ITT304 AAD Module 2 - safefgrsr

Algorithm Analysis and Design (APJ Abdul Kalam Technological University)

Module-II

Divide and Conquer

Divide and Conquer - Control Abstraction, Finding Maximum and Minimum, Binary Search, Strassen's Matrix Multiplication, Quick Sort, Merge Sort

1. Divide and Conquer

A general paradigm for algorithm design; inspired by emperors and colonizers.

Given a function to compute on n inputs the divide-and-conquer strategy suggests splitting the inputs into k distinct subsets, $1 < k \leq n$ yielding k subproblems. These subproblems must be solved and then a method must be found to combine subsolutions into a solution of the whole. If the subproblems are still relatively large, then the divide-and-conquer strategy may possibly be reapplied. Often the subproblems resulting from a divide-and-conquer design are of the same type as the original problem.

The principle behind this strategy is that it is easier to solve several small instances of a problem than one large complex problem. The "divide - and - conquer" technique involves in solving a particular computational problem by dividing it into smaller sub problems, solving the problem recursively and then combining the results of all the sub problems to produce the result for the original complex problem 'P'.

An effective approach to design fast algorithms in sequential computation is the method known as ***divide and conquers***

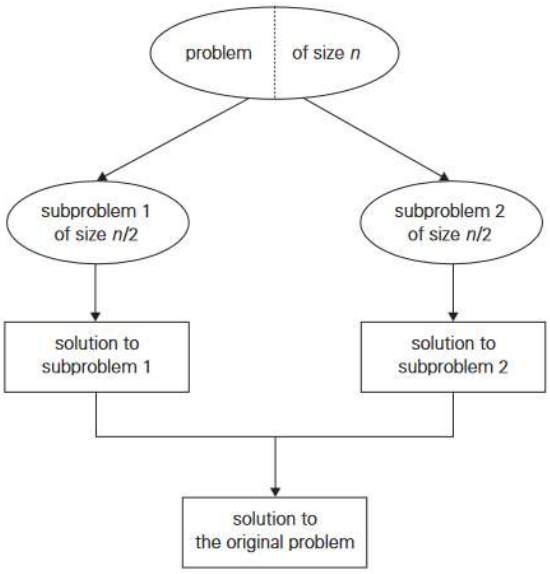
The strategy involves three steps at each level of recursion.

1. **Divide:-** Divide the problem into a number of sub problems.
2. **Conquer:-** Conquer the sub problems by solving them recursively. If the sub problem sizes are small enough, then just solve the sub problems in a straightforward manner.
3. **Combine:-** Combine the solutions to the sub problems to form the solution for the original problem.

Let 'n' represent the size of the original problem. Let S(n) denote this problem. We solve the problem S(n) by solving a collection of k sub problems- $S(n_1)$, $S(n_2)$,…$S(n_k)$, where $n_i<n$ for i=1,2,..k. Finally, we merge the solutions to these problems.

When the subproblems are large enough to solve recursively, we call that the **recursive case**. Once the subproblems become small enough that we no longer recurse, we say that the recursion "bottoms out" and that we have gotten down to the **base case**.

Examples: Binary Search, Merge sort, Quick sort, Matrix multiplication etc



We can write a control abstraction which mirrors the way an actual program based upon divide-and-conquer will look.

2. Control Abstraction

Control abstraction refers to a general procedure whose flow of control is clear but whose primary operations are specified by procedures which are not defined, or they are different for different procedures. The control abstraction can be written either iteratively or recursively.

Let the n inputs are stored in an array A[1: n] and assume that this array be global. Procedure DANDC is a function, which is initially invoked as DANDC(1,n). DANDC(p,q) solves a problem instance defined by the inputs A[p:q]. Then a control abstraction for divide and conquer can be written as follows.

```
Procedure DANDC(p,q)
Global n, A(1:n); integer m,p,q;     //1<=p<=q<=n
if (SMALL(p,q))
   then return(G(p,q));
else
    m←DIVIDE(p,q);    //p<=m<=q
    return(COMBINE(DANDC(p,m), DANDC(m+1,q)));
endif
end DANDC.
```

SMALL(p,q) is Boolean valued function, which returns true if the input size q-p+1 is small enough so that the answer can be computed without splitting. In this case, the function G is invoked. Otherwise the function DIVIDE(p,q) is called. This function returns an integer, which specifies where the input is to be split. Let m = DIVIDE(p,q). The input is splitted so that A(p,m) and A(m+1,q) defined instances of the two sub problems. Let x and y be the solutions of these two sub problems obtained by the recursive application of DANDC on instances A(p..m) and A(m+1..q) respectively. COMBINE(x,y) is a function which returns the solution to A(p..q).

If the sizes of these two sub problems are approximately equal, then the following recurrence relation can determine the computing time of DANDC.

$$T(n) = \begin{cases} g(n), & \text{n small} \\ 2T(n/2) + f(n), & \text{otherwise} \end{cases}$$

where T(n) is the time for DANDC for n inputs. g(n) is the time to compute the answer directly for small inputs and f(n) is the time for DIVIDE and COMBINE

The solution to the above problem is described by the recurrence, assuming size of P denoted by n

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

where f(n) is the time to divide n elements and to combine their solution.

Recursively decompose a large problem into a set of smaller problems

– Decomposition is directly reflected in analysis

– Run-time determined by the size and number of sub problems to be solved in addition to the time required for decomposition.

The complexity of many divide-and-conquer algorithms is given by 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. We assume that T(1) is known and n is a power of b. (ie, $n=b^k$). One method of solving such recurrence relation in substitution method.

3. Binary Search

Problem: Given a list of n sorted elements a_i, $1 \leq i \leq n$ sorted in ascending order. Check whether an element x is present in the list or not. If present, find the position of x in the list else return zero.

P(n, a_i,….,a_l, x) denote an instance of the problem.

where n : no of elements

a_i,….,a_l: elements in the list

x : element to be searched

Divide and conquer can be used for solving this problem. Let Small(p) is true if n=1.

S(P) will take the value of i if x=ai. Otherwise 0. Then g(1)= Θ (1).

If P has more than one element, divide the problem into sub problems as given below

- Pick an index q in the range [1, n]
- Compare x with a_q
  - $x = a_q$, solved
  - $x < a_q$, search x in sub list $a_i, a_{i+1}, \ldots, a_{q-1}$.

    Then P reduced to (q-i, a_i,….,a_{q-1}, x)
  - $x > a_q$, search x in sub list $a_{q+1}, a_{q+2}, \ldots, a_l$.

    Then P reduced to (*l*-q, a_{q+1},….,a_l, x)

This division takes θ(1) time. After comparing, remaining can be solved by using the same method. q is selected such that

q = $\lfloor (n+1)/2 \rfloor$ .Answer to new sub problem is the answer to the original problem P. So no need for combine function.

Program 2.1

```
int BinSearch (int a[ ], int n, int x)
{
    int low=1, high=n, mid;
    while (low <= high)
    {
            mid = (low + high)/2;
            if (x < a[mid])
                    high = mid – 1;
            else if (x > a[mid])
                    low = mid + 1;
```

```
            else
                    return (mid);
        }
        return (0);
    }
```

Let a_i, $1 \le i \le n$ be a list of elements which are stored in ascending order. We have to find out whether a given element x is present in this list by using binary search. If x is present, we have to determine the value of j, such that a(j)=x. If x is not in the list, the j is to be set to 0. Divide and Conquer suggests breaking up any instance I=(n, a_1, $a_2 \ldots a_n$, x) of this search problem into sub instances. One possibility is to pick an index k and obtain three instances:

$I_1 = (k-1, a_1, a_2, \ldots\ldots\ldots\ldots a_{k-1}, x)$
$I_2 = (1, a_k, x)$
$I_3 = (n-k, a_{k+1}, \ldots\ldots\ldots\ldots a_n, x)$

The search problem for two of these 3 instances is easily solved by comparing x with a_k. If $x = a_k$, then j=k and I_1 and I_3 need not be solved. If $x < a_k$, then for I_2 and I_3, j=0 and only I_1 need to be solved. If $x > a_k$, then for I_1 and I_2, j=0 and only I_3 remains to be solved. After comparison with a_k, the instance remaining be solved (if any) can be solved by using this divide and conquer scheme again. If k is always chosen such that a_k is the middle element (i.e. k=(n+1)/2), then the resulting search algorithm is known as binary search. The algorithm is as follows:

Algorithm Binsrch(a, i, *l*, x)

//Given an array a[i : *l*] of elements in non-decreasing order, $1 \le i \le l$, determine whether x is //present, and if so, return i such that x=a[i].

```
{
  if (l = i) then
     {
       if (x = a[i] then return i
       else return 0;
      }
  else
      {
        mid = [(i+l)/2];
        if (x = a[mid] then return mid;
        else if (x < a[mid]) then
                return Binsrch(a,i,mid-1,x);
            else return Binsrch(a,mid+1,l,x);
       }
}
```

Simulation

Consider 10 elements 10, 20, 30, 40, 50, 60, 70, 80, 90, 100

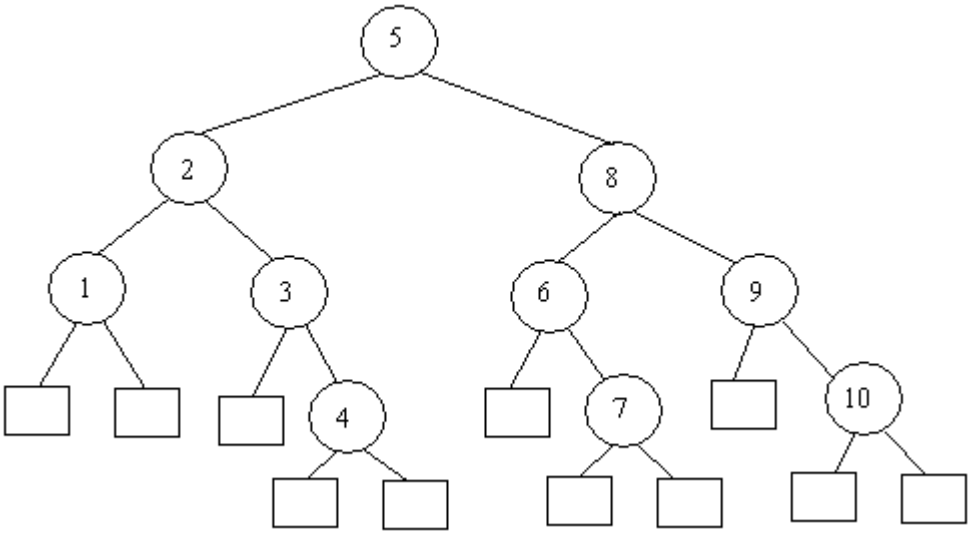
Number of comparisons needed to search element is as per the table given below

Position	1	2	3	4	5	6	7	8	9	10
Element	10	20	30	40	50	60	70	80	90	100
No. of comparisons required	3	2	3	4	1	3	4	2	3	4

No element requires more than 4 comparisons.

Average comparison=29/10=2.9 (for successful search)

The binary decision tree for the above list of elements will be as given below.



If the element is present, then search end in a circular node (inner node), if the element is not present, then it ends up in a square node (leaf node)

There are 11 possible ways that an unsuccessful search may terminate depends on value of x.

Now we will consider the worst case, average case and best-case complexities of the algorithm for a successful and an unsuccessful search.

Worst Case

If $2^{k-1} <= n <= 2^k$ then all circular nodes at level 1, 2,….k and all square nodes at levels k and k+1. Root at level 1.

Then for a successful search, it will end up in either of the k inner nodes. Hence the complexity is O(k), which is equal to $O(\log_2 n)$.

For an unsuccessful search, it needs either k-1 or k comparisons. Hence complexity isΘ (k), which is equal to $\Theta(\log_2 n)$.

Theorem

If n is in the range [2^{k-1}, 2^k) then BinSearch makes atmost k element comparisons for a successful search and either k-1 or k comparisons for an unsuccessful search. In other words the time for a successful search is O(log n) and for an unsuccessful search is θ(log n).

Proof

Consider the binary decision tree describing the action of BinSearch on n elements. All successful searches end at a circular node whereas all unsuccessful searches end at a square node. If $2^{k-1} <= n <= 2^k$, then all circular nodes are at levels 1, 2,...,k whereas all square nodes are at levels k and k+1. The number of element comparisons needed to terminate at a circular node on level i is i whereas the number of element comparisons needed to terminate at a square node at level i is i-1.

Average Case

Let I (Internal path length) and E (External path length) represent the sum of distance of all internal nodes from root and sum of distance of all external nodes from the root respectively.

$E = I + 2n$

Number of element comparison to terminate at circular node at level i is i.

Number of element comparison to terminate at square node at level i is i-1.

Let $A_s(n)$ and $A_u(n)$ represents average case complexity of a successful and unsuccessful search respectively.

The number of comparisons needed to find an element represented by an internal node is more than the distance of the node from the root.

$A_s(n) = 1 + I/n$

Let $A_u(n)$ is the number of comparisons needed to find an element represented by an external node.

$A_u(n) = E / (n+1)$

$A_s(n) = 1 + I/n$

$= 1 + (E-2n)/n$

$= 1 + (A_u(n)(n+1) – 2n)/n$

$= (n + (A_u(n)(n+1) – 2n))/n$

$= (n(A_u(n) -1) + A_u(n))/n$

$= A_u(n) – 1 + A_u(n)/n$

$A_s(n) = A_u(n)(1+1/n) – 1$

From this formula we see that As(n) and Au(n) are directly related. The minimum value of As(n) (and hence Au(n)) is achieved by an algorithm whose binary decision tree has minimum external and internal path length. This minimum is achieved by the binary tree all of whose external nodes are on adjacent levels, and this is precisely the tree that is produced by binary search. $A_s(n)$ and $A_u(n)$ are directly related and are proportional to $\log_2 n$.

Hence the average case complexity for a successful and unsuccessful search is $O(\log_2 n)$

Best Case

Best case for a successful search - there is only one comparison (element at middle)

Hence complexity is $\Theta(1)$

Best case for an unsuccessful search is $O(\log_2 n)$

	Best Case	**Average Case**	**Worst Case**
Successful Search	$\Theta(1)$	$O(\log_2 n)$	$O(\log_2 n)$
Unsuccessful Search	$O(\log_2 n)$	$O(\log_2 n)$	$O(\log_2 n)$

4. Merge Sort

The merge sort algorithm divides an array into two equal pieces, sorts them recursively, and then merges them back together. There for it is an example of the **divide and conquer a**lgorithmic paradigm. **Merge sort** is an $\underline{O}(n \log n)$ comparison-based sorting algorithm. In most implementations it is **stable,** meaning that it preserves the input order of equal elements in the sorted output.

Conceptually, a merge sort works as follows:

1. If the list is of length 0 or 1, then it is already sorted. Otherwise:
2. Divide the unsorted list into two sub lists of about half the size.
3. Sort each sub list **recursively** by re-applying merge sort.
4. **Merge** the two sub lists back into one sorted list.

Merge sort incorporates two main ideas to improve its runtime:

1. A small list will take fewer steps to sort than a large list.
2. Fewer steps are required to construct a sorted list from two sorted lists than two unsorted lists. For example, you only have to traverse each list once if they're already sorted (see the **merge** function below for an example implementation).

Procedure:

Using merge sort to sort a list of integers contained in an **array.** As the basic idea behind merge sort is splitting the given array into two equal sized sets using the divide and conquer method, and combining is done called a merging of these two sets, after sorting.

Suppose we have a sequence of 'n' elements A[1]……A[n] , it is split into subsequent set as A[1],…..,A[(n/2)] and A[(n/2)+1],……..,A[n], where each set is individually sorted, by again splitting it into subsequent smaller sets.

The merge sort Algorithm proceeds using recursion and function Merge, which merges two sorted sets to form a single set. The algorithm **Merge Sort (1, n)** rearranges the values in the array 'A' in non-decreasing order.

Algorithm: (To perform sort using *Merge sort* method)

Algorithm MERGE_SORT (low, high)

```
// A [Low: High] is a global array to be sorted //
{
        If (Low < High)
        {
                Mid = [ (Low + High) /2 ];
                                //finding position to split the array //
                Mergesort (Low, Mid);
                Mergesort (Mid + 1, High);
                                // solving the Sub problems //
                Merge (Low, Mid, High);
                                // Combine the solutions .//
        }
}
```

Sub-Algorithm: (To merge the sub lists generated in the *Merge sort* method)

Algorithm Merge (Low, Mid, High)

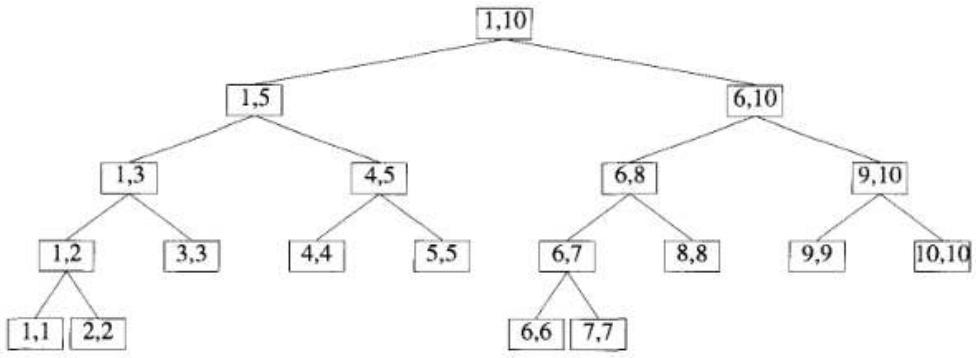
/* A [Low: High] is a global array with two sorted subset A [Low: Mid] and A [Mid + 1: High]. The goal is to merge these two sets into a single set residing in A[Low: High]. B [ ] is an auxiliary global array used to temporarily store the intermediate values */

{
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 ];
}
```

Example

Consider 10 elements 310, 285, 179, 652, 351, 423, 861, 254, 450 and 520. The recursive call can be represented using a tree as given below.



Tree of calls of MergeSort(1, 10)

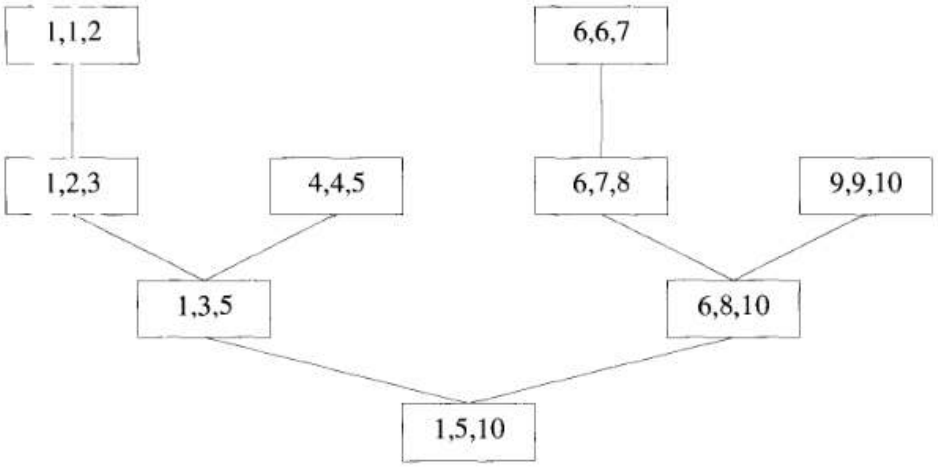
After the execution of the Algorithm Merge sort (A [1 : 10]), splits the array A [1: 10] into A [1 : 5] , A [6 : 10]. The elements in the array A [1 : 5] is again split into A [1 : 3] and A [4 : 5]. The array A [1 : 3] is again split into sub array as A [1 : 2] and A [3 : 3], where the array A [3 : 3] is already sorted. The array A [1 : 2] is again split into one element sub arrays.

The array A [ ] now becomes,

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

Similarly, the array A [6 : 10] is split as the first array and the sub arrays are merged as by sorting each of the smaller sub arrays, to get a independent sorted files, which at the end is merged to get a completely sorted array of 'n' elements. The Merge Sort uses 2n locations, i.e., additional 'n' locations are needed, because in the Algorithm array A[i] and array B[i], which is an auxiliary array are used to execute the sorting operation. After the completion of the recursive procedure calls by the Merge Sort Algorithm applied in the ten elements in the array A [i], the pair of parameters of the function **Mergesort**, Low and High could be represented as a tree of calls as shown in the figure: above.

After the completion of the calls of Merge Function recursively, the Low, Mid, High values are as represented in the figure below



Tree of calls of Merge

Complexity of Merge sort

The complexity of Merge Sort is O (n log n). The disadvantage of Merge Sort is the use of 2n locations, i.e., additional 'n' locations are needed, because in the Algorithm array A[i] and array B[i], which is an auxiliary array are used to execute the sorting operation.

The recurrence relation for the algorithm can be written as

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

When n is power of 2, we can write $n = 2^k$

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

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

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

$= \ldots$

$= 2^kT(1) + kcn$

$T(n) = an + cnlogn$

If $2^k<n\leq2^{k+1}$, $T(n)\leq T(2^{k+1})$

Hence **T(n) = O(nlogn)**

One complaint we might raise concerning merge sort is its use of 2n locations. The additional n locations were needed because we couldn't reasonably merge two sorted sets in place.
Solution????
Another complaint we could raise about Merge Sort is the stack space that is necessitated by the use of recursion.
Solution???

5. Quick Sort (https://www.youtube.com/watch?v=7h1s2SojIRw)

Problem: Given a sequence of n elements a[1],...,a[n]. Sort this list in ascending order.

Divide the list into two sub arrays and sorted so that the sorted sub arrays need not be merged later. This is done by rearranging elements in the array a[1:n] such that a[i]<=a[j] for $1\leq i\leq m$ and $m+1\leq j\leq n$, $1\leq m\leq n$. Thus the elements in a[1:m] and a[m+1:n] can be independently sorted and no merging is needed.

This rearrangement is done by picking an element, m=a[s], reorder other elements so that all elements appearing before m is less than or equal to m, all elements after m is greater than or equal to m. This rearranging is called **partitioning**.

Procedure:

In quick sort, the array A [1: n] is divided into two sub arrays where the sorted sub arrays need not be merged at the ends, as in a merge sort algorithm. It is done by rearranging the elements A[1:n], such that A[i] <= A[j], for all 'i' between 1 and n, and all 'j' between m+1 and n, for any value of m considered in the list as 1<=m<=n. Therefore the elements, A [1: m] and A [m+1: n] is independently sorted, without any merging. The rearrangement of elements as A[1: m] and A[m+1 : n] is achieved by selecting any element, consider t = A[s]. Based on t=A[s], all elements appearing before t=A[s] in A[1:n] are less than or equal(<=) to ' t', and all elements after 't' are greater than or equal to 'y'. This rearrangement is referred as

Partitioning and the element A[s] is referred as **pivot or partition element**. Based on the method of partitioning a set of elements, based on a chosen element (which is selected based on divide and conquer method) for sorting 'n' elements, will divide the set into two subsets as S1 and S2. Here, all elements in S1 is S1<=A[m] and S2.

The steps are:

1. Pick an element, called a ***pivot***, from the list.
2. Reorder the list so that all elements which are less than the pivot come before the pivot and so that all elements greater than the pivot come after it (equal values can go either way). After this partitioning, the pivot is in its final position. This is called the **partition** operation.
3. **Recursively** sort the sub-list of lesser elements and the sub-list of greater elements.

Algorithm: (To perform sort using *Quick sort* method)

Algorithm QUICK_SORT (A, N)

```
// The Algorithm sorts all elements A[p]……A[q] in the array A [1: n] in ascending
order, A [n+1] is defined and must be >= to all elements in A [1: n] //
    {
    If (p<q) then
      {
            j = partition (A, p, q+1);          // j refers to position of partition
    element //
      Quicksort ( p, j-1);
      Quicksort (j+1 , q);
            // solve the sub problems which are not combined for solutions //
      }

}
```

Sub-Algorithm: (To solve the sub lists, partitioned using a partition element in the Selection *sort* method)

Algorithm partition (A, m, p)

```
//Set A[p] =∞, the partition element is t= A[m], within A[m], A [m+1]……., A[p-1]
elements that area to be rearranged.//
{
        V = A[m];          i = m;     j = p;
        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;
        }
```

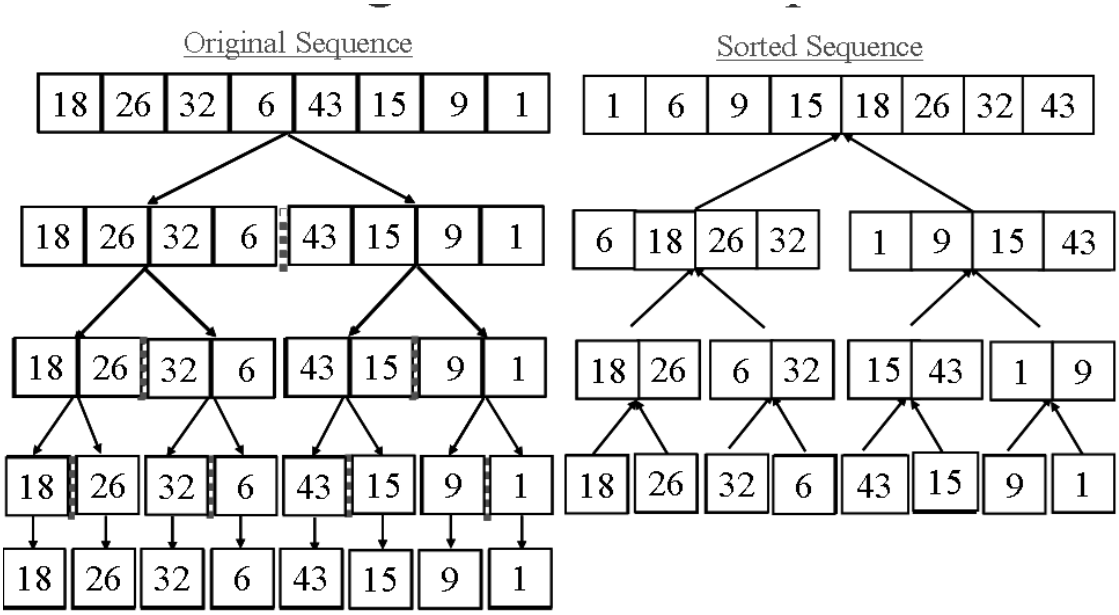
Sub-Algorithm: (To exchange or interchange A[i] with A[j])

Algorithm Interchange (A, i, j)

```
        // To interchange A[i] with A[j] //
{
        P = A[i];
        A[i] = A[j];
        A[j] = p;

    }
```

Example:



In analysing quick sort, we count only the element comparisons C(n). It is easy to see that the frequency count of other operations is of the same order as C(n). Assume that n elements are distinct, and the input distribution is such that the partition element v=a[m] in the call of Partion(a,m,p) has the equal probability of being the ith smallest element 1<=i<=(p-m) in a[m:p-1].

Worst case complexity (https://www.youtube.com/watch?v=-qOVVRIZzao)

Assume that each time after fixing an element the file is divided into a smaller partition and a bigger partition. Say, during i^{th} partition it is divided into 1 and (i-1).

No. of comparison = $n + (n-1) + (n-2) + \ldots + 2$

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

$= \quad O(n^2)$

First, let us obtain the worst-case value $C_w(n)$ of C(n). The number of element comparisons in each call of Partition is at most p-m + 1. Let r be the total number of elements in all the calls to Partition at any level of recursion. At level one only one call, Partition (a, l, n+l), is made and r = n; at level two at most two calls are made and r = n - 1; and soon. At each level of recursion,0(r) element comparisons are made by Partition. At each level, r is at least one less than the r at the previous level as the partitioning elements of the previous level are eliminated. Hence $C_w(n)$ is the sum on r as r varies from 2 to n, or $0(n^2)$.

Average case complexity

The average value $C_A(n)$ of C(n) is much less than $C_w(n)$.

The partitioning element has equal probability of becoming the i^{th} smallest element, $1 \leq i \leq p-m$ in the array a[m:p-1].

Hence probability for sub arrays being sorted, a[m:j], a[j+1:p-1]is 1/(p-m), $m \leq j < p$.

$$C_A(n) = n+1 + 1/n \sum_{k=1}^{n} [C_A(k-1) + C_A(n-k)]$$

$$nC_A(n) = n(n+1) + \sum_{k=1}^{n} [C_A(k-1) + C_A(n-k)]$$

$$nC_A(n) = n(n+1) + 2 [C_A(0) + C_A(1)+\ldots+ C_A(n-1)] \quad \text{---------- (1)}$$

Replacing n by n-1 in equ(1)

$$(n-1)C_A(n-1) = n(n-1) + 2 [C_A(0) + C_A(1)+\ldots+ C_A(n-2)] \quad \text{---------- (2)}$$

(1) – (2) =>

$$nC_A(n) - (n-1)C_A(n-1) = 2n + 2C_A(n-1)$$

Dividing by n(n+1)

$$nC_A(n)/(n+1) = 2/(n+1) + ((n+1)\ C_A(n-1))/(n(n+1))$$
$$C_A(n)/(n+1) = 2/(n+1) + C_A(n-1)/n$$
$$C_A(n)/(n+1) = C_A(n-1)/n + 2/(n+1)$$

Substituting for $C_A(n-1)$, $C_A(n-2)$, …

$$C_A(n)/(n+1) = C_A(n-2)/(n-1) + 2/n + 2/(n+1)$$
$$= \ldots.$$
$$= C_A(1)/2 + 2/2^{n-1} + 2/(n+1)$$
$$= C_A(1)/2 + 2 \quad 1/k$$
$$\leq \int_{2}^{n+1} 1/x\ dx = \log_e(n+1) - \log_e 2$$

$$C_A(n) = 2(n+1)[\log_e(n+1) - \log_e 2]$$
$$C_A(n) = O(n \log n)$$

6. Finding Maximum and Minimum (https://www.youtube.com/watch?v=-7h1AlVKijA)

Problem: Given a list of n elements a_i, $1 \leq i \leq n$, find the maximum and the minimum element from the list.

Given below is a straightforward method to calculate the maximum and minimum from the list of n elements.

1. Straight forward algorithm.

This algorithm is used to find the maximum and minimum items from a set of n elements.

```
Procedure straightmaxmin (A, n, max, min)
 // set max to the maximum and min to the minimum of A[1:n] //
{
integer i, n;
max ←min←A[i];
for i: =2 to n do
 {
 if A[i] > max then
          max = A[i];
      if A[i] <min then
          min = A[i];
 }
}.
```

Number of comparisons required in this algorithm is 2*(n-1), in the best, average, and worst cases. To reduce the number of comparisons, we can replace the contents of the for loop by,

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

According to this modification, the number of comparisons in the best case = n-1. Best case occurs when the numbers are in increasing order. The number of comparisons in the worst case = 2*(n-1). The worst case occurs when the elements are in decreasing order.

2. A recursive algorithm to find maximum and minimum using divide and conquer

Using divide and conquer technique the given instance I = (n, A [1], A[2],……..A[n]) can be divided into two smaller instances I_1 and I_2 such that.

I_1= [n/2, A(1), A(2),……..A(n/2)]

I_2= [(n-n/2), A(n/2+1), A(n/2+2)…….A(n)].

If max(I) and min(I) are the maximum and minimum elements in I, then

max(I) = the larger of max(I_1) &max (I_2)

min(I) = the smallest of min(I_1) &min(I_2).

If I contain only one element, then the answer can be computed without splitting.

The following procedure MaxMin is a recursive procedure, which finds the maximum and minimum of the set of elements {A[i], A[i+1], …….,A[j]}. The situation of set sizes i = j and i = j-1 are handled separately. For set contain more than two elements, the midpoint is determined, and two new sub problems are generated. Then the maximum and minimum of the sub problems are found out separately and combine these results to find out the maximum and minimum of the entire list.

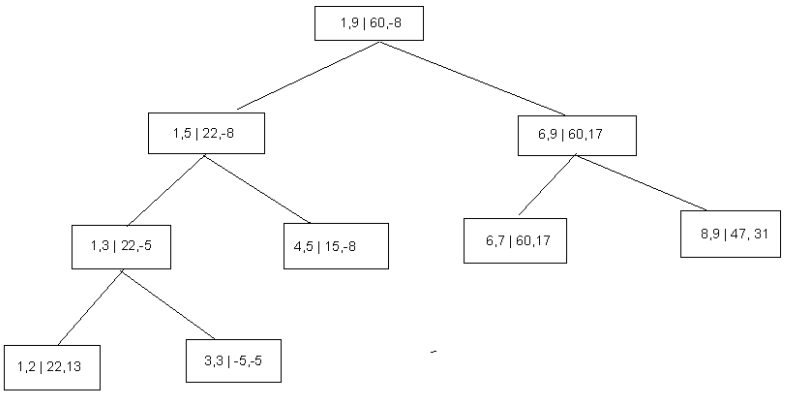
```
Algorithm MaxMin (i, j, max, min)
{
 integer i, j;
 global n, A[1:n]
 case
    : i = j :  max = min = A[i];
    : i = j-1:  if (A[i] < A[j] ) then max: =A[j];
                      min: =A[i];
                else
                      max= A[i];  min=A[j];
                endif.
        : else: mid = [(i+j)/2]
                call MaxMin (i, mid, max, min)
                call MaxMin(mid+1, j, max1, min1)
                if (max<max1) then max=max1;
                if (min>min1) then min=min1;

  endcase
 }MAXMIN
```

Example:

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

The recursive call can be represented using a tree as given below



The maximum and minimum of a[1] and a[2] is 22 and 13. The maximum and minimum of a[3] and a[3] is -5 and -5. Thus the maximum and minimum of a[1], a[2] and a[3] is 22 and -5 and so on. Finally we get the maximum and minimum of the given array as 60 and -8 Time complexity can be represented by the recurrence

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

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

$$\begin{aligned} T(n) &= 2T(n/2) + 2 \\ &= 2\,(2T(n/4) + 2) + 2 \\ &= 4T(n/4) + 4 + 2 \\ &= \ldots \\ &= 2^{k-1} T(2) + \sum_{i=1}^{k-1} 2^i \\ &= 2^{k-1} + 2^k - 2 \\ &= 2^{k-1} + n - 2 \\ &= n/2 + n - 2 \end{aligned}$$

$T(n) = 3n/2 - 2$ (Best, average, Worst)

Comparing with the Straightforward method, Divide and Conquer is 50% more faster. But additional stack space is required for recursion.

If we include position comparison also, while calculating complexity. For this rewrite Small(P) as

if (i >= j-1) (Small(P)}

$$C(n) = \begin{cases} 2C(n/2) + 3 & n>2 \\ 2 & n=2 \end{cases}$$

$C(n) = 2C(n/2) + 3$
$= 4C(n/4) + 6 + 3$
$= 4C(n/4) + 3(1+2)$
$= ...$
$= 2^{k-1}C(2) + 3\sum_{i=1}^{k-2} 2^i$
$= 2^{k-1}.2 + 3[2^{k-1}-2]$
$= 2^k + 3.2^{k-1}-3$
$= n + 3.n/2 - 3$
$C(n) = 5n/2 - 3$

While using StraightMinMax, the comparison is 3(n-1) which is greater than 5n/2–3. Even though the element comparison is less for MaxMin, it is slower than normal method because of the stack.
Both algorithm having the complexity O(n).

If element comparison is costlier than the index comparison divide and conquer is good else otherwise is good.

7. Matrix Multiplication

Let A and B are two nxn matrices. Suppose we want to multiply these two matrices *A* x *B* = *C*.

Algorithm

```
void matrix_mult (){
 for (i = 1; i <= n; i++) {
     for (j = 1; j <= n; j++) {
                    compute Ci,j;
                      }
 }}
```

$$C_{i,j} = \sum_{k=1}^{N} a_{i,k} b_{k,j}$$

$$\text{Thus } T(N) = \sum_{i=1}^{N}\sum_{j=1}^{N}\sum_{k=1}^{N} c = cN^3 = O(N^3)$$

7.1 Matrix Multiplication Divide & Conquer

For this matrix multiplication divide and conquer method can be used. Assume n is the power of 2, that is, that there exists a nonnegative integer k such that $n = 2^k$. If not, add enough zeros to rows and columns to make power of two.

Given two matrices A and B of size n x n.

$$\begin{vmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{vmatrix} = \begin{vmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{vmatrix} \begin{vmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{vmatrix}$$

C11 = a11b11 + a12b21

C12 = a11b12 + a12b22

C21 = a21b11 + a22b21

C22 = a21b12 + a22b22

```
SQUARE-MATRIX-MULTIPLY-RECURSIVE(A, B)
1  n = A.rows
2  let C be a new n × n matrix
3  if n == 1
4      c11 = a11 · b11
5  else partition A, B, and C as in equations (4.9)
6      C11 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A11, B11)
           + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A12, B21)
7      C12 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A11, B12)
           + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A12, B22)
8      C21 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A21, B11)
           + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A22, B21)
9      C22 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A21, B12)
           + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A22, B22)
10 return C
```

If n=2 we can use the above formula. If n>2 calculate product by recursively using the same formula. Multiplication, addition operation applied to matrices of size n/2 X n/2. This algorithm will continue applying itself to smaller- sized sub matrices until n becomes small (n=2). So the product is computed directly.

$$\begin{array}{|c|c|} \hline A_0 & A_1 \\ \hline A_2 & A_3 \\ \hline \end{array} \times \begin{array}{|c|c|} \hline B_0 & B_1 \\ \hline B_2 & B_3 \\ \hline \end{array} = \begin{array}{|c|c|} \hline A_0\times B_0+A_1\times B_2 & A_0\times B_1+A_1\times B_3 \\ \hline A_2\times B_0+A_3\times B_2 & A_2\times B_1+A_3\times B_3 \\ \hline \end{array}$$

2x2 matrix multiplication can be accomplished in 8 multiplications and 4 additions. Since two n/2 x n/2 matrices can be added in time cn^2 for some constant c, the overall computing time T(n) is

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

Where a and b are constants.

By solving $T(n)= O(n^3)$

In the above method no improvement has been made.

Since matrix multiplications are more expensive than matrix additions ($O(n^3)$ VS $O(n^2)$), Strassen reformulate the equation for C_{ij} so as to have fewer multiplications and possibly more additions.

7.2 Strassen's Matrix Multiplication

Volkar Strassen discovered a way to compute 2x2 matrix multiplication can be accomplished in 7 multiplication and 18 additions or subtractions.

Consider the follwing example.

$$\begin{vmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{vmatrix} = \begin{vmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{vmatrix} \begin{vmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{vmatrix}$$

$$P_1 = (A_{11}+ A_{22})(B_{11}+B_{22})$$
$$P_2 = (A_{21} + A_{22}) * B_{11}$$
$$P_3 = A_{11} * (B_{12} - B_{22})$$
$$P_4 = A_{22} * (B_{21} - B_{11})$$
$$P_5 = (A_{11} + A_{12}) * B_{22}$$
$$P_6 = (A_{21} - A_{11}) * (B_{11} + B_{12})$$
$$P_7 = (A_{12} - A_{22}) * (B_{21} + B_{22})$$

$$C_{11} = P_1 + P_4 - P_5 + P_7$$
$$C_{12} = P_3 + P_5$$
$$C_{21} = P_2 + P_4$$
$$C_{22} = P_1 + P_3 - P_2 + P_6$$

It has 4 steps

1. Divide the input matrices A and B and output matrix C into $n/2 \times n/2$ submatrices, as in equation (4.9). This step takes $\Theta(1)$ time by index calculation, just as in SQUARE-MATRIX-MULTIPLY-RECURSIVE.
2. Create 10 matrices $S_1, S_2, \ldots, S_{10}$, each of which is $n/2 \times n/2$ and is the sum or difference of two matrices created in step 1. We can create all 10 matrices in $\Theta(n^2)$ time.
3. Using the submatrices created in step 1 and the 10 matrices created in step 2, recursively compute seven matrix products $P_1, P_2, \ldots, P_7$. Each matrix P_i is $n/2 \times n/2$.
4. Compute the desired submatrices $C_{11}, C_{12}, C_{21}, C_{22}$ of the result matrix C by adding and subtracting various combinations of the P_i matrices. We can compute all four submatrices in $\Theta(n^2)$ time.

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

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

where a and b are constants. Working with this formula, we get

$$\begin{aligned} T(n) &= an^2[1 + 7/4 + (7/4)^2 + \cdots + (7/4)^{k-1}] + 7^k T(1) \\ &\le cn^2(7/4)^{\log_2 n} + 7^{\log_2 n}, \ c \text{ a constant} \\ &= cn^{\log_2 4 + \log_2 7 - \log_2 4} + n^{\log_2 7} \\ &= O(n^{\log_2 7}) \approx O(n^{2.81}) \end{aligned}$$

So T(n)=O($n^{2.81}$)