MODULE 2:

Divide and Conquer Algorithm | Introduction

DIVIDE AND CONQUER General Method : In divide and conquer method, a given problem is,

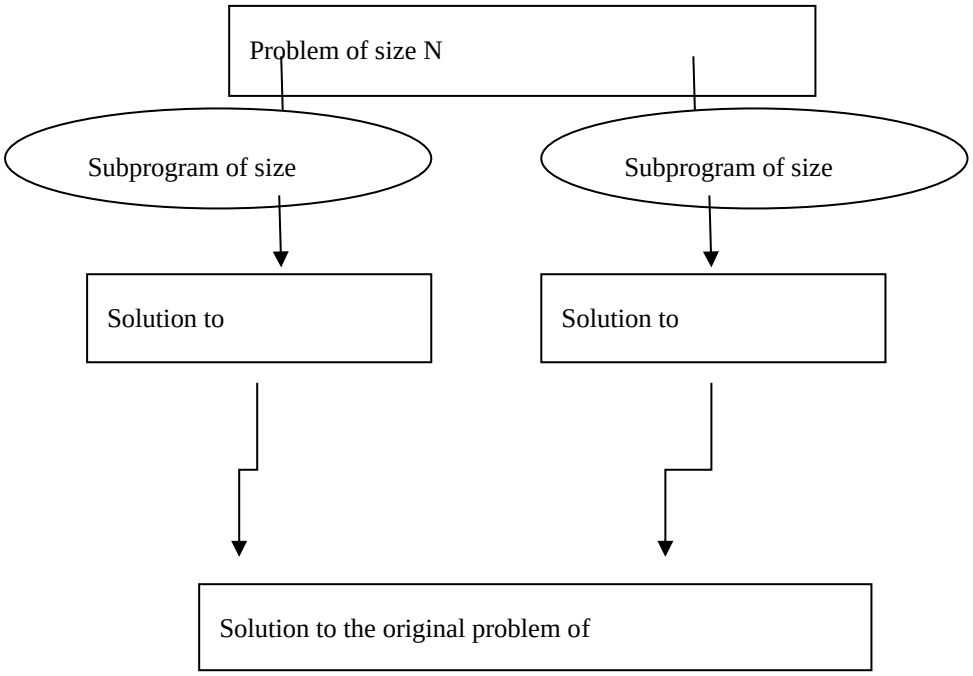
i) Divided into smaller subproblems.

ii) ii) These subproblems are solved independently.

iii) iii) Combining all the solutions of subproblems into a solution of the whole. If the subproblems are large enough then divide and conquer is reapplied.

iv) The generated subproblems are usually of some type as the original problem. Hence recurssive algorithms are used in divide and conquer strategy.

Hence recurssive algorithms are used in divide and conquer strategy.



Following are some standard algorithms that are Divide and Conquer algorithms.

1) Binary Search is a searching algorithm. In each step, the algorithm compares the input element x with the value of the middle element in array. If the values match, return the index of middle. Otherwise, if x is less than the middle element, then the algorithm recurs for left side of middle element, else recurs for right side of middle element.

2) Quicksort is a sorting algorithm. The algorithm picks a pivot element, rearranges the array elements in such a way that all elements smaller than the picked pivot element move to left side of pivot, and all greater elements move to right side. Finally, the algorithm recursively sorts the subarrays on left and right of pivot element.

3) Merge Sort is also a sorting algorithm. The algorithm divides the array in two halves, recursively sorts them and finally merges the two sorted halves.

Control abstraction for divide and conquer

- **P**:problem to be solved
- **Small (P)** : Boolean valued function that determines wether 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 problrm P is divided into small sub problems.
- These subproblems P1,P2,..Pk are solved recursive application of DAndC
- **Combine** is a function that determine the solution to P using the solutions to the k subproblems.

```
Algorithm DAndC(P)
{
 if small(P) then return S(P)
else{ divide P into smaller instances P1,P2,P3…Pk;  K>=1
apply DAndC to each of these subprograms; // means DAndC(P1), DAndC(P2)…..DAndC(Pk)
 return combine(DAndC(P1), DAndC(P2)…..DAndC(Pk));
 } }
```

Then computing time of DAndC is described by the recurrence relation

```
T(n)= g(n)     n small                    T(n)::time for DAndC on any input size n
                    F(n) : time for divide P and combining the solution to sub problems
T(n1)+T(n2)+...+T(nk)+f(n)   otherwise
```

```
T(n) = T(1) if n=1                                  a,b:constantsThis is called the general
                                                    divide and-conquer recurrence
aT(n/b)+f(n) if n>1
```

The time spent on executing the problem using DAndC is smaller than other method. This technique is ideally suited for parallel computation. This approach provides an efficient algorithm in computer science.

BINARY SEARCH

Binary search or Half-interval search algorithm:

1. This algorithm finds the position of a specified input value (the search "key") within an array sorted by key value.

2. In each step, the algorithm compares the search key value with the key value of the middle element of the array.

3. If the keys match, then a matching element has been found and its index, or position, is returned.

4. Otherwise, if the search key is less than the middle element's key, then the algorithm repeats its action on the sub-array to the left of the middle element or, if the search key is greater, then the algorithm repeats on sub array to the right of the middle element.

5. If the search element is less than the minimum position element or greater than the maximum position element then this algorithm returns not found.

Recursive binary search

```
1.Algorithm Binsrch(a,i,l,x)
2.//Give an array a[i:l] of elements in nondecreasing
3.//order,1<=i<=l,determine whether x is present,and
4.//if so,return j such that x=a[j];else return 0.
5.{
6. if(l=i) then // if Small(P)
7.{
8.if (x=a[i]) then return I;
9. else return 0;
10.}
```

```
13. mid =(i+l)/2;
14. if (x= a[mid]) then return mid;
15.   else if (x< a[mid]) then
16.   return BinSrch(a,I,mid-1,x);
 17.Else return BinSrch(a,mid+1,l,x);
18.}}
```

Worst case:O(log n) or θ(log n)

Average case:O(log n) or θ(log n)

Best case:O(1) or θ(1)

Unsuccessful search θ(log n):- for all cases.

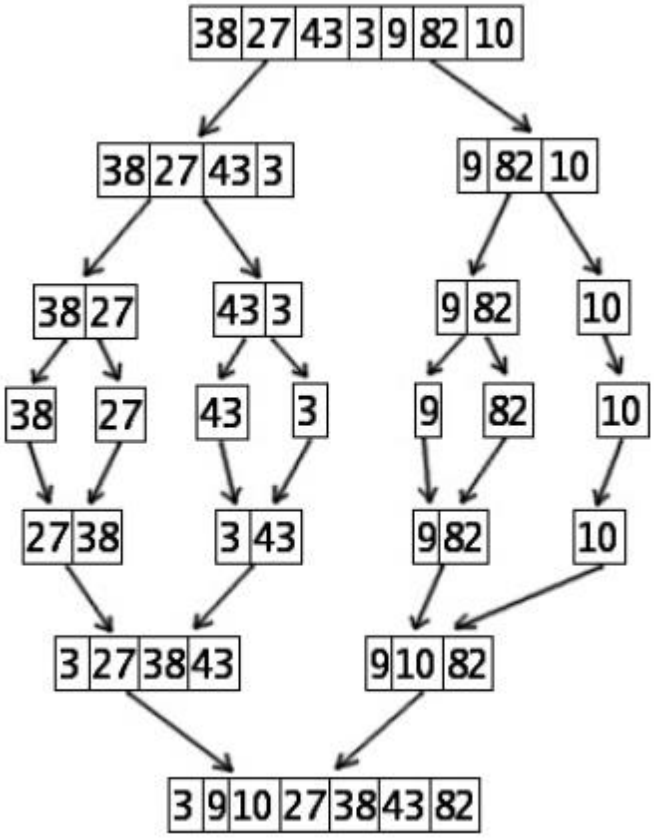
Iterative binary search

```
1. Algorithm BinSearch(a,n,x)
2. //Give an array a[i:n] of elements in nondecreasingorder,n>=0,determine whether x is
   present and if so, return j such that x=a[j];else return 0.
3. {
4. low=l;high=n;
5. while (low<=high) do
6. {
7. Mid=(low+high)/2;
8. If(x<a[mid] then high=mid-1;
```

Merge Sort:

1. The merge sort splits the list to be sorted into two equal halves, and places them in separate arrays.
2. This sorting method is an example of the DIVIDE-AND-CONQUER paradigm i.e. it breaks the data into two halves and then sorts the two half data sets recursively, and finally merges them to obtain the complete sorted list.
3. The merge sort is a comparison sort and has an algorithmic complexity of $O(n \log n)$. Elementary implementations of the merge sort make use of two arrays - one for each half of the data set.

The following image depicts the complete procedure of merge sort.



Advantages of Merge Sort:

1. Marginally faster than the heap sort for larger sets

2. Merge Sort always does lesser number of comparisons than Quick Sort. Worst case for merge sort does about 39% less comparisons against quick sort's average case.

3. Merge sort is often the best choice for sorting a linked list because the slow random-access performance of a linked list makes some other algorithms (such as quick sort) perform poorly, and others (such as heap sort) completely impossible.

Algorithm for Merge sort:

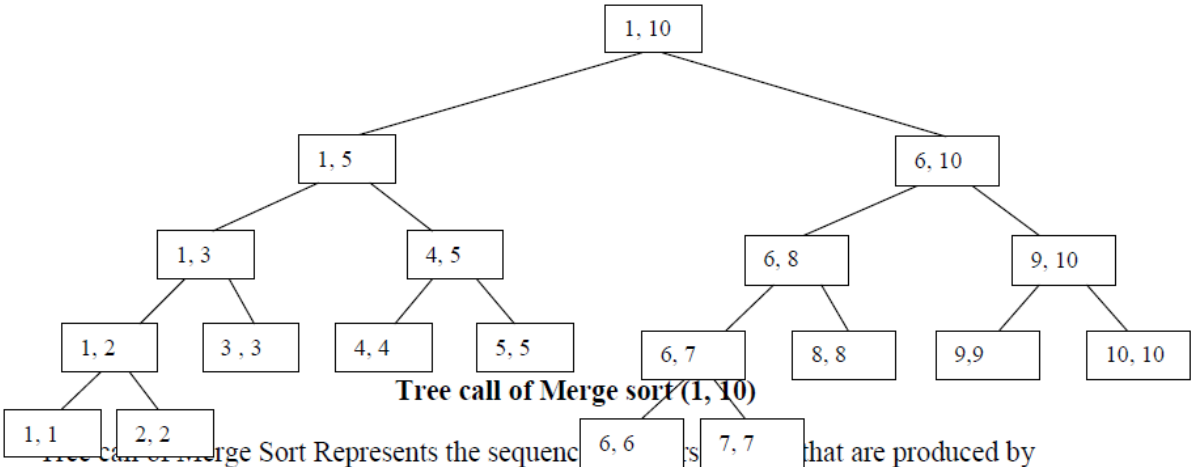
```
Algorithm mergesort(low, high)
//a[low:high]is a global array to be sorted
//Small(P) is true if there is only one element to sort.In this
//case the list is already sorted
{
if(low<high) then
{
mid=(low+high)/2;
//Solve the sub-problems
mergesort(low,mid);
mergesort(mid+1,high);
// Combine the solution
Merge(low,mid,high);
}
}
```

// Dividing Problem into Sub-problems and this "mid" is for finding where to split the set.

If more than 1 element

```
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].The goal is to
merge these two sets intoa single set residing in
a[low:high].b[] is an auxiliary global array.
h=low;
i=low;
j=mid+1;
while(h<=mid&&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
{
```

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



Tree call of Merge sort (1, 10)

Tree call of Merge Sort Represents the sequenc rs that are produced by

Computing Time for Merge sort:

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

T(n)= a if n=1;
2T(n/2)+cn if n>1

Here c, a Constants.

If n is power of 2, n=2k

Time complexity of merge sort is : By representing it by in the form of Asymptotic notation O is

T(n)=O(nlog n)

FINDING THE MAXIMUM AND MINIMUM

Maximum and Minimum:

1. Let us consider simple problem that can be solved by the divide-and conquer technique.
2. The problem is to find the maximum and minimum value in a set of 'n' elements.
3. By comparing numbers of elements, the time complexity of this algorithm can be analyzed.
4. Hence, the time is determined mainly by the total cost of the element comparison.

Explanation:

a. Straight MaxMin requires 2(n-1) element comparisons in the best, average & worst cases.

b. By realizing the comparison of a [i]max is false, improvement in a algorithm can be done.

c. Hence we can replace the contents of the for loop by, If (a [i]> Max) then Max = a [i]; Else if (a [i]< 2(n-1)

d. On the average a[i] is > max half the time, and so, the avg. no. of comparison is 3n/2-1.

A Divide and Conquer Algorithm for this problem would proceed as follows:

a. Let P = (n, a [i],……,a [j]) denote an arbitrary instance of the problem.

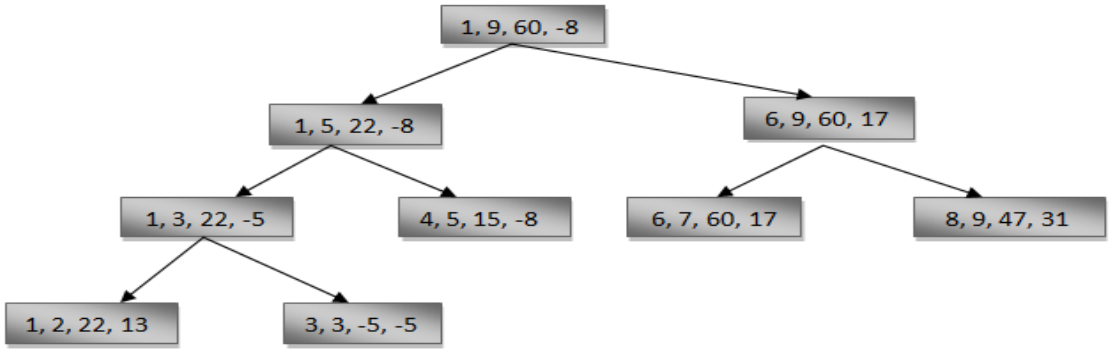
b. Here 'n' is the no. of elements in the list (a [i],….,a[j]) and we are interested in finding the maximum and minimum of the list.

c. If the list has more than 2 elements, P has to be divided into smaller instances.

d. For example, we might divide 'P' into the 2 instances, P1=([n/2],a[1],……..a[n/2]) & P2= (n-[n/2], a[[n/2]+1],….., a[n]) After having divided 'P' into 2 smaller sub problems, we can solve them by recursively invoking the same divide-and-conquer algorithm.

```
1.  Algoritham MaxMin(i, j, max, min)
2.  // 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],respectivily
3.  {
4.  if (i=j) then max := min := a[i]; //Small(P)
5.  else if (i=j-1) then // Another case of Small(P)
6.  {
            if (a[i] < a[j]) then max := a[j]; min := a[i];
            else
            max := a[i]; min := a[j];
7.  }
8.  else
9.  {
10. // if P is not small, divide P into sub-problems.
11. // Find where to split the set.
12. mid := ( i + j )/2;
13. // Solve the sub-problems.
14. MaxMin( i, mid, max, min );
15. MaxMin( mid+1, j, max1, min1 );
    // Combine the solutions.
    if (max < max1) then max := max1;
    if (min > min1) then min := min1;
```

new call is made. On the array a[ ] above, the following tree is produced.



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).

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

$$T(n) = \begin{cases} 0 & n=1 \\ 1 & n=2 \\ T(n/2) + T(n/2) + 2 & n>2 \end{cases}$$

When n is a power of two, $n = 2^k$

-for some positive integer k, then

T(n) = 2T(n/2) + 2…..

…..

. **= 3n/2 – 2 = O(n)**

Note that 3n/2 – 2 is the best, average, wo

two.

QUICK SORT

Quicksort is a divide-and-conquer sorting algorithm in which division is dynamically carried out (as opposed to static division in Mergesort).

The three steps of Quicksort are as follows: Divide: Rearrange the elements and split the array into two subarrays and an element in between such that so that each element in the left subarray is less than or equal the middle element and each element in the right subarray is greater than the middle element.

Conquer: Recursively sort the two subarrays.

Combine: None.

The Algorithm Quicksort(A, n) 1:

Quicksort0 (A, 1, n)

Quicksort0 (A, p, r)

1: if p ≥ r then return

2: q = Partition(A, p, r)

3: Quicksort0 (A, p, q − 1)

4: Quicksort0 (A, q + 1, r)

- Quick Sort is an algorithm based on the DIVIDE-AND-CONQUER paradigm that selects a pivot element and reorders the given list in such a way that all elements smaller to it are on one side
- Those bigger than it are on the other. Then the sub lists are recursively sorted until the list gets completely sorted. The time complexity of this algorithm is O (n log n).

 Auxiliary space used in the average case for implementing recursive function calls is O (log n) and hence proves to be a bit space costly, especially when it comes to large data sets.

 Its worst case has a time complexity of O (n^2) which can prove very fatal for large data sets. Competitive sorting algorithms

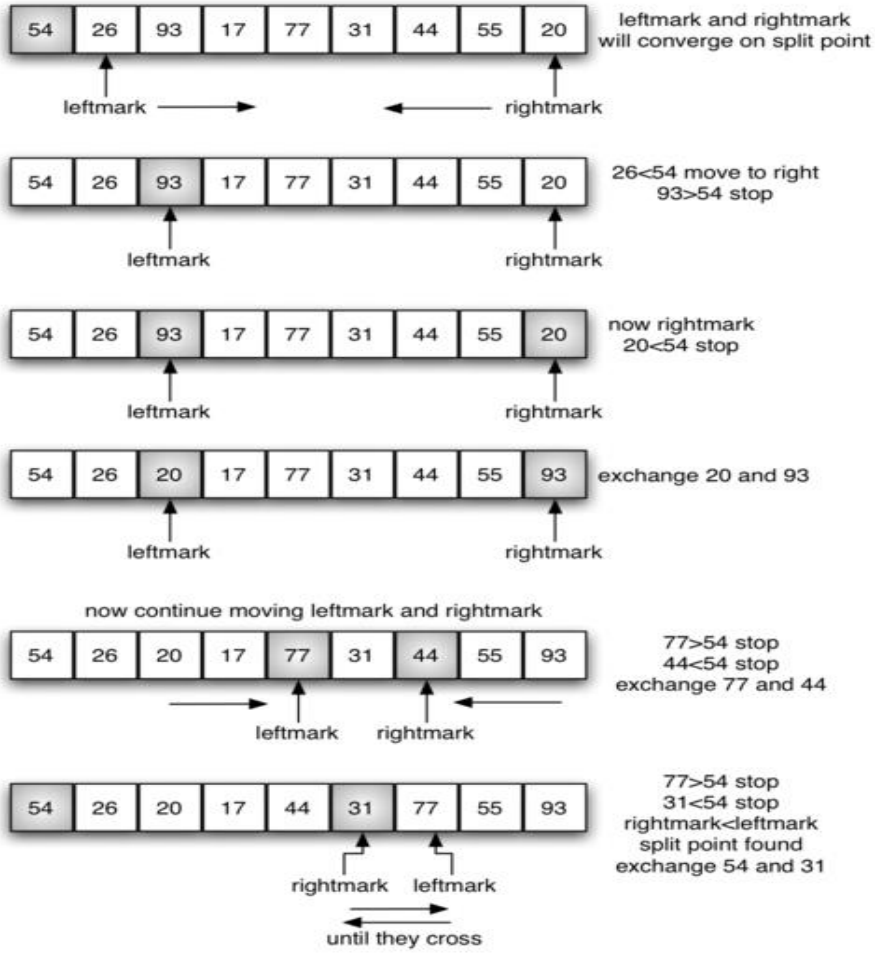
- A quick sort first selects a value, which is called the **pivot value**.
- we will simply use the first item in the list.
- The role of the pivot value is to assist with splitting the list.
- The actual position where the pivot value belongs in the final sorted list, commonly called the **split point**, will be used to divide the list for subsequent calls to the quick sort.

The subroutine Partition

- Given a subarray A[p .. r] such that $p \leq r - 1$, this subroutine rearranges the input subarray into two subarrays, A[p .. q − 1] and A[q + 1 .. r], so that
- each element in A[p .. q − 1] is less than or equal to A[q] and
- each element in A[q + 1 .. r] is greater than or equal to A[q]
- Then the subroutine outputs the value of q. Use the initial value of A[r] as the "pivot," in the sense that the keys are compared against it. Scan the keys A[p .. r − 1] from left to right and flush to the left all the keys that are greater than or equal to the pivot

Example:

Partitioning begins by locating two position markers—let's call them leftmark and rightmark—at the beginning and end of the remaining items in the list
The goal of the partition process is to move items that are on the wrong side with respect to the pivot value while also converging on the split point



Condition for quick sort

- We begin by incrementing leftmark until we locate a value that is greater than the pivot value.
- We then decrement rightmark until we find a value that is less than the pivot value.
- At this point we have discovered two items that are out of place with respect to the eventual split point. For our example, this occurs at 93 and 20. Now we can exchange these two items and then repeat the process again.
- At the point where rightmark becomes less than leftmark, we stop.
- The position of rightmark is now the split point.
- The pivot value can be exchanged with the contents of the split point and the pivot value is now in place
- In addition, all the items to the left of the split point are less than the pivot value, and all the items to the right of the split point are greater than the pivot value. The list can now be divided at the split point and the quick sort can be invoked recursively on the two halves.

Algorithm of QUICK sort

1.Sorting by partitioning

```
1. Algorithm QuickSort(p,q)
2. //sort 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]
3. {
4. If(p<q) then              //if there are more than one element
5. {
6. //divide P into two subproblems
7. J=Partition(a,p,q+1);          //j is the position of the partitioning element
8. //Solve the subproblems
9. QuickSort(p,j-1);
10. Quicksort(j+1,q)
11. //There is no need for combining solutions
12. }
```

2.Partition the array a[m:p-1] about a[m]

```
1. Algorithm Partition(a,m,p)
2. //Within a[m],a[m+1],…..a[p-] the elements are rearranged in such a manner that if initially
   t=a[m],then after completion a[q]=t for some q between m and p-1,a[k],+t for m<=k<q,and
   a[k]>=t //for q<k<p.q is returned.Set a[p]=∞
3. { V=a[m]; i=m; j=p
4. Repeat
5. { Repeat
6. i=i+1;
7. Until(a[i]>=v);
8. Repeat
9. j=j-1
10. until(a[j]<=v)
11. if(i<j) then Interchange(a,i,j);
12. }until (i>=j)
13. a[m]=a[j] ;      a[j]= v;          return j;          }
1.Algortithm Intechange(a,i ,j)
2.//Exchange a[i] with a[j]
3.{p=a[i]
5.a[i]= a[j];
6.a[j]=p;
7.}
```

Time complexity of quick sort

Worst Case: θ (n^2). The worst case occurs when the partition process always picks greatest or smallest element as pivot. If we consider above partition strategy where last element is always picked as pivot, the worst case would occur when the array is already sorted in increasing or decreasing order. Following is recurrence for worst case.

Best Case:θ(nlogn). The best case occurs when the partition process always picks the middle element as pivot.

Average Case:O(nlogn): To do average case analysis, we need to consider all possible permutation of array and calculate time taken by every permutation which doesn't look easy.

Comparison between Merge and Quick Sort:

- Both follows Divide and Conquer rule.
- Statistically both merge sort and quick sort have the same average case time i.e. O(n log n).
- Merge Sort Requires additional memory. The pros of merge sort are: it is a stable sort, and there is no worst case (means average case and worst case time complexity is same).
- Quick sort is often implemented in place thus saving the performance and memory by not creating extra storage space.
- But in Quick sort, the performance falls on already sorted/almost sorted list if the pivot is not randomized. Thus why the worst case time is O(n^2).

SELECTION SORT

- This type of sorting is called **Selection Sort** as it works by repeatedly sorting elements.
- It works as follows: first find the smallest in the array and exchange it with the element in the first position, then find the second smallest element and exchange it with the element in the second position, and continue in this way until the entire array is sorted.
- The worst case occurs if the array is already sorted in a descending order and we want to sort them in an ascending order.
- For each ***i*** from ***1*** to ***n - 1***, there is one exchange and ***n - i*** comparisons, so there is a total of
- ***n - 1*** exchanges and
- ***(n − 1) + (n − 2) +...+ 2 + 1 = n(n − 1)/2*** comparisons.
- These observations hold, no matter what the input data is.
- In the worst case, this could be quadratic, but in the average case, this quantity is ***O(n log n)***. It implies that the **running time of Selection sort is quite insensitive to the input**.
-

Example

Unsorted list:	5	2	1	4	3

1st iteration:

Smallest = 5

2 < 5, smallest = 2

1 < 2, smallest = 1

4 > 1, smallest = 1

3 > 1, smallest = 1

Swap 5 and 1	1	2	5	4	3

2nd iteration:

Smallest = 2

2 < 5, smallest = 2

2 < 4, smallest = 2

2 < 3, smallest = 2

No Swap	1	2	5	4	3

3rd iteration:

Smallest = 5

4 < 5, smallest = 4

3 < 4, smallest = 3

Swap 5 and 3	1	2	3	4	5

4th iteration:

Smallest = 4

4 < 5, smallest = 4

No Swap	1	2	3	4	5

Finally,

the sorted list is	1	2	3	4	5

Time Complexity: $O(n^2)$ as there are two nested loops.

AuxiliarySpace: O(1)

The good thing about selection sort is it never makes more than O(n) swaps and can be useful when memory write is a costly operation.

Algorithm:find the kth smallest element

1. Algorithm Select1(a,n,k)
2. //selecet the kth smallest element in a[1:n] and place it in the kth position of a[].The remaining elements are rearranged such that a[m]<=a[k] for 1<=m<k,and a[m]>=a[k] for k<m<=n.
3. {low=1; up=n+1;
4. a[n+1]=∞ // a[n+1] is set to infinity

```
5..repeat
6. {
7. //each time the loop is entered 1<=low<=k<=up<=n+1
8. j=Partition(a,low,up)
9. //j is such that a[j] is the jth smallest value in a[].
10. If(k=j) then return;
11. Else if (k<j) then up=j // j is the new upper limit
12. Else low=j+1;//j+1 is the new lower limit
13. }until(flase);
14. }
```

```
14. Algorithm Partition(a,m,p)
15. //Within a[m],a[m+1],…..a[p-] the elements are rearranged in such a manner that if initially t=a[m],then after completion a[q]=t for some q between m and p-1,a[k],+t for m<=k<q,and a[k]>=t //for q<k<p.q is returned.Set a[p]=∞
16. { V=a[m]; i=m; j=p
17. Repeat
18. { Repeat
19. i=i+1;
20. Until(a[i]>=v);
21. Repeat
22. j=j-1
23. until(a[j]<=v)
24. if(i<j) then Interchange(a,i,j);
25. }until (i>=j)
26. a[m]=a[j] ;      a[j]= v;          return j;          }
1.Algortithm Intechange(a,i ,j)
2.//Exchange a[i] with a[j]
3.{p=a[i]
5.a[i]= a[j];
6.a[j]=p;
7.}
```

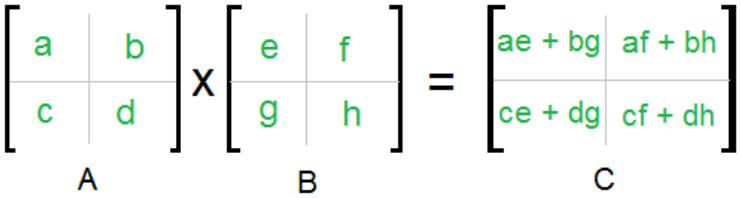
STRASSEN'S MATRIX MULTIPLICATION

Divide and Conquer

Following is simple Divide and Conquer method to multiply two square matrices.

1) Divide matrices A and B in 4 sub-matrices of size N/2 x N/2 as shown in the below diagram.

2) Calculate following values recursively. ae + bg, af + bh, ce + dg and cf + dh.



A, B and C are square metrices of size N x N
a, b, c and d are submatrices of A, of size N/2 x N/2
e, f, g and h are submatrices of B, of size N/2 x N/2

In the above method, we do 8 multiplications for matrices of size N/2 x N/2 and 4 additions. Addition of two matrices takes $O(N^2)$ time. So the time complexity can be written as

$T(N) = 8T(N/2) + O(N^2)$

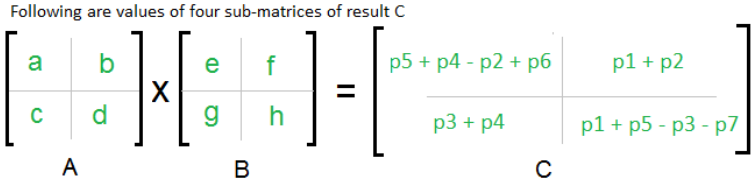
From Master's Theorem, time complexity of above method is $O(N^3)$

Simple Divide and Conquer also leads to $O(N^3)$, can there be a better way?

- In the above divide and conquer method, the main component for high time complexity is 8 recursive calls.
- The idea of **Strassen's method** is to reduce the number of recursive calls to 7. Strassen's method is similar to above simple divide and conquer method in the sense that this method also divide matrices to sub-matrices of size N/2 x N/2 as shown in the above diagram, but in Strassen's method, the four sub-matrices of result are calculated using following formulae.

p1 = a(f - h)
p2 = (a + b)h
p3 = (c + d)e
p4 = d(g - e)
p5 = (a + d)(e + h)
p6 = (b - d)(g + h)
p7 = (a - c)(e + f)

The A x B can be calculated using above seven multiplications.
Following are values of four sub-matrices of result C



A, B and C are square metrices of size N x N
a, b, c and d are submatrices of A, of size N/2 x N/2
e, f, g and h are submatrices of B, of size N/2 x N/2
p1, p2, p3, p4, p5, p6 and p7 are submatrices of size N/2 x N/2

Time Complexity of Strassen's Method

Addition and Subtraction of two matrices takes $O(N^2)$ time. So time complexity can be

written as
$T(N) = 7T(N/2) + O(N^2)$

From Master's Theorem, time complexity of above method is
$O(N^{Log7})$ which is approximately $O(N^{2.8074})$

Generally Strassen's Method is not preferred for practical applications for following reasons.

1) The constants used in Strassen's method are high and for a typical application Naive method works better.

2) For Sparse matrices, there are better methods especially designed for them.

3) The submatrices in recursion take extra space.

4) Because of the limited precision of computer arithmetic on noninteger values, larger errors accumulate in Strassen's algorithm than in Naive Method .