MARUDHAR KESARI JAIN COLLEGE FOR WOMEN

SUBJECT NAME: DESIGN AND ANALYSIS OF ALGORITHMS

SUBJECT CODE: CCA53

UNIT- II

General Method – Binary Search – Recurrence Equation for Divide and Conquer – Finding the Maximum and Minimum–– Merge Sort – Quick Sort – Performance Measurement – Randomized Sorting Algorithm – Selection Sort – A Worst Case Optimal Algorithm – Implementation of Select2 – Stassen"s Matrix Multiplications.

General Method:

Divide-and-conquer method: Divide-and-conquer is probably the best known general algorithm design technique. The principle behind the Divide-and-conquer algorithm design technique is that it is easier to solve several smaller instance of a problem than the larger one.

The "divide-and-conquer" technique involves solving a particular problem by dividing it into one or more cub-problems of smaller size, recursively solving each sub-problem and then "merging" the solution of sub-problems to produce a solution to the original problem.

Divide-and-conquer algorithms work according to the following general plan.

1. **Divide:** Divide the problem into a number of smaller sub-problems ideally of about the same size.
2. **Conquer:** The smaller sub-problems are solved, typically recursively. If the sub-problem sizes are small enough, just solve the sub-problems in a straight forward manner.
3. **Combine:** If necessary, the solution obtained the smaller problems are connected to get the solution to the original problem.

The following figure shows-

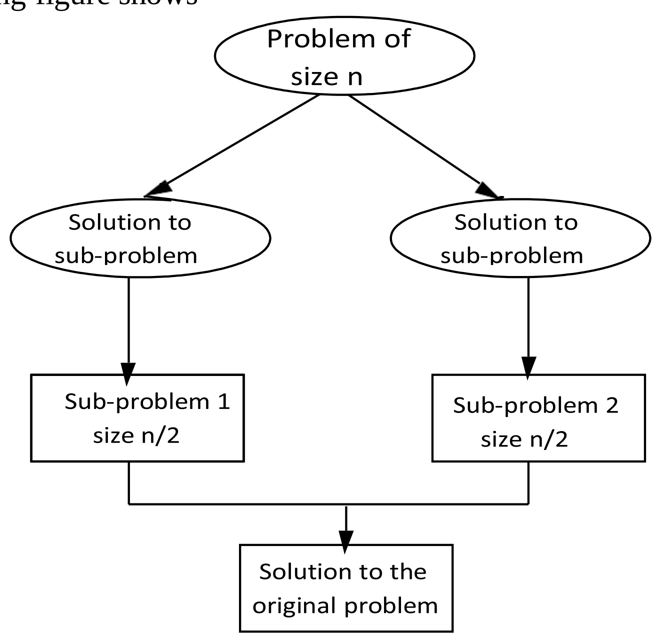


Fig: Divide-and-Conquer technique (Typical case).

Control abstraction for divide-and-conquer technique:

Control abstraction means a procedure whose flow of control is clear but whose primary operations are satisfied by other procedure whose precise meanings are left undefined.

Algorithm DandC(p)

```
{
   if small (p) then
   return S(p)
   else
     {
        Divide P into small instances P1, P2, P3 .........Pk, k≥1;
        Apply DandC to each of these sub-problems;\
        return combine (DandC(P1), DandC(P1),…. (DandC(Pk);
     }
 }
```

Algorithm: Control abstraction for divide-and-conquer

DandC(p) is the divide-and-conquer algorithm, where P is the problem to be solved. Small(p) is a Boolean valued function(i.e., either true or false) 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 sub-problems. These sub-problemsP_1, P_2, P_3.............. P_k, are solved by receive applications of DandC.

Combine is a function that combines the solution of the K sub-problems to get the solution for original problem 'P'.

Example: Specify an application that divide-and-conquer cannot be applied.

Solution: Let us consider the problem of computing the sum of n numbers a_0, a_1, a_{n-1}. If n>1, we divide the problem into two instances of the same problem. That is to compute the sum of the first [n/2] numbers and to compute the sum of the remaining [n/2] numbers. Once each of these two sum is compute (by applying the same method recursively), we can add their values to get the sum in question-

$$a_0+a_1+....+a_{n-1}= (a_0+a_1+....+a_{[n/2]-1})+a_{[n/2]-1}+......+a_{n-1}).$$

For example, the sum of 1 to 10 numbers is as follows- (1+2+3+4+ ....+10) = (1+2+3+4+5)+(6+7+8+9+10)

= [(1+2) + (3+4+5)] + [(6+7) + (8+9+10)]

= .....

= .....

= (1) + (2) +…………..+ (10).

This is not an efficient way to compute the sum of n numbers using divide-and-conquer technique. In this type of problem, it is better to use brute-force method.

Applications of Divide-and Conquer: The applications of divide-and-conquer methods are-

1. Binary search.
2. Quick sort
3. Merge sort.

Binary Search:

Binary search is an efficient searching technique that works with only sorted lists. So the list must be sorted before using the binary search method. Binary search is based on divide-and-conquer technique.

The process of binary search is as follows:

The method starts with looking at the middle element of the list. If it matches with the key element, then search is complete. Otherwise, the key element may be in the first half or second half of the list. If the key element is less than the middle element, then the search continues with the first half of the list. If the key element is greater than the middle element, then the search continues with the second half of the list. This process continues until the key element is found or the search fails indicating that the key is not there in the list.

Consider the list of elements: -4, -1, 0, 5, 10, 18, 32, 33, 98, 147, 154, 198, 250, 500.
Trace the binary search algorithm searching for the element -1.

Sol: The given list of elements are:

Low High

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
-4	-1	0	5	10	18	27	32	33	98	147	154	198	250	500

Searching key '-1': Here the key to search is '-1'First calculate mid;

Mid = (low + high)/2

= (0 +14) /2 =7

Low Mid High

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
-4	-1	0	5	10	18	27	32	33	98	147	154	198	250	500

←——First Half ——→ ←——Second Half——→

Here, the search key -1 is less than the middle element (32) in the list. So the search process continues with the first half of the list.

Low						High								
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
-4	-1	0	5	10	18	27	32	33	98	147	154	198	250	500

Now mid = (0+6)/2
=3.

Low			Mid			High								
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
-4	-1	0	5	10	18	27	32	33	98	147	154	198	250	500

← First Half ⟶ ⟵Second Half⟶

The search key '-1' is less than the middle element (5) in the list. So the search process continues with the first half of the list.

Low		High												
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
-4	-1	0	5	10	18	27	32	33	98	147	154	198	250	500

Now mid= (0+2)/2
=1

Low	Mid	High												
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
-4	-1	0	5	10	18	27	32	33	98	147	154	198	250	500

Here, the search key -1 is found at position 1.

The following algorithm gives the *iterative binary Search Algorithm*

```
Algorithm BinarySearch(a, n, key)
 {
   // a is an array of size n elements
   // key is the element to be searched
   // if key is found in array a, then return j, such that
   //key = a[i]
   //otherwise return -1.
     Low: = 0;
     High: = n-1;
    While (low ≤ high) do
      {
        Mid: = (low +
        high)/2;If ( key =
        a[mid]) then
             Return mid;
        Else if (key < a[mid])
            {
              High: = mid +1;
            }
       Else if( key > a[mid])
           {
```

```
                  Low: = mid +1;
                }
        }
```

The following algorithm gives *Recursive Binary Search*

```
  Algorithms Binsearch ( a, n, key, low, high)
      {
       // a is array of size n
      // Key is the element to be searched
     // if key is found then return j, such that key = a[i].
    //otherwise return
-1If ( low ≤ high) then
  {
    Mid: = (low + high)/2;
         If ( key = a[mid])
             thenReturn mid;
         Else if (key < a[mid])
             Binsearch ( a, n, key, low, mid-
         1);Else if ( key > a[mid])
             Binsearch ( a, n, key, mid+1, high);
    }
  Return -1;
}
```

Advantages of Binary Search: The main advantage of binary search is that it is faster than sequential (linear) search. Because it takes fewer comparisons, to determine whether the given key is in the list, then the linear search method.

Disadvantages of Binary Search: The disadvantage of binary search is that can be applied toonly a sorted list of elements. The binary search is unsuccessful if the list is unsorted.

Efficiency of Binary Search: To evaluate binary search, count the number of comparisons inthe best case, average case, and worst case.

<u>Best Case:</u> The best case occurs if the middle element happens to be the key element. Then only one comparison is needed to find it. Thus the efficiency of binary search is O(1).

Ex: Let the given list is: 1, 5, 10, 11, 12.

Low		Mid		High
1	5	$\boxed{10}$	11	12

Let key = 10.

Since the key is the middle element and is found at our first attempt.

<u>Worst Case:</u> Assume that in worst case, the key element is not there in the list. So the process of divides the list in half continues until there is only one item left to check.

Items left to search	*Comparisons so far*
16	0
8	1
4	2
2	3
1	4

For a list of size 16, there are 4 comparisons to reach a list of size one, given that there is one comparison for each division, and each division splits the list size in half.

In general, if n is the size of the list and c is the number of comparisons, then

$$C = \log_2 n$$

$$\therefore \text{Eficiency in worst case} = O(\log n)$$

Average Case: In binary search, the average case efficiency is near to the worst case efficiency. So the average case efficiency will be taken as O(log n).

∴ Efficiency in average case = O (log n).

Binary Search	
Best Case	O(1)
Average Case	O(log n)
Worst Case	O(log n)

Space Complexity is O(n)

Quick Sort:

The quick sort is considered to be a fast method to sort the elements. It was developed by CAR Hoare. This method is based on divide-and-conquer technique i.e. the entire list is divided into various partitions and sorting is applied again and again on these partitions. This method is also called as partition exchange sorts.

The quick sort can be illustrated by the following example

12 6 18 4 9 8 2 15 |

The reduction step of the quick sort algorithm finds the final position of one of the numbers. In this example, we use the first number, 12, which is called the pivot (rotate) element. This is accomplished as follows-

Let 'I' be the position of the second element and 'j' be the position of the last element. i.e. I =2 and j =8, in this example.

Assume that a [n+1] =**∞**, where '**a**' is an array of size n.

[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	i	j
12	6	18	4	9	8	2	15	α	2	8

First scan the list from left to right (from I to j) can compare each and every element withthe pivot. This process continues until an element found which is greater than or equal to pivot element. If such an element found, then that element position becomes the value of 'i'.

Now scan the list from right to left (from j to i) and compare each and every element withthe pivot. This process continues until an element found which is less than or equal to pivot element. If such an element finds then that element's position become 'j' value.

Now compare 'i' and 'j'. If i <j, then swap a[i] and a[j]. Otherwise swap pivot element and a[j].

Continue the above process the entire list is sorted.

[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	i	j
12	6	18	4	9	8	2	15	α	2	8
12	6	18	4	9	8	2	15	α	3	7
12	6	2	4	9	8	18	15	α	7	6

Since i = 7 ≮ j=6, then swap pivot element and 6th element (j^{th} element), we get

8 6 2 4 9 12 18 15

Thus pivot reaches its original position. The elements on left to the right pivot are smallerthan pivot (12) and right to pivot are greater pivot (12).

8 6 2 4 9 12 18 15

Sublist 1 Sublist 2

Now take sub-list**1** and sub-list**2** and apply the above process recursively, at last we getsorted list.

Ex 2: Let the given list is-

8 18 56 34 9 92 6 2 64

[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	i	j
8	18	56	34	9	92	6	2	64	α	2	98
8	18	56	34	9	92	6	2	64	α	2	8
8	2	56	34	9	92	6	18	64	α	3	7
8	2	6	34	9	92	56	18	64	α	4	3

Since i $\nless$ j, then swap jth element, and pivot element, we get

6 2 8 34 9 92 56 18 64

$<$ $>$ $\longleftrightarrow$

Sublist 1 Sublist 2

Now take a sub-list that has more than one element and follow the same process as above. At last, we get the sorted list that is, we get

2 6 8 9 18 34 56 64 92

The following algorithm shows the quick sort algorithm-

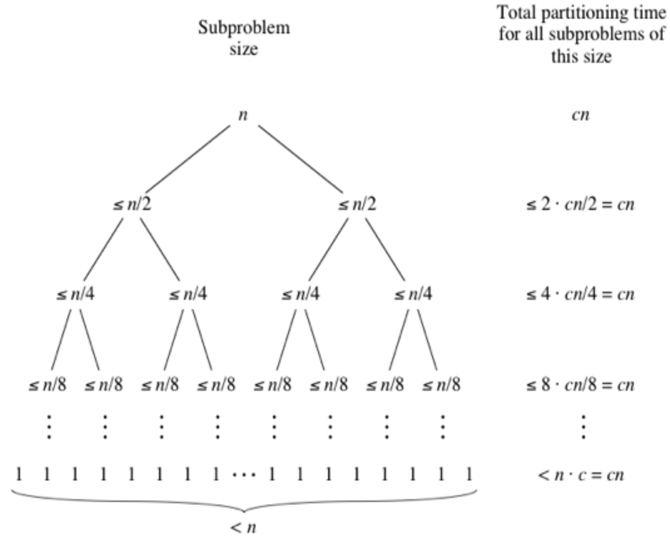
Algorithm Quicksort(i, j)

```
{
  // sorts the array from a[i] through a[j]
   If ( i <j) then     //if there are more than one element
    {
         //divide P into two sub-
                programsK: = partition
                (a, i, j+1);
                //Here K denotes the position of the partitioning element
                 //solve the sub
                 problemsQuicksort(i,
                 K-1); Quicksort(K=1,
                 j);
              // There is no need for combining solution
    }
 }
```

Algorithm Partition (a, left, right)

```
 {
    // The element from a[left] through a[right] are rearranged in such a manner that if initially
    // pivot =a[left] then after completion a[j]= pivot, then return. Here j is the position where
    // pivot partition the list into two partitions. Note that a[right]= ∞.
            pivot: a[left];
             i: left;
        j:=right;repeat
```

```
{
   repeat
     i: =i+1;
   until (a[i] ≥ pivot);
   repeat
     j: =j-1;
     until (a[j] < pivot);if(
        i<j) then Swap (a, i,
        j);
   }until (i ≥ j);
   a[left]: =
   a[j]; a[j]: =
   pivot; return
   j;
}
Algorithm Swap (a, i, j)
{
  //Example a[i] with
      a[j]temp:= a[i];
     a[i]: =
     a[j];
     a[j]:=
     temp;
}
```



Advantages of Quick-sort: Quick-sort is the fastest sorting method among all the sorting methods. But it is somewhat complex and little difficult to implement than other sorting methods.

Efficiency of Quick-sort: The efficiency of Quick-sort depends upon the selection of pivot element.

Best Case: In best case, consider the following two assumptions-

1. The pivot, which we choose, will always be swapped into the exactly the middle of thelist. And also consider pivot will have an equal number of elements both to its left andright.
2. The number of elements in the list is a power of 2 i.e. $n= 2^y$

Merge Sort:

Merge sort is based on divide-and-conquer technique. Merge sort method is a two phaseprocess-

1. Dividing
2. Merging

Dividing Phase: During the dividing phase, each time the given list of elements is divided into two parts. This division process continues until the list is small enough to divide.

Merging Phase: Merging is the process of combining two sorted lists, so that, the resultant list isalso the sorted one. Suppose A is a sorted list with **n** element and B is a sorted list with $\mathbf{n_2}$ elements. The operation that combines the elements of A and B into a single sorted list C with $n=n_1 + n_2$, elements is called merging.

Algorithm-(Divide algorithm)

Algorithm Divide (a, low, high)

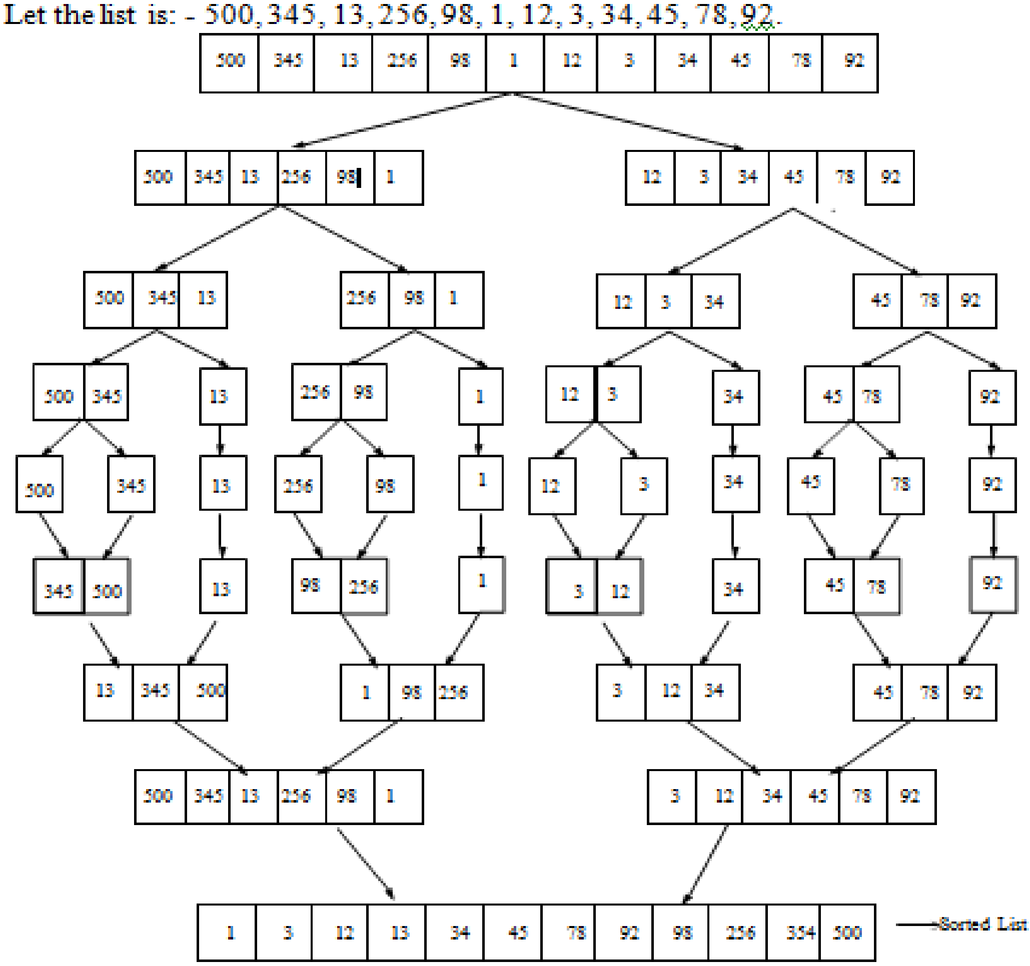
```
{
  // a is an array, low is the starting index and high is the end index of a

   If( low < high) then
     {
       Mid: = (low + high) /2;
       Divide( a, low, mid);
       Divide( a, mid +1,
       high);Merge(a, low,
       mid, high);
     }

}
```

The merging algorithm is as follows-

```
Algorithm Merge( a, low, mid, high)
  {
          L:= low;
           H:=
           high;
            J:= mid
           +1;K:=
           low;

           While (low ≤ mid AND j ≤  high) do
            {
                      If (a[low < a[j]) then
                       {
                         B[k] =
                         a[low];K:=
                         k+1;
                         Low:= low+1;
                       }
                       Else
                      {
                        B[k]=
                        a[j];K:
                        = k+1;
                        J: = j+1;
                      }
           }

          While (low ≤ mid) do
          {
             B[k]=a[lo
             w];K: =
             k+1;
```

```
            Low: =low + 1;
          }

        While (j ≤ high) do
        {
            B[k]=a[
            j]; K:  =
            k+1;   j:
            =j + 1;
          }

          //copy elements of b to a
          For i: = l to n do
          {
           A[i]: =b[i];
          }
}
```

Ex: Let the list is: - 500, 345, 13, 256, 98, 1, 12, 3, 34, 45, 78, 92.



The merge sort algorithm works as follows-

Step 1: If the length of the list is 0 or 1, then it is already sorted, otherwise,

Step 2: Divide the unsorted list into two sub-lists of about half the size.

Step 3: Again sub-divide the sub-list into two parts. This process continues until each element in the list becomes a single element.

Step 4: Apply merging to each sub-list and continue this process until we get one sorted list.

Efficiency of Merge List: Let 'n' be the size of the given list/ then the running time for merge sort is given by the recurrence relation.

$$T(n) = \begin{cases} a & \text{if } n=1, \text{ a is a constant} \\ 2T(n/2) + Cn & \text{if } n>1, \text{ C is constant} \end{cases}$$

Assume that 'n' is a power of 2 i.e. $n=2^k$.

This can be rewritten as $k=\log_2 n$.

$$\text{Let } T(n) = 2T(n/2) + Cn \quad \text{——— (1)}$$

We can solve this equation by using successive substitution.

Replace n by n/2 in equation, (1) we get

$$T(n/2) = 2T(n/4) + \frac{Cn}{2} \quad \text{——— (2)}$$

$$\text{Thus, } T(n) = 2\left(2T(n/4) + \frac{Cn}{2}\right) + Cn$$

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

$$= 4T\left(2T(n/8) + \frac{Cn}{4}\right) + 2Cn$$

...

...

...

$$= 2^k T(1) + KCn \quad \left(\because k = \log_2 n\right)$$

$$= a\,n + Cn \log n$$

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

Worst case	$O(n \log n)$
Best case	$O(n)$
Average case	$O(n \log n)$
Space Complexity	$O(n)$

Finding Maximum and Minimum Problem

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

Problem: Analyze the algorithm to find the maximum and minimum element from an array.

Algorithm: Max ?Min Element (a [])

```
Max:  a [i]
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]
return (max, min)
```

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.

```
Algorithm straight MaxMin (a, n, max, min)
// Set max to the maximum & min to the minimum of a [1: n]
{
Max = Min = a [1];
For i = 2 to n do
{
If (a [i] > Max) then Max = a [i];
If (a [i] < Min) then Min = a [i];
}}
```

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.

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.

Algorithm:

```
MaxMin (i, j, max, min)
 // a [1: n] is a global array, parameters i& j are integers, 1<= i<=j <=n. The effect is
to4.
// Set max & min to the largest & smallest value 5 in a [i: j], respectively.
{
If (i=j) then Max = Min = a[i];
Else if (i=j-1) then
{
if (a[i] < a[j]) then
          {
                    Max = a[j];
                    Min = a[i];
          }
          Else
          {
                    Max = a[i];
                    Min = a[j];
          }
}Else
{
Mid = (i + j) / 2;
MaxMin (I, Mid, Max, Min);
MaxMin (Mid +1, j, Max1, Min1);
If (Max < Max1) then Max = Max1;
If (Min > Min1) then Min = Min1;
}}
The procedure is initially invoked by the statement, MaxMin (1, n, x, y)
```

Example:

A	1	2	3	4	5	6	7	8	9
Values	22	13	-5	-8	15	60	17	31	47

Tree Diagram:

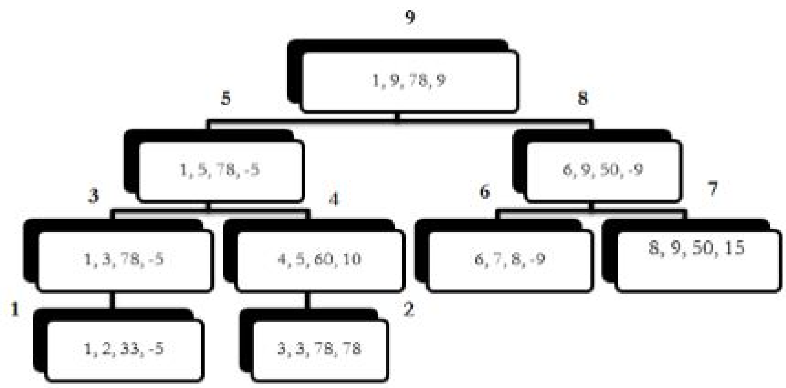


Figure 4

<u>Recurrence Equation for Divide and Conquer</u>

A divide-and-conquer algorithm consists of three steps:

- dividing a problem into smaller subproblems
- solving (recursively) each subproblem
- then combining solutions to subproblems to get solution to original problem

We use recurrences to analyze the running time of such algorithms. Suppose Tn is the number of steps in the worst case needed to solve the problem of size n. Let us split a problem into a ¥ 1 subproblems, each of which is of the input size nb

where b > 1.

Observe, that the number of subproblems a is not necessarily equal to b. The total number

of steps Tn is obtained by all steps needed to solve smaller subproblems Tnêb plus the number needed to combine solutions into a final one. The following equation is called divide-and-conquer recurrence relation

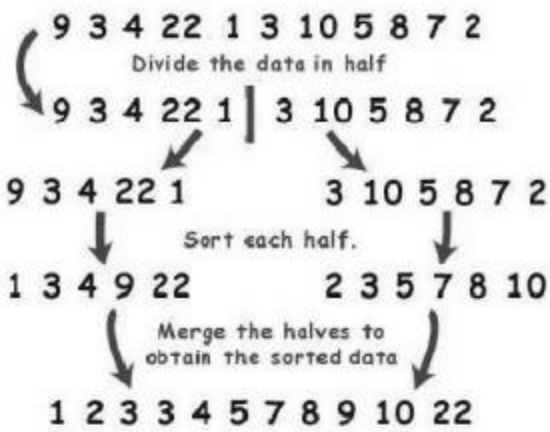
Tn = a Tnêb + f HnL

As an example, consider the mergesort:

-divide the input in half

-recursively sort the two halves

-combine the two sorted subsequences by merging them.



Let $T(n)$ be worst-case runtime on a sequence of n keys:

If $n = 1$, then $T(n) = \Theta(1)$ constant time

If $n > 1$, then $T(n) = 2\,T(n/2) + \Theta(n)$

here Θ(n) is time to do the merge. Then

Tn = 2 Tnê2 +QHnL

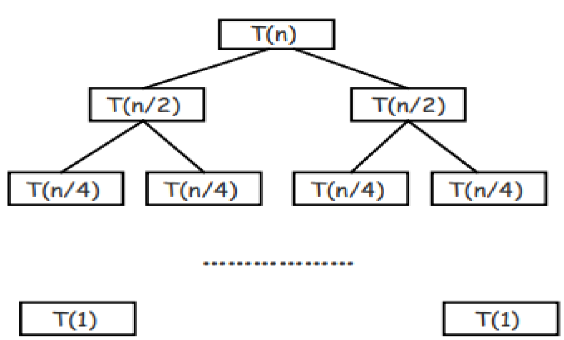
Other examples of divide and conquer algorithms: quicksort, integer multiplication, matrix multiplication, fast Fourier trnsform, finding conver hull and more.

There are several techniques of solving such recurrence equations:

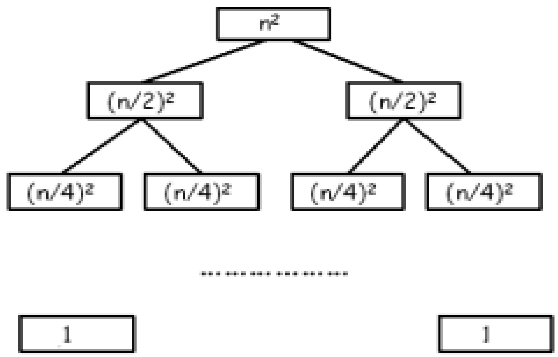
- the iteration method
- the tree method
- the master-theorem method
- guess-and-verify

Tree method

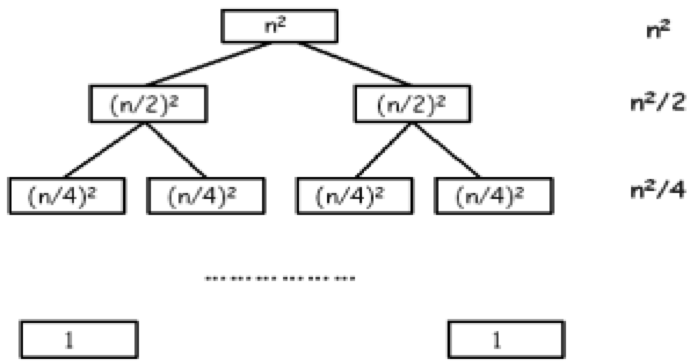
We could visualize the recursion as a tree, where each node represents a recursive call. The root is the initial call. Leaves correspond to the exit condition. We can often solve the recurrence by looking at the structure of the tree. To illustrate, we take this example THnL = 2 TK n 2 O +n 2 TH1L = 1 Here is a recursion tree that diagrams the recursive function calls



Using a recursion tree we can model the time of a recursive execution by writing the size of the problem in each node.



Using a recursion tree we can model the time of a recursive execution by writing the size of the problem in each node.



Selection Sort

The selection sort enhances the bubble sort by making only a single swap for each pass through the rundown. In order to do this, a selection sort searches for the biggest value as it makes a pass and, after finishing the pass, places it in the best possible area. Similarly, as with a bubble sort, after the first pass, the biggest item is in the right place. After the second pass, the following biggest is set up. This procedure proceeds and requires n-1 goes to sort n item since the last item must be set up after the (n-1) th pass.

ALGORITHM: SELECTION SORT (A)

```
1. k ← length [A]
2. for j ←1 to n-1
3. smallest ←  j
4. for I ← j + 1 to k
5. if A [i] < A [ smallest]
6. then smallest ←  i
7. exchange (A [j], A [smallest])
```

How Selection Sort works

In the selection sort, first of all, we set the initial element as a minimum.

Now we will compare the minimum with the second element. If the second element turns out to be smaller than the minimum, we will swap them, followed by assigning to a minimum to the third element.

Else if the second element is greater than the minimum, which is our first element, then we will do nothing and move on to the third element and then compare it with the minimum.

We will repeat this process until we reach the last element.

After the completion of each iteration, we will notice that our minimum has reached the start of the unsorted list.

For each iteration, we will start the indexing from the first element of the unsorted list. We will repeat the Steps from 1 to 4 until the list gets sorted or all the elements get correctly positioned.

Consider the following example of an unsorted array that we will sort with the help of the Selection Sort algorithm.

A [] = (7, 4, 3, 6, 5).

A [] =

1st Iteration:

Set minimum = 7

- Compare a_0 and a_1

As, $a_0 > a_1$, set minimum = 4.

- Compare a_1 and a_2

As, $a_1 > a_2$, set minimum = 3.

- Compare a_2 and a_3

As, $a_2 < a_3$, set minimum= 3.

Compare a_2 and a_4

Stassen‟s Matrix Multiplications

Consider two n x n matrices A and B

Recall that the matrix product C = AB of two n x n matrices is defined

as the n x n matrix that has the coefficient

ckl = ∑ akm ∑ ml

in row k and column l, where the sum ranges over the integers from 1 to n; the scalar product of the kth row of a with the lth column of B.

The straightforward algorithm uses O(n3) scalar operations.

Can we do better?

Idea: Use Divide and Conquer

The divide and conquer paradigm is important general technique

for designing algorithms. In general, it follows the steps:

- divide the problem into subproblems

- recursively solve the subproblems

- combine solutions to subproblems to get solution to original problem

Divide-and-Conquer

• Divide matrices A and B into four submatrices each

• We have 8 smaller matrix multiplications and 4 additions. Is it faster?

Let write the product A B = C as follow

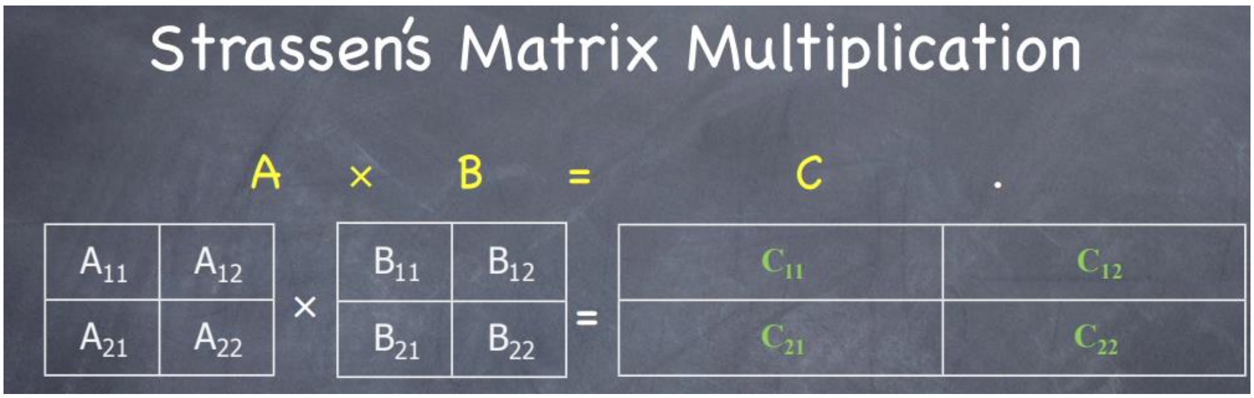
$$\begin{array}{|c|c|}\hline A_0 & A_1 \\ \hline A_2 & A_3 \\ \hline \end{array} \times \begin{array}{|c|c|}\hline B_0 & B_1 \\ \hline B_2 & B_3 \\ \hline \end{array} = \begin{array}{|c|c|}\hline A_0\times B_0+A_1\times B_2 & A_0\times B_1+A_1\times B_3 \\ \hline A_2\times B_0+A_3\times B_2 & A_2\times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

Divide matrices A and B into four submatrices each We have 8 smaller matrix multiplications and 4 additions. Is it faster?

Let us investigate this recursive version of the matrix multiplication. Since we divide A, B and C into 4 submatrices each, we can compute the resulting matrix C by

- 8 matrix multiplications on the submatrices of A and B,
- plus Θ(n2) scalar operations

Running time of recursive version of straightfoward algorithm is T(n) = 8T(n/2) + Θ(n2) and T(2) = Θ(1) where T(n) is running time on an n x n matrix • Master theorem gives us: ! ! T(n) = Θ(n3) • Can we do fewer recursive calls (fewer multiplications of the n/2 x n/2 submatrices)?



P1 = (A11+ A22)(B11+B22)

P2 = (A21 + A22) * B11

P3 = A11 * (B12 - B22)

P4 = A22 * (B21 - B11)

P5 = (A11 + A12) * B22

P6 = (A21 - A11) * (B11 + B12)

P7 = (A12 - A22) * (B21 + B22)

C11 = P1 + P4 - P5 + P7

C12 = P3 + P5

C21 = P2 + P4

C22 = P1 + P3 - P2 + P6