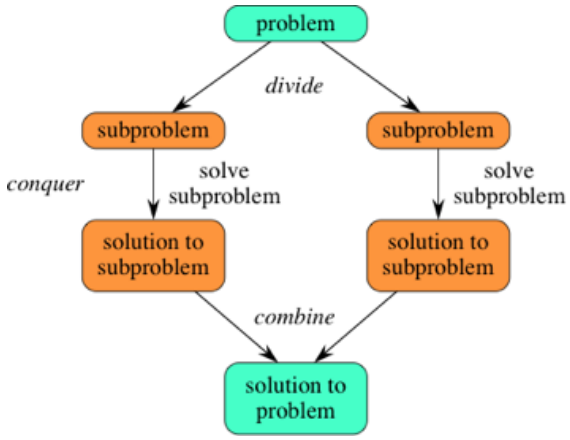
Divide and Conquer (DAC)Technique

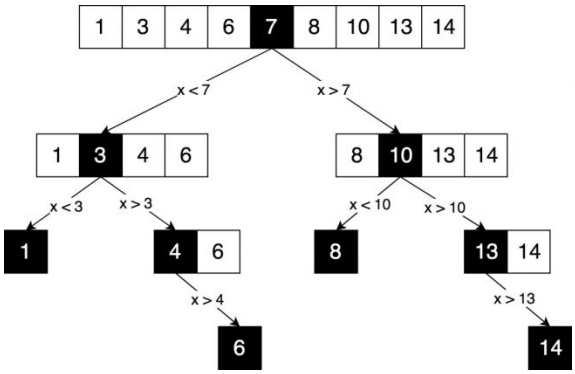
The name \divide and conquer" has been given to a powerful algorithm design technique that is used to solve a variety of problems. In its simplest form, a divide-and-conquer algorithm divides the problem instance into a number of sub instances (in most 3 cases), *recursively*

- Divide: the instance ***I*** into ***p*** sub instances $I_1, I_2, \ldots I_p$ of approximately the same size.
- Conquer: Recursively call the algorithm on each sub instance I_j, $1 < j < p$, to obtain p partial solutions.
- Combine the results of the ***p*** partial solutions to obtain the solution to the original instance ***I***. Return the solution of instance ***I***.



Flowchart (1) Divide and Conquer (DAC)

To illustrate this approach, consider the problem of finding both the minimum and maximum in an array of integers *A*[1..*n*] and assume for simplicity that *n* is a power of 2.



Example Divide and Conquer (DAC)

The following computer algorithms are based on divide-and-conquer programming approach

- Binary Search
- Merge Sort
- Quick Sort
- Strassen's Matrix Multiplication
- Closest pair (points)

Binary Search: Binary search is a fast search algorithm with run-time complexity of O(log n). This search algorithm works on the principle of divide and conquers. For this algorithm to work properly, the data collection should be in the sorted form. Binary search looks for a particular item by comparing the middle most item of the collection.

- If a match occurs, then the index of item is returned.
- If the middle item is greater than the item, then the item is searched in the sub-array to the left of the middle item.
- Otherwise, the item is searched for in the sub-array to the right of the middle item.

This process continues on the sub-array as well until the size of the sub-array reduces to zero.

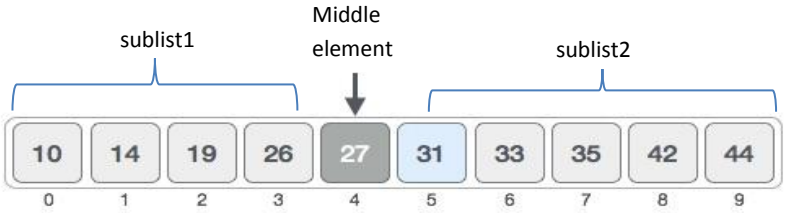
Algorithm binarysearch
Input: An array A[1..n] of n elements sorted in no decreasing order and an element x.
Output: j if x = A[j]; 1 ≤ j ≤ n; and 0 otherwise.
1. binarysearch (1,n)
Procedure binarysearch (low, high)
1. if low > high then return 0
2. else
3. mid← ⌊(low + high)/2⌋
4. if x = A[mid] then return mid
5. else if x < A[mid] then return binary search(low; mid - 1)
6. else return binarysearch(mid + 1; high)
7. end if

Example 1: The following is our sorted array; we need to search the location of value 31 using binary search.

Step1: Find the middle element of the array by using this formula −

```
mid = (high + low) / 2
```

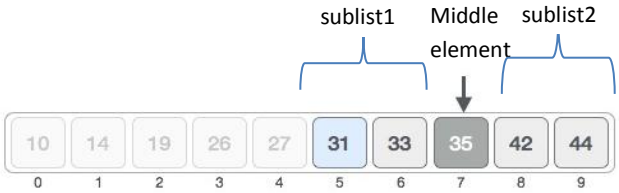
Here it is, (9 + 0) / 2 = 4 (integer value of 4.5). So, 4 is the mid of the array.



Step2: Compare the value stored at location 4, with the value being searched, i.e. 31. We find that it the value at location 4 is 27, which is not a match.

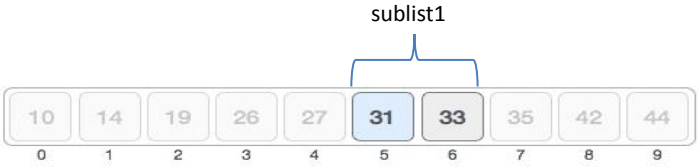
if 31> 27 search the sublist 2

Now handle only sublist2. Again divide it, find mid of sublist 2, here it is, (9 + 5) / 2 = 7. So, 7 is the mid of the array. We find that it the value at location 7 is 35, which is not a match.



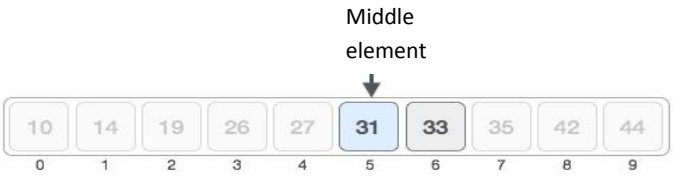
if 31< 35 search the sublist 1

Now handle only sublist1. Again divide it, find mid of sublist 1, here it is, (6 + 5) / 2 = 5. So, 5 is the mid of the array.



if middle=key

We find that it the value at location 5 is 31, so match is found at 5th position of array i.e. at array [5].



Example 2: Using tables to find key by binary search.

Data[]=15,-6,0,7,9,23,54,82,101

Elements that searching; x=101, x=82, x=-14? Assignment (deadline at 29-4-2019//8:00 AM)

Low	Mid	high	Key
1	5	9	9
6	7	9	54
8	8	9	82
9	9	9	101

Found 101 at 9th position of array

Low	Mid	high	Key
1	5	9	9
6	7	9	54
8	8	9	82

Found 82 at 8th position of array

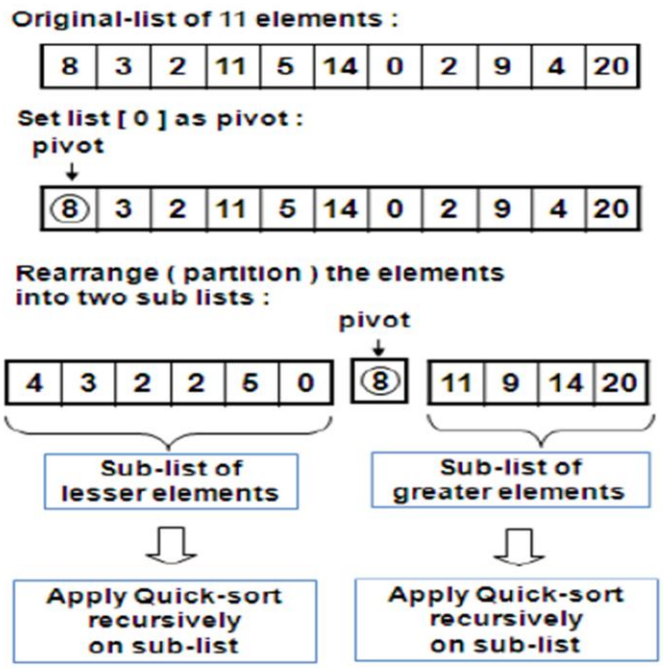
Quick Sort(QS): A quick sort first selects a value, which is called the **pivot value**. Although there are many different ways to choose the pivot value, we will simply use the first item in the list. The role of the pivot value is to assist with splitting the list. The actual position where the pivot 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. Using pivot algorithm recursively, we end up with smaller possible partitions. Each partition is then processed for quick sort. We define recursive algorithm for quicksort as follows –

Step 1 *– Make the right-most index value pivot*
Step 2 *– partition the array using pivot value*
Step 3 *– quicksort left partition recursively*
Step 4 *– quicksort right partition recursively*

Quick Sort Pseudo code: To get more into it, let see the pseudo code for quick sort algorithm –

```
procedure quickSort(left, right)
   if right-left <= 0
      return
   else
      pivot = A[right]
      partition = partitionFunc(left, right, pivot)
      quickSort(left,partition-1)
      quickSort(partition+1,right)
   end if
end procedure
```

Example: - A list of unsorted elements are: 8 3 2 11 5 14 0 2 9 4 20



Merge Sort Algorithm

Merge sort is one of the most efficient sorting algorithms. It works on the principle of Divide and Conquer. Merge sort repeatedly breaks down a list into several sublist until each sublist consists of a single element and merging those sublist in a manner that results into a sorted list.

Basic Idea: It divides the array in two equal parts (based on the median). Each set is individually sorted and the resulting sorted sequences are merged to get a single sorted sequence. Runs in O (n*log n) time in all the cases.

//Algorithm of Merge Sort

MERGE-SORT(A, p, r)

1. If p < r
2. q = [(p + q) /2]
3. MERGE-SORT(A, p, q)
4. MERGE-SORT(A, q+1, r)
5. MERGE(A, p, q, r)

MERGE (A, p, q, r)

1. $n_1 = q – p +1$
2. $n_2 = r – q$
3. let L [1.. $n_1 + 1$] and L [1.. $n_2 + 1$] be new arrays
4. for i=1 to n_1
5. L[i] = A [p + i -1]
6. for j=1 to n_2
7. R[j]= A[q + j]
8. L [$n_1 + 1$] = ∞
9. R [$n_2 + 1$] = ∞
10. i = 1
11. j = 1
12. for k = p to r
13. if L[i] ≤ R [j]
14. A[k] = L[i]
15. i = i + 1
16. else A[k] = R [j]
17. j = j + 1

Example: Let’s consider an array with values {14, 7, 3, 12, 9, 11, 6, 12} Below, we have a pictorial representation of how merge sort will sort the given array. In merge sort we follow the following steps:

1. We take a variable p and store the starting index of our array in this. And we take another variable r and store the last index of array in it.
2. Then we find the middle of the array using the formula (p + r)/2 and mark the middle index as q, and break the array into two subarrays, from p to q and from q + 1 to r index.
3. Then we divide these 2 subarrays again, just like we divided our main array and this continues.

4. Once we have divided the main array into subarrays with single elements, then we start merging the subarrays.