Chapter -2 Divide and Conquer

- ✓ **General Method ,Binary Search, finding the min and max**
- ✓ **Merge Sort analysis.**
- ✓ **Quick Sort performance measurement**
- ✓ **Randomized version of quick sort and analysis.**

In computer science, **divide and conquer** (**D&C**) is an important algorithm design paradigm based on multi-branched recursion. A divide and conquer algorithm works by recursively breaking down a problem into two or more sub-problems of the same (or related) type, until these become simple enough to be solved directly. The solutions to the sub-problems are then combined to give a solution to the original problem. It is a top-down technique.

Divide-and-conquer paradigm consists of following major phases:

- Breaking the problem into several sub-problems that are similar to the original problem but smaller in size,
- Solve the sub-problem recursively (successively and independently), and then
- Combine these solutions to subproblems to create a solution to the original problem.

General Form

T (n) = aT (n/b) + g(n)
n: size of original problem.
T(i): time to solve problem of size i.
a: number of subproblems.
b: size of each subproblem.
g(n): time to divide and combine subproblems

Binary Search : (Simplest application of divide-and-conquer)

Binary Search is an extremely well-known instance of divide-and-conquer paradigm. Given an ordered array of n elements, the basic idea of binary search is that for a given element we "probe" the middle element of the array. We continue in either the lower or upper segment of the array, depending on the outcome of the probe until we reached the required (given) element.

Find an element in a sorted array:
1.Divide:Check middle element.
2. Conquer: Recursively search 1subarray.
3.Combine:Trivial.

Example: Find 9

3 5 7 **8** 9 12 15 //Given Set --- middle element is 8

3 5 7 **8** 9 12 15 //Take upper limit

3 5 7 **8** 9 **12** 15 //Find middle element -12

3 5 7 **8** **9** **12** 15 //Take lower limit –only one element –compare 9

Algorithm

BINSRCH(A,n,x,j)

Given an array A(1:n) of elements in non decreasing order
n>=0 ,determine if x is present ,and if so set j such that x=A(j) else j=0

```
integer  low ,high,mid,j,n;
low ←1; high ←n;
while  low<=high  do
      mid←[(low+high)/2]
      case
            x<A(mid) : high ←mid-1
            x>A(mid):low← mid+1
           else
           j ←mid return
       end case
repeat
 j ← 0
end BINSRCH
```

Analysis

- **Space Requirement :-** It required n elements of array plus storage for the variable low, mid, high, x, j or n+5.
- **Time requirement :-**
  Comparisons per successful search :

  Elements = > 3 5 7 8 9 12 15

  Comparisons := > 3 2 3 1 3 2 3

  Avg Comparisons : 17/7 =>2.42

  Comparison per unsuccessful search : For given problem at any location it will require 3 comparison so 3+3+3+3+3+3+3 =21/7 =>3

Divide Step: Compute m; takes constant time.
Number of subproblem to solve is 1.

No combining required.
Recurrence: T(n) = T(n/2) + c.

Binary Search: Recurrence Solving
Basic Recurrence: T(n) = T(n/2) + c.
End case: T(1) = 1.
T(n) = T(n/2) + c
= (T(n/2^2) + c) + c
= T(n/2^2) + 2c
= (T(n/2^3) + c) + 2c
= T(n/2^3) + 3c
...
= T(n/2^k) + kc
Solving n/2^k = 1 gives k = log n.
We have the solution: T(n) = 1 + c log n:
Or
The complexity of such an algorithm –the work required to complete it given a list of size n initially –is the number of probes that are required. That is, how many times can a list of size n be halved before it becomes of size one? Let k be the smallest integer such that
n / 2^k = 1
Then k = ceiling (log n). Of course, sometimes the target is found before k probes. It has been proved that the average time for a successful search by this method is approximately log n –1 .
So binary search is O(log n)

Finding The Maximum and Minimum

Simple Algorithm :

```
Integer i,n;
Max ← min ←A(1)
for i ← 2 to n do
    if  A(i) >  max then
        max ← A(i) end if
    if A(i) <min then
        min← A(i) end if
repeat
```

Analysis :
Procedure is easy .It requires 2(n-1) comparison.
if A(i) > max then max ← A(i)
else if A(i) <min then min← A(i) end if
By making above change now it requires n-1 comparison.
Divide and conquer Approach :

Divide the problem into smaller instance. Find the max and min for smaller instance and compare the max and min of different instance.

Algorithm

```
Integer I,j; global n,A(1:n)
Case
     I=j  :fmax← fmin← A(i)
     I=j-1 ; if A(i) <A(j)  then fmax ←A(j) ; fmin ← A(i)
                            Else fmax ← A(i) ; fmin ← A(j)
             End if
Else
       Mid← [(i+j)/2]
       Call  MAXMIN (I,mid,gmax ,gmin )
       Call MAXMIN (mid+1,j,hmax ,hmin)
       fmax ← max(gmax ,hmax)
       fmin ←min(gmin,hmin)
end case
```

Analysis :

_ Space Complexity:

Storage requirement of MAXMIN is worse than simple algorithm. It requires stack space for I, j , fmax, fmin. Given n element we require $\log_2 n+1$

Time Complexity :

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

The best ,worst and avg case number of comparison O(n).

Note : Recursive way of finding maximum and minimum is slow than simple as it requires more space (stack).

Merge Sort

Merge-sort is a sorting algorithm based on the divide-and-conquer paradigm

Merge-sort on an input sequence ***S*** with ***n*** elements consists of three steps:

 Divide: partition ***S*** into two sequences ***S***1 and ***S***2of about ***n***/2 elements each

 Recur: recursively sort ***S***1 and ***S***2

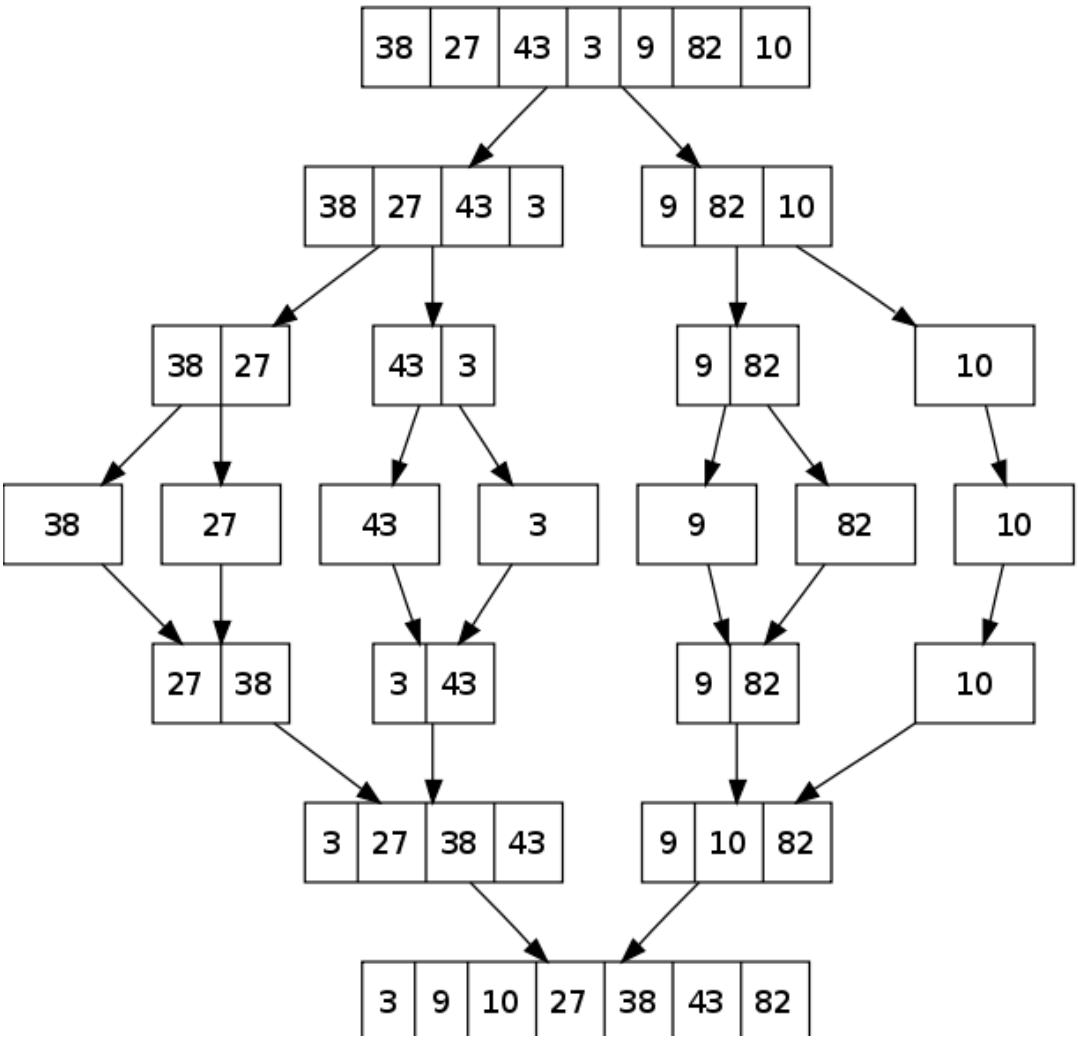
 Conquer: merge ***S***1 and ***S***2 into a unique sorted sequence

 Like heap-sort It uses a comparator It has ***O***(***n*** log ***n***) running time

 Unlike heap-sort It does not use an auxiliary priority queue

 It accesses data in a sequential manner (suitable to sort data on a disk)

Example



Algorithm

MERGESORT(low, high)

```
Integer low, high;
If low <high  then
  Mid← [(low+high)/2]
   Call MERGESORT (low ,mid)       //sort one subset
   Call MERGESORT(mid+1,high)  //Sort other subset
   Call MERGE (low, mid, high)       //Combine the result
End if
```

MERGE(low, mid, high)

```
Integer h,I,j,k,low ,mid,high
global  A(low:high); local B(low,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
 endif
repeat

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;
                  Repeat
End if
For k← low to high do
 A(k) ← B(k)
Repeat
End
```

Analysis

Space Complexity

It requires 2n storage space .One for auxiliary array.

Time complexity:

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

T(1) = 1

T(n)= 2 times running time on a list of n/2 elements + linear merge

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

There are k=log n equations to get to T (1): (when n is power of 2 , $n=2^k$)

T(n) = nT(1)+nlog n = nlog n + n

Quick Sort sorts by employing a divide and conquer strategy to divide a list into two sub-lists.

The steps are:

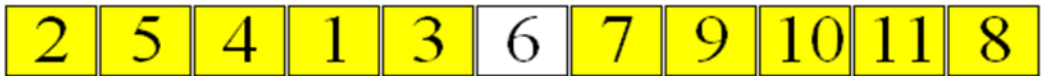
1. Pick an element, called a *pivot*, from the list.
2. Reorder the list so that all elements which are less than the pivot come before the pivot and so that all elements greater than the pivot come after it (equal values can go either way). After this partitioning, the pivot is in its final position.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

The base case of the recursion are lists of size zero or one, which are always sorted.

- Small instance has n <= 1. Every small instance is a sorted instance.
- To sort a large instance, select a pivot element from out of the n elements.
- Partition the n elements into 3 groups left, middle and right.
- The middle group contains only the pivot element.
- All elements in the left group are <= pivot.
- All elements in the right group are >= pivot.
- Sort left and right groups recursively.
- Answer is sorted left group, followed by middle group followed by sorted right group.

Example

Use 6 as the pivot.



Sort left and right groups' recursively.

Procedure QUICKSORT (p,q)

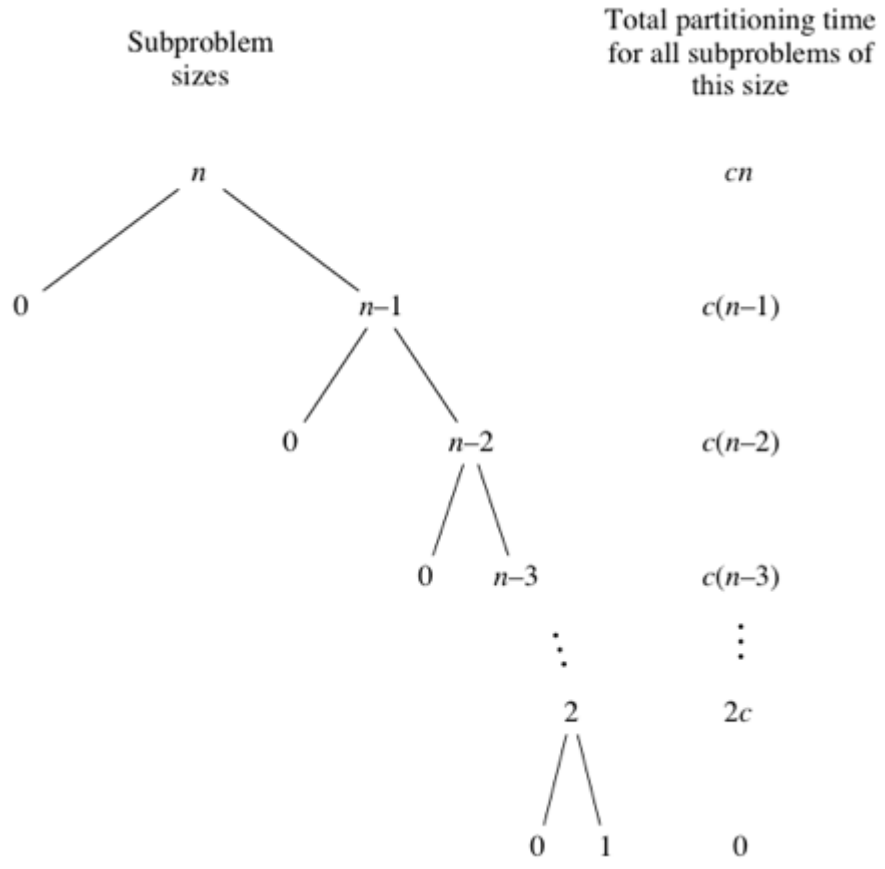
//sorts elements A(p),……A(q) which reside in the global array A(1:n) into ascending order.A(n+1) is considered to be defined and must be >= all elements in A(p:q), A(n+1)=+infinity

```
Integer p,q;global n,A(1:n)
If  P<q  then  j← q+1
         Call PARTITION(p,j)
        CALL QUICKSORT( p,j-1)
        CALL QUICKSORT( j+1,q)
Endif
End QUICKSORT
```

PROCDEURE PARTITION(m,p)
//within A(m),A(m+1)…a(p-1) the elements are rearranged in such a way that if initially t= A(m) then after completion a(q)=t for some q between m and p-1.
Integer m,p,I;global A(m:p)
v←A(m) ;i←m //A(,) s the partition element
loop
loop i←I+ 1 until A(i)>=v repeat //I moves from left to right
loop P←p- 1 until A(p)<=v repeat //p moves from right to left
if i <p then **call INTERCHANGE(A(i),A(P))** //exchange
else exit
repeat
A(m)←A(p);
A(p)← v
End PARTITION

Worst-case running time

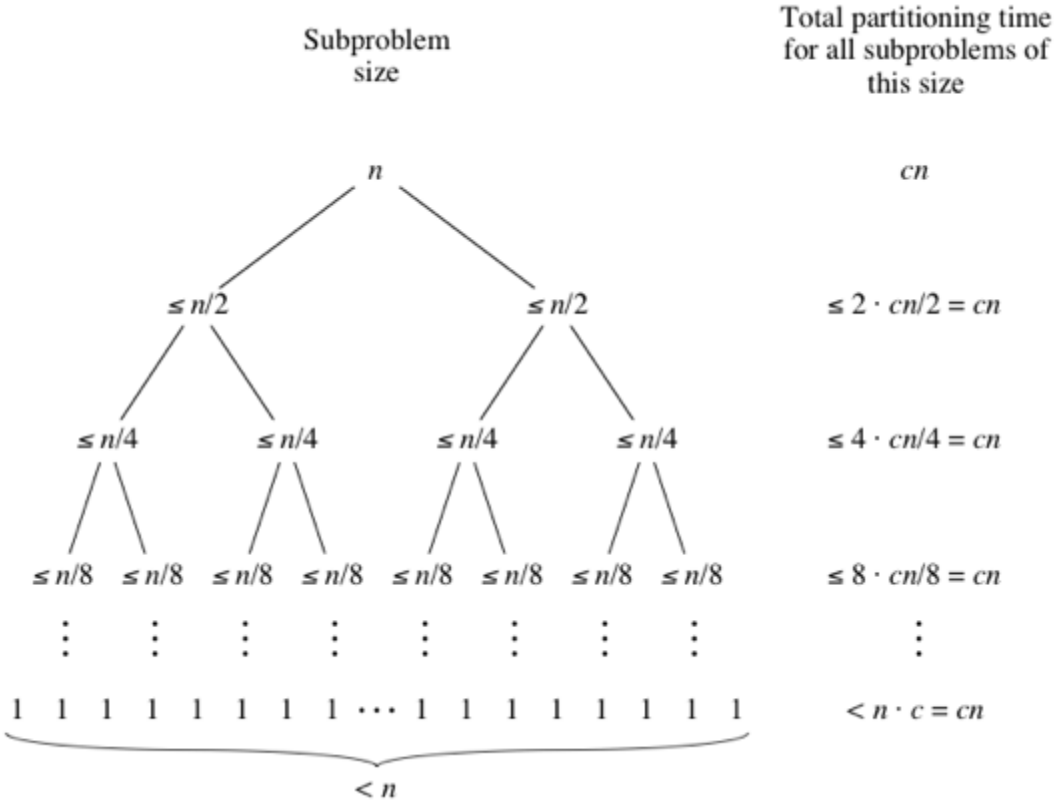
When quicksort always has the most unbalanced partitions possible, then the original call takes n time for some constant , the recursive call on n-1takes c($n-1$)time , the recursive call on n-2c(n-2)time . Here's a tree of the sub problem sizes with their partitioning times:



Quicksort's worst-case running time is $0(n^2)$.

Best-case and Average case running time

Quicksort's best case occurs when the partitions are as evenly balanced as possible: their sizes either are equal or are within 1 of each other. The former case occurs if the subarray has an odd number of elements and the pivot is right in the middle after partitioning, and each partition has (n-1)/2 left parenthesis.



We get the same result as for Quick sort best case Ω (nlogn) and average case complexity Θ(nlogn) .