sub table are less than the partitioned element 26 and the right subtable elements are greater than = 26.

Repeat this algorithm on left side elements of 26 and right side elements of 26 untill we obtain a Partition of each element.

The remaining process is.

[11 5 19 1 15] 26 [59 61 48 37]

Let $V = a[i] = 11$; $i \leftarrow 1$; $j \leftarrow 6$.

$i++ \Rightarrow i = 2$

$a[i] \geq V \Rightarrow 5 > 11$ false $\Rightarrow i++$

$i = 3$.

$a[i] > V \Rightarrow 19 > 11$ true.

$j-- ; \Rightarrow j = 5$

$a[j] \leq V \Rightarrow 15 \leq 11$ false

$j-- \Rightarrow j = 4$.

$a[j] \leq V \Rightarrow 1 \leq 26$. true.

So Swap $a[i]$ and $a[j]$ // $a[3]$ and $a[4]$.

ie [11 5 1 19 15] 26 [59 61 48 37]

$i = 3$.

$a[i] \geq V \Rightarrow 1 \geq 15$ false $\Rightarrow i++$.

$i = 4$

$a[i] \geq V \Rightarrow 19 \geq 15$ true.

$j-- \Rightarrow j = 3$.

$a[j] \leq V \Rightarrow 1 \leq 15$ true

but the condition $i \geq j$ is be satisfied.

So partition the sublist by interchanging pivot with $a[j]$.

Now the list is

[1 5] 11 [19 15] 26 [59 61 48 37].

~~There is no~~ Then we have to apply partition algorithm for both the sublists. when we apply partition on [1 5] list there is no interchange will be taken, because the elements are placed in it's actual positions.

and then when we apply partition on [19 15] sublist ~~without a partition of pivot & a[j]~~ within a single iteration the elements will be ~~partion~~ partitioned.

so the list is

[1] [5] [11] [15] [19] 26 [59 61 48 37].

The remaining process is.

[1] [5] [11] [15] [19] 26 [48 37] 59 [61]

[1] [5] [11] [15] [19] [26] [37] [48] [59] [61].

So it is the final sorted array.

—o—

Strassen's Matrix Multiplication.

* Let A and B be two $n \times n$ matrices. The product matrix $C = AB$ is also an $n \times n$ matrix whose i,j^{th} elements are formed by taking the elements in the i^{th} row of A and j^{th} column of B and multiplying them to get

$$C(i,j) = \sum_{1 \le k \le n} A(i,k)\, B(k,j)$$

for all i and j between 1 and n.

To compute $C(i,j)$ using this formula we need n multiplications. The divide and conquer strategy suggest another way to compute the product of two $n \times n$ matrices.

for simplicity we assume that n is power of 2, that is, there exists a non negative integer k such that $n = 2^k$.

* Imagine that A and B are partitioned into four Square Sub matrices, each Sub matrix having dimensions $n/2 \times n/2$.

* then the product AB can be computed by using the above formula for the product of 2×2 matrices.

If AB is.

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} \quad \text{---(1)}$$

Then.

then

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

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

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

$$C_{22} = A_{21}B_{12} + A_{22}B_{22} \quad \text{—— (2)}$$

* If $n=2$, then formulas (1) and (2) are computed using a multiplication operation for the elements of A and B.

* for $n>2$ the elements of 'c' can be computed using matrix multiplication and addition operations applied to matrices of size $n/2 \times n/2$.

* To compute A*B using formulas (2) we need to perform eight multiplications of $n/2 \times n/2$ matrices and four additions of $n/2 \times n/2$ matrices. Since two $n/2 \times n/2$ matrices can be added in time cn^2 for some constant c,

* The overall computing time $T(n)$ of the resulting divide and conquer algorithm is given by the recurrence

$$T(n) = \begin{cases} b & n \leq 2 \\ 8T(n/2) + cn^2 & n > 2 \end{cases}$$

where b and c are constants.

Calculation of time complexity of Divide and Conquer matrix multiplication.

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

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

$$T(n/4) = 8T(n/8) + cn^2/16$$

$$T(n) = 8\left[8T(n/4) + cn^2/4\right] + cn^2$$

$$= 8^2 T(n/4) + 3cn^2 \Rightarrow 8^2 T(n/4) + (2^2-1)cn^2$$

$$= 8^2\left[8T(n/8) + cn^2/16\right] + 3cn^2$$

$$= 8^3 T(n/8) + cn^2(8/4)^2 + 3cn^2 \Rightarrow$$

$$= 8^3 T(n/2^3) + 7cn^2$$

$$= 8^3 T(n/2^3) + (2^3-1)cn^2$$

$$\vdots$$

$$= 8^k T(n/2^k) + (2^k-1)cn^2$$

$$= 2^{3k} T(n/2^k) + (2^k-1)cn^2$$

Let $n = 2^k$, then $k = \log_2 n$ and

$$= n^3 b + (n-1)cn^2$$

$$= O(n^3)$$

Therefore the time complexity is $O(n^3)$.

* Strassen's has discovered a way to compute the C_{ij}'s of (2) only 7 multiplications and 18 additions or subtractions.

* His method involves first computing the seven $n/2 \times n/2$ matrices P, Q, R, S, T, U and V.

Then C_{ij}'s are computed using the formulas.

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

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

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

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

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

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

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

$$C_{11} = P + S - T + V$$

$$C_{12} = R + T$$

$$C_{21} = Q + S$$

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

The resulting recurrence relation for $T(n)$ is

$$T(n) = \begin{cases} b & n \leq 2 \\ 7T(n/2) + an^2 & n \geq 2 \end{cases}$$

where a and b are constants.

$$T(n) = 7T(n/2) + cn^2$$

$$T(n/2) = 7T(n/4) + cn^2/4$$

$$T(n/4) = 7T(n/8) + cn^2/16$$

$$T(n) = 7[7T(n/4) + cn^2/4] + cn^2$$

$$= 7^2[7T(n/8) + cn^2/16] + 7cn^2/4 + cn^2$$

$$= 7^3[7^3 T(n/8^3) + cn^2(7/4)^2 + (7/4)cn^2 + cn^2$$

$$\vdots$$

$$T(n) = 7^k T(n/2^k) + cn^2 [(7/4)^{k-1} + (\frac{7}{4})^{k-2} + \cdots + 7/4 + 1]$$

$$= 7^k T(n/2^k) + cn^2 (7/4)^k$$

Let $n = 2^k$, then $k = \log_2 n$

$$T(n) = 7^{\log_2 n} T(1) + cn^2 (7/4)^{\log_2 n}$$

Since $a \log_b c = c \log_b a$

$$T(n) = n^{\log_2 7} T(1) + cn^2 * n^{\log_2 7/4}$$

$$T(n) = n^{\log_2 7} T(1) + cn^{\log_2 7 + \log_2 4 - \log_2 4}$$

$$T(n) = n^{\log_2 7} + c n^{\log_2 7}$$

$$T(n) = (1+c) n^{\log_2 7}$$

Therefore, the Time complexity is $O(n^{\log_2 7})$

i.e $O(n^{2.81})$

— o —

UNIT-IV :

Greedy Method : General method, applications - job sequencing with deadlines, 0/1 knapsack problem, Minimum cost spanning trees, Single source shortest path problem.

Greedy Method

* Greedy algorithms are simple and straightforward design technique. It can be applied to a wide variety of problems.
* most of these problems have n inputs and requires us to obtain a set that satisfies some constraints.
* Any subset that satisfies these constraints is called a feasible solution.
* our objective is to find a feasible solution that either maximizes or minimizes a given objective function
* For every problem there is a way to determine feasible solution. A feasible solution may or may not be an optimal solution. But An optimal solution is a feasible solution.
* Greedy method suggests that one can device an algorithm that works in stages, considering one input at time.
* At each stage, a decision is made, regarding whether a particular input is in an optimal solution.
* Considering inputs in an order determined by some selection Procedure.

* If the inclusion of the next input into the partially constructed optimal solution will result in an infeasible solution, then this input is not added to the partial solution. otherwise it is added.

* The selection of inputs, is based on some optimization measure. This measure is called objective function.

The control abstraction for greedy method is.

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

Greedy method control abstraction for the subset paradigm.

* function Select, selects an input from a[] and removes it
* The selected input's value is assigned to x
* Feasible is a boolean values function that determines whether x can be added to solution vector
* The function Union combines x with the solution and updates the objective function.

* In the greedy method we make decisions by considering the inputs in some order. Each decision is made using an optimization criterion. This version of Greedy method is called ordering paradigm.

=

KNAPSACK PROBLEM

* knapsack problem is one application of greedy method. This problem can be solved by using Greedy mechanism.
* The requirements for the knapsack problem is; There are given 'n' objects and a knapsack of capacity 'm',
* The object 'i' has a profit p_i and there is associated weight w_i.
* If a fraction x_i, $0 < x_i \leq 1$ of object 'i' is placed into the knapsack then a profit of $p_i x_i$ is earned.
* The objective is to fill the knapsack that maximizes the total profit earned.
* Since the knapsack capacity is m, so we require to hold the total weight of all choosen objects to be atmost 'm'.

The problem is stated as

Maximize $\sum_{i=1}^{n} p_i x_i$

Subject to $\sum_{i=1}^{n} w_i x_i \leq m$ where.

$0 \leq x_i \leq 1$

and $1 \leq i \leq n$.

→ profits and weights are positive numbers.

Example :-

Consider the following instance of the knapsack problem $n=3$, $m=20$, $(p_1,p_2,p_3)=(25,24,15)$ and $(w_1,w_2,w_3)=(18,15,10)$.

1. First, we try to fill the knapsack by selecting the objects in some order.

x_1	x_2	x_3	$\sum w_i x_i$	$\sum p_i x_i$
$1/2$	$1/3$	$1/4$	$18 \times 1/2 + 15 \times 1/3 + 10 \times 1/4 = 16.5$	$25 \times 1/2 + 24 \times 1/3 + 15 \times 1/4 = 24.25$

(2.) Next, consider, Select the objects according to maximum profit to minimum profit.

x_1	x_2	x_3	$\sum w_i x_i$	$\sum p_i x_i$
1	$2/15$	0	$18 \times 1 + 15 \times 2/15 + 10 \times 0 = 20$	$25 \times 1 + 24 \times 2/15 + 15 \times 0 = 28.2$

Select the object with the maximum profit first i.e, $(p=25)$. So $x_1=1$ and profit earned is 25. now only 2 units of space is left in the knapsack., Select the object with next largest profit $(p=24)$. and it's weight is 15, so complete placement of second object is not possible, only the fraction of object is placed into the knapsack. So $x_2=2/15$

(3) Considering the objects according to minimum weight to maximum weight w_i.

x_1	x_2	x_3	$\sum w_i x_i$	$\sum p_i x_i$
0	2/3	1	$18\times0 + 15\times2/3 + 10\times1 = 20$	$25\times0 + 24\times2/3 + 15\times1 = 31$

(4). Selecting the objects in the order, from maximum P_i/w_i to minimum P_i/w_i.

P_1/w_1	P_2/w_2	P_3/w_3
25/18	24/15	15/10
1.4	1.6	1.5

Sort the objects in order of the decreasing order of the ratio P_i/w_i.

x_1	x_2	x_3	$\sum w_i x_i$	$\sum p_i x_i$
0	1	1/2	$18\times0 + 15\times1 + 10\times1/2 = 20$	$25\times0 + 24\times1 + 15\times1/2 = 31.5$

* The solution vector $(0, 1, 1/2)$ is the optimal solution of the knapsack problem. which yields maximum profit.
* So, the greedy approach to solve the knapsack problem is arranging input in the order of P_i/w_i ratio.
*

The Algorithm for Greedy knapsack is.

Algorithm Greedy knapsack (m, n)

```
// P[1:n] and w[1:n] contain the profits and weights
// respectively of the n objects ordered such that
// P[i]/w[i] ≥ P[i+1]/w[i+1]. m is the knapsack size
// and x[1:n] is the solution vector
{
    for i:=1 to n do x[i]:=0.0 //Initialize x
        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];
}.
```

Analysis:-

* If the items are already sorted into decreasing order of $P[i]/w[i]$. Then the ~~while~~ loop for loop takes a time in $O(n)$. Therefore the total time including the sort is in $O(n \log n)$.

* But if we discarded the time to initially sort the objects, the algorithm requires only $O(n)$ time

Job Sequencing with Deadlines

* In the job sequencing with deadlines problem, we are given a set of n jobs. Each job 'i' associated with an integer deadline. $d_i \geq 0$ and a profit $P_i \geq 0$.
* For any job i, the profit P_i is earned iff the job is completed by its deadline.
* To complete the 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 a 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.
* Since the problem solves the identification of a subset, it fits to the Subset paradigm.

=

Example:- Apply job seq 100 19 38 27 52

Let $n=4$, $(P_1, P_2, P_3, P_4) = (100, 10, 15, 27)$ and $(d_1, d_2, d_3, d_4) = (2, 1, 2, 1)$ generate feasible solutions. 2 1 2 1 3

Job seq Selection Activity

	Feasible solution	Processing sequence	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)	3,2	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 non increasing order of profits subject to the constraint that the resulting job sequence J is a feasible solution.

* In this example considered before, the non-increasing profit vector is $(P_1, P_4, P_3, P_2) = (100, 27, 15, 10)$; $(d_1, d_4, d_3, d_2) = (2, 1, 2, 1)$.

$J = \{1\}$ is a feasible solution.

$J = \{1, 4\}$ is a feasible solution

$J = \{1,3,4\}$ is not feasible.

$J = \{1,2,4\}$ is not feasible

Therefore $J = \{1,4\}$ is optimal.

Algorithm

The algorithm constructs an optimal set J of jobs that can be processed by their deadlines.

```
Algorithm Greedy-Job(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 ∪ {i} can be completed
            by their deadlines)
        then J = J ∪ {i};
    }
}.
```

* In the above example, the optimal solution is a set $J = (1,4)$, only jobs 1 and 4 are processed and the value is 127.
* These jobs must be processed in the order job 4 followed by job 1.
* The processing of job 4 begins at time zero and that

Job 1 is completed at time 2.

* In the above algorithm, to represent the set J, how we have to carryout the task that how the next job is added to the set J.
* This task can be analysed and devised by the following procedure.

(1) Sort the jobs in J ordered by their deadlines.

(2) The array d[1:n] is used to store the deadlines of the order of their p-values.

(3) The set of jobs J[1:K] such that J[r], $1 \leq r \leq K$ are the jobs in 'j' and

$$d[J[1]] \leq d[j[2]] \leq \cdots \leq d[J[K]).$$

(4) Test whether $J \cup \{i\}$ is feasible, we have just to insert i into J preserving the deadline ordering and then verify that $d[J[r]] \leq r$,

$$1 \leq r \leq K+1.$$

Time Analysis of job sequencing Algorithm

* Let 'n' be the number of jobs and 's' be the number of jobs included in the solution
* The loop is iterated (n-1) times.
* Each iteration takes O(k) where k is the number of existing jobs
* The time needed by the algorithm is O(sn)
* $s \leq n$, so the worst case time is $O(n^2)$

gorithm JS(d, j, n)

// d[i] ≥ 1, 1 ≤ i ≤ n are the dead lines for n ≥ 1.

// The jobs are ordered such that p[1] ≥ p[2] ≥ ... ≥ p[n]

// J[i] is the ith job in the optimal solution 1 ≤ i ≤ K.

// 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 non increasing order of P[i]. Find position for i and check feasibility of 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;

}

Tracing:-

Consider example written before after sorting $P_1, P_2, \ldots P_4$.

P (100, 27, 15, 10) (P1) (P4) (P3) (P2)

d (2, 1, 1, 2, 1) d1 d4 d3 d4

n = 5, $(P_1, P_2, P_3, P_4, P_5) = (20, 15, 10, 5, 1)$ and

$(d_1, d_2, d_3, d_4, d_5) = (2, 2, 1, 3, 3)$

Algorithm JS(d, i, 3)

{

d[0] = J[0] = 0;

J[1] := 1;

```
r=1;
for i=2 to 5 do
{
    r=1
    while (2>2 & (2≠1)) → false
    if ((2≤2)&(2>1))
    {
        for q=1 to (r+1)(i.e) 2
            step+1 do J[2]=J[1];
        for 2 to 2 step 1 do J[3]=J[2];
        J[2]=2;
        K=2.
    }
    return 2;
for i=3 to 5 do
{
    r=2;
    while ((2>1)&(2≠2)) X
    if ((2≤1)&(1>2)) X
}
for i=4 to 5 do
{
    r=2;
    while(2>3) X
    if ((2≤3 & 3>2)
    {
        for q=2 step 1 do J[3]=J[2];
        for q=3 step+1 do J[4]=J[3];
        J[3]=4 (i);
        K=K+1=3;
    }
    return 3;
for i=5 to 5 do
{
    r=3;
    while (3>3 & 2≠2) X
    if (3≤3 & 3>3) X
```

	0	1	2	3	4
d	0	2	1		

	0	1	2	3	4	5
d	0	2	2	1	3	3

	0	1	2	3	4
J	0	1	2	1	

	0	1	2	3	4
J	0	1	2	4	X

consider jobs (1,2,4) as optimal solution costs max val of 40.

Minimum Cost Spanning Trees.

* A given graph can have many spanning Trees, from these many spanning trees, we have to select a optimal one.

Minimum Spanning Tree:-

* A spanning tree for a connected graph is a
  - tree whose vertex set is same as the vertex set of the given graph.
  - and whose edge set is a subset of the edge set of the given graph.

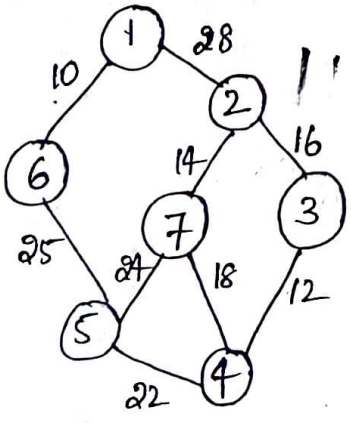
* Minimal cost spanning tree is a ~~spanning~~ connected undirected graph G in which each edge is labeled with a cost
* Minimal cost spanning tree is a spanning tree for which the sum of the edge labels is as small as possible.
* To find a minimal cost spanning tree, we use algorithms called prim's algorithm and kruskal's algorithm.

=

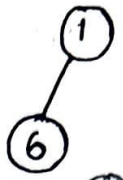
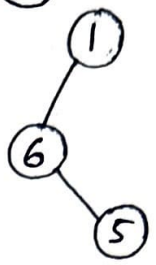
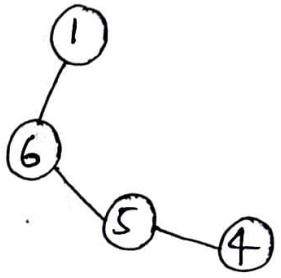
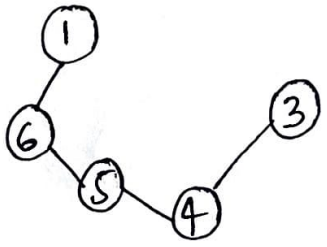
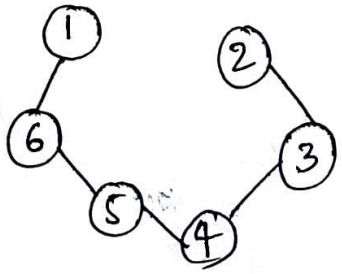
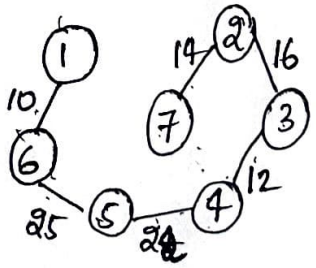
Prim's algorithm ✓

* A Greedy method to obtain a minimum-cost spanning tree builds this tree edge by edge.
* The next edge to include is choosen according to some optimization criterion.
* The simplest such criterion is to choose an edge that results in a minimum increase in the sum of the costs of the edges so far included.
* prim's algorithm has the property that, 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 A, with the property that $A \cup \{(u, v)\}$ is also a tree.
* This process is repeated until a spanning tree is formed.

Example:- consider the given graph and construct minimum spanning tree.



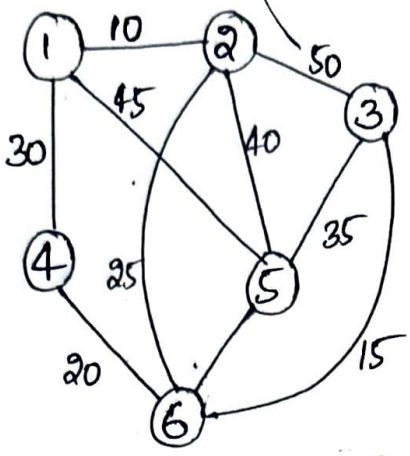
The spanning tree obtained for this graph is.

edge	cost	spanning tree.
(1,6)	10	
(6,5)	25	
(5,4)	22	
(4,3)	12.	
(3,2)	16	
(2,7)	14	

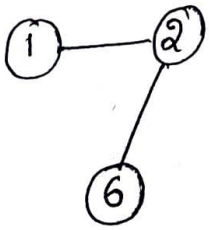
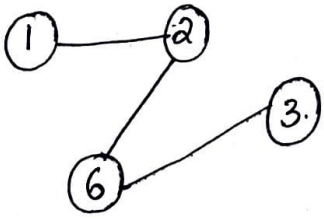
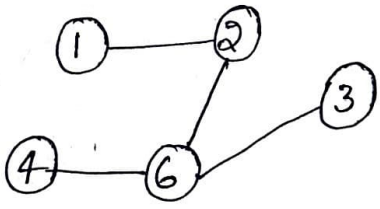
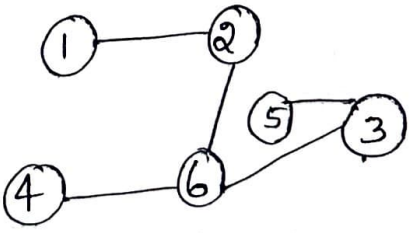
Step by step procedure for constructing minimum spanning tree.

* cost for this spanning tree is sum of the edges included in the tree. so cost is 99 ✓

Example 2:-

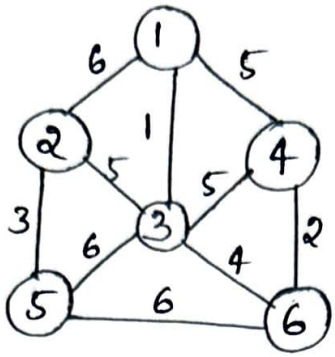


A Graph with six vertices.

Edge	cost	Spanning tree
(1,2)	10	
(2,6)	25	
(3,6)	15	
(6,4)	20	
(3,5)	35	

total cost of this minimum spanning tree is 105

Example 3:-



Step 1: $u = \{1\}$

$V-u = \{2,3,4,5,6\}$.

V-u	u	cost
2	1	6
3	1	1 (minimum) add vertex 3 to u.
4	1	5
5	1	∞
6	1	∞.

So $u = \{1,3\}$

$V-u = \{2,4,5,6\}$

V-u	u	cost
2	3	5
4	3	5
5	3	6
6	3	4 (minimum), so add vertex 6 to u.

now $u = \{1,3,6\}$

$V-u = \{2,4,5\}$.

V-u	u	cost
2	3	3
4	6	2 (minimum), so add vertex 4 to u
5	3	6

Now $u = \{1, 3, 6, 4\}$

$V - u = \{2, 5\}$

V-u	u	cost
2	3	5 (minimum) so add 2 to u.
5	6	6

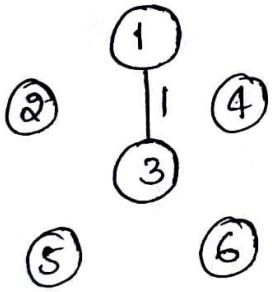
Now $u = \{1, 3, 6, 4, 2\}$

$V - u = \{5\}$.

V-u	u	cost
5	2	3. (minimum) so add 5 to u.
5	3	6.
5	6	6

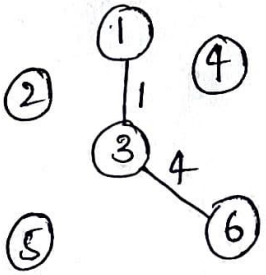
finally $u = \{1, 3, 6, 4, 2, 5\}$

Iteration 1



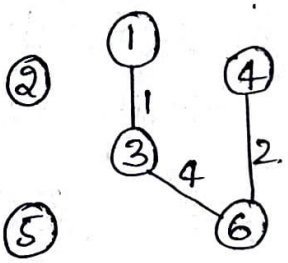
$V = \{1\}$

Iteration 2



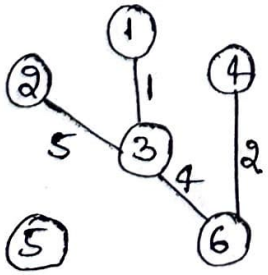
$u = \{1, 3\}$

Iteration 3



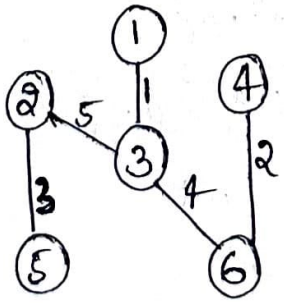
$u = \{1, 3, 6\}$.

Iteration 4



$u = \{1, 3, 6, 4\}$

Iteration 5



$u = \{1, 3, 6, 4, 2\}$.

Finally the total cost = 15

```
Algorithm prim(E, cost, n, t)
// E is the set of edges in G. cost[1:n, 1:n] is the
// cost adjacency matrix of an 'n' vertex graph.
// cost[i,j] is either a positive real number or ∞ if
// no edge (i,j) exists. A minimum spanning tree is
// computed and stored as a set of edges in the array
// t[1:n-1, 1:2]. (t[i,1], t[i,2]) is an edge in
// the minimum-cost spanning tree
{
    Let (k,l) be an edge of minimum cost in E.
    mincost := cost[k,l];
    t[1,1] := k ; t[1,2] := l;
Line12: for i:=1 to n do // Initialize near
            if (cost[i,l] < cost[i,k]) then
                near[i] := l;
            else
                near[i] := k;
        near[k] := near[l] := 0;
Line16: for i:=2 to n-1 do
        {
            // Find n-2 additional edges for t
Line18:     Let j be an index such that near[j] ≠ 0
Line19:     and cost[j, near[j]] is minimum;
            t[i,1] := j; t[i,2] := near[j];
            mincost := mincost + cost[j, near[j]];
            near[j] := 0;
Line23:     for k:=1 to n do
                if ((near[k] ≠ 0) and (cost[k, near[k]]
                    > cost[k, j])).
```

```
        then
            near[k] := j;
    }
    return mincost;
}.
```

* The 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, ~~and the cost of~~ and the cost[i,j] is minimum among all edges
* we determine near[j] = 0 for all vertices j that are already in the tree.
* The next edge to include is defined by the vertex j such that near[j] ≠ 0 and cost[j, near[j]] is minimum.
* The time required by algorithm prim is $O(n^2)$. where n is number of vertices in the Graph.

* The for loop of line 12 takes $O(n)$ time.
* Lines 18, & 19 and the for loop of line 23 require $O(n)$ time
* So, Each iteration of the for loop of line 16 takes $O(n)$ time.
* The total time for the for loop of line 16 is $\therefore O(n^2)$

Hence prim's runs in $O(n^2)$ time

Kruskal's algorithm ✓

Kruskal's algorithm for computing the minimum spanning tree is directly based on the generic MST algorithm.

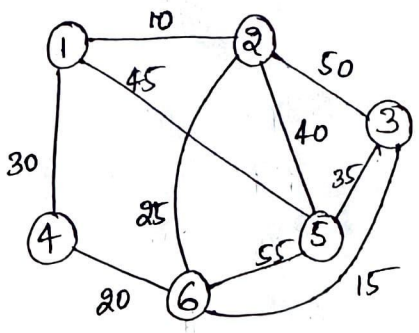
* Edges of the graph are considered in nondecreasing order of the cost.
* If an edge (u, v) connects two different trees, then (u, v) is added to the set of edges of the minimum spanning tree. and two trees connected by an edge (u, v), are merged into a single tree, on the other hand, if edge (u, v) connects two vertices in the same tree, then edge (u, v) is discarded.

Step 1: Sort all edges into nondecreasing order of costs.

Step 2: Add the next smallest weight edge to the forest if it will not cause a cycle.

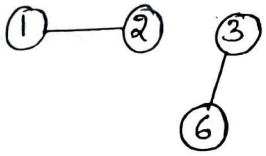
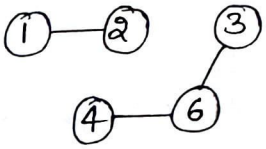
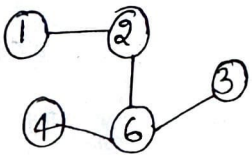
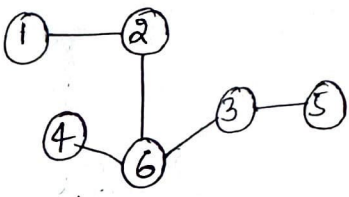
Step 3: Stop if $n-1$ edges. Otherwise go to step 2.

Example:- consider the graph given and construct minimum spanning tree using Kruskal's algorithm.



Arrange all the edges in the decreasing order of their costs.

cost	10	15	20	25	30	35	40	45	50	55
Edge.	(1,2)	(3,6)	(4,6)	(2,6)	(1,4)	(3,5)	(2,5)	(1,5)	(2,3)	(5,6)

Edge	Cost	Spanning tree
(1,2)	10	
(3,6)	15	
(4,6)	20	
(2,6)	25	
(1,4)	30	reject, addition of (1,4) causes a cycle.
(3,5)	35	

The vertices 3 and 5 are in different sets, so the edge is combined.

Total cost for the spanning tree is 105

```
Algorithm kruskal (E, cost, n, t)
// E is the set of edges in G. G has n vertices.
// cost[u,v] is the cost of edge (u,v). t is
// the set of edges in the minimum spanning tree.
// The final cost is returned.
{
  Construct a heap out of the edge cost using
  heapify;
  for i:=1 to n do parent[i] := -1;
  //Each vertex is in a different set
     i:=0; mincost := 0.0;
  while ((i<= n-1) and (heap not empty)) do
  {
     Delete a minimum cost edge (u,v) from
     the heap and reheapify using adjust;
     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 mincost;
}
```

Single Source Shortest path problem
(Dijkstra's Algorithm)

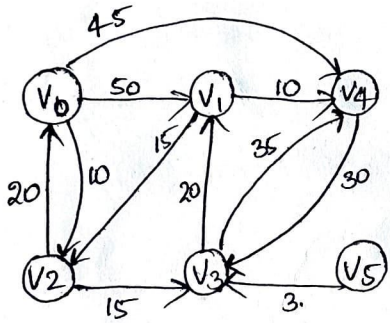
* In the single source, all destinations, shortest path Problem, we must find a shortest path from a given Source vertex to each of the vertices (called destinations) in the graph to which there is a path.

for Example:- A ~~mot~~ motorist coishing to drive from a city A to B, & he have to know about the following 2 questions.

(1) Is there a path from A to B.?

(2) If there is morethan one path from A to B, which is the shortest path.?

Ex:



(a)

Path	Length
(1) V_0V_2	10
(2) $V_0V_2V_3$	25
(3) $V_0V_2V_3V_1$	45
(4) V_0V_4	45

(b)

Fig: Graph and shortest paths from Vertex V_0 to all destinations

* To formulate a greedy based algorithm to generate the shortest paths, we use multistage solution to the problem and also of an optimization measure.

* As an optimization measure, the sum of the lengths of all paths so far generated can be considered and each individual path must be of minimum length.

* If we have already constructed i shortest paths, then using this optimization measure, the next path to be constructed.

* The Greedy way to generate the shortest paths

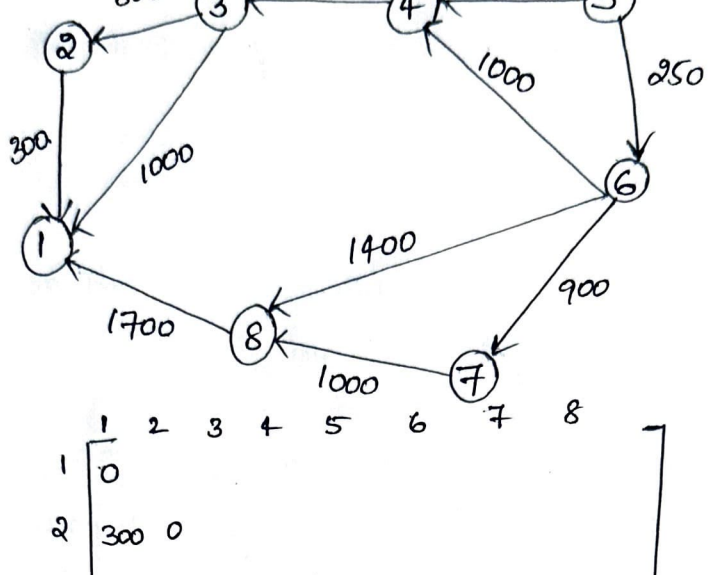
(1) First a shortest path to the nearest vertex is generated.

(2) Then a shortest path to the second nearest vertex is generated, and so on.

* Let's call the node we are starting with an initial node.
* Let a distance of node 'x' be the distance from the initial node to node 'x'
* and shortest path algorithm will assign some initial distance values and will try to improve them step-by-step.

* → Assign to every node a distance value. set it to '0' for initial node and ∞ to all other nodes.
  → mark all nodes as unvisited, set initial node as current node.
  → Then, consider neighbours of the current node, and calculate their distances (from initial node)
  → This distances are less than previously recorded distance, overwrite the distance
  → When we are completed the process of calculating the distances of neighbours of current node, then current node is said to Visited
  → A visited node will not be checked again
  → Next current node is choosen from the unvisited vertices which have the minimum distance. and continue these steps for n-2

Example:-



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

cost Adjacency matrix.

Iteration S	vertex selected	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)
Intial	----	∞	∞	∞	∞	0	∞	∞	∞
1 {5}	6	∞	∞	∞	1500	0	250	∞	∞
2. {5,6}	7	∞	∞	∞	1250	0	250	1150	1650
3. {5,6,7}	4	∞	∞	∞	1250	0	250	1150	1650
4. {5,6,7,4}	8.	∞	∞	2450	1250	0	250	1150	1650
5. {5,6,7,4,8}	3.	3350	∞	2450	1250	0	250	1150	1650
6. {5,6,7,4,8,3}	2	3350	3250	2450	1250	0	250	1150	1650
7. {5,6,7,4,8,3,2}									

Shortest paths

```
Algorithm shortest paths(v, cost, dist, n)
// dist[j], 1≤j≤n, is set to the length of the
// shortest paths from vertex v to vertex j in a
// graph G with n vertices. dist[v] is set to zero.
// G is represented by it's cost adjacency matrix
// cost[1:n, 1:n].
{
   for i:=1 to n do
   {
      //Intialize S.
      S[i]:= false; dist[i]:= cost[v,i];
   }
   S[v]:= true; dist[v]:= 0.0; // put v in S.
   for num:= 2 to n-1 do
   {
      // Determine n-1 paths from v,
   Choose u among those vertices not in S
   Such that dist[u] is minimum.
      S[u]:= true; // put u in S.
   for(each w adjacent to u with S[w]=false)
                                        do
      //update distances
      if ( dist[w] > dist[v] + cost[v,w]) then
         dist[w]:= dist[u] + cost[v,w];
   }
}.
```

* The time taken by the algorithm on a graph with n vertices is $O(n^2)$
* In the above algorithm the time for the loop for (i:= 1 to n) takes $O(n)$ times

* The for loop [for (num := 2 to n-1] is executed n-2 times. Each execution of this loop requires O(n) times, at times choosing the next vertex and

* ~~So the total time is difficult for this~~ again a for loop for updating distances, total requires a time for this for loop is $O(n^2)$

=

Algorithm Analysis. V = 5

for i := 1 to n.

S[i] := false. dist[i] = cost[v, i]

S[1] := false dist[1] = cost[5, 1] = ∞

S[2] = false dist[2] = cost[5, 2] = ∞

S[3] = false dist[3] := cost[5, 3] = ∞

S[4] = false dist[4] := cost[5, 4] = 1500

S[5] = false. dist[5] := cost[5, 5] = 0

S[6] = false dist[6] := cost[5, 6] = 250

S[7] = false dist[7] := cost[5, 7] = ∞

S[8] = false dist[8] := cost[5, 8] = ∞.

S[5] := true ; dist[5] = 0.0;

for num := 2 to n-1 do.

{

num = 2.

Choose u. from among those vertices not in S, such that dist[u] is minimum.

S[u] := true; dis[6] is minimum.

S[6] := true;

for (each w adjacent to u with S[w] = false) do

w = 4 if (dist[w] > dist[v] + cost[v,w])) then

dist[4] > dist[6] + cost[6,4]

1500 > 250 + 1000 = 1250.

dist[4] := dist[6] + cost[u,w];

= 1250.

w = 7 if (dist[7] > dist[6] + cost[6,7])

∞ > 250 + 900 = 1150

dist[7] = 1150.

w = 8 if (dist[8] > dist[6] + cost[6,8])

∞ > 250 + 1400

∞ > 1650

dist[8] = 1650.

for num = 3.

v = 7, S[7] = true, w = 8.

if (dist[8] > dist[7] + cost[7,8])

1650 > 1150 + 1000

1650 > 2150 false

so 1650 remains same.

and so on this process is iterated for num = n-1 ✓