Data Structures - Divide and Conquer

To understand the divide and conquer design strategy of algorithms, let us use a simple real world example. Consider an instance where we need to brush a type C curly hair and remove all the knots from it. To do that, the first step is to section the hair in smaller strands to make the combing easier than combing the hair altogether. The same technique is applied on algorithms.

Divide and conquer approach breaks down a problem into multiple sub-problems recursively until it cannot be divided further. These sub-problems are solved first and the solutions are merged to form the final solution.

The standard procedure for the divide and conquer design technique is as follows −

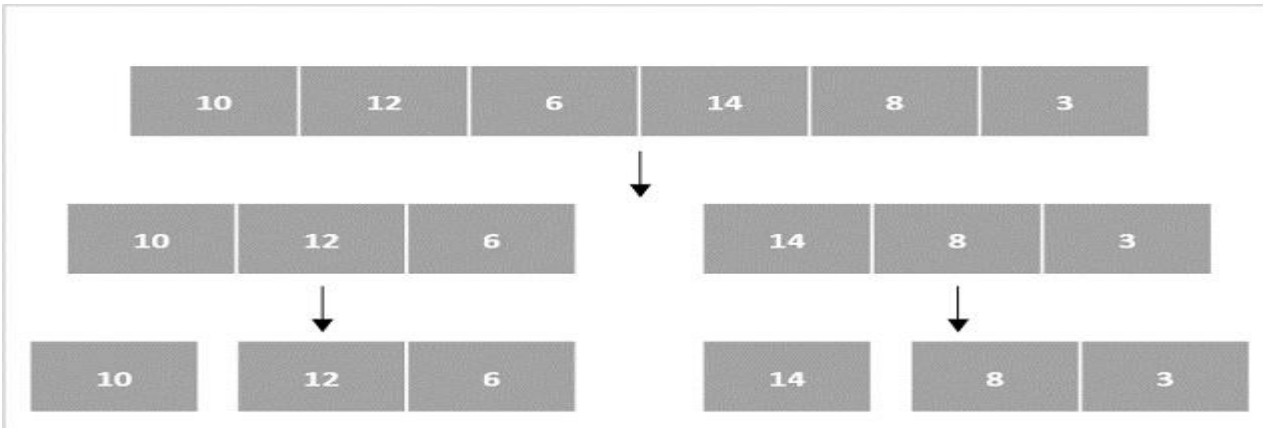
- **Divide** − We divide the original problem into multiple sub-problems until they cannot be divided further.
- **Conquer** − Then these subproblems are solved separately with the help of recursion
- **Combine** − Once solved, all the subproblems are merged/combined together to form the final solution of the original problem.

There are several ways to give input to the divide and conquer algorithm design pattern. Two major data structures used are − **arrays** and **linked lists**. Their usage is explained as

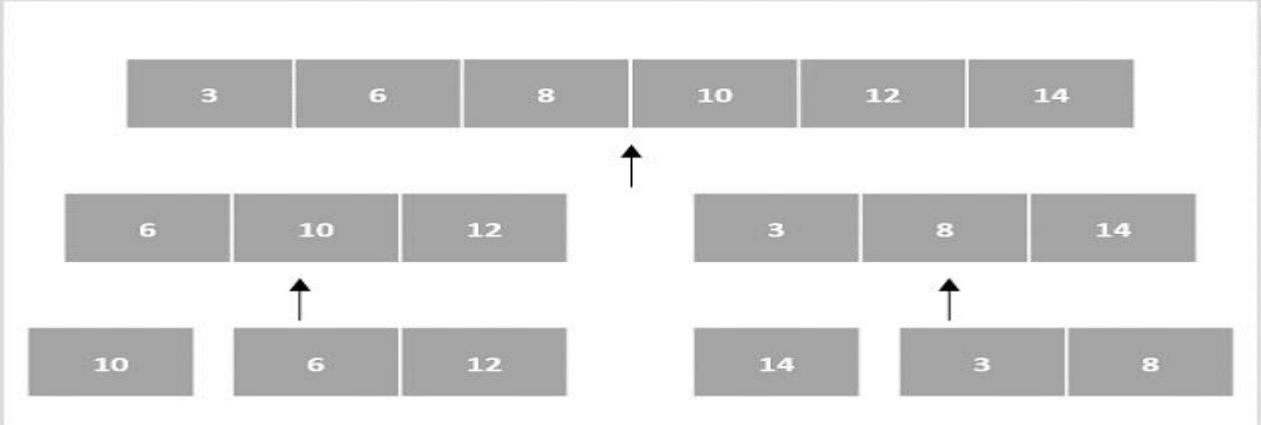
Arrays as Input

There are various ways in which various algorithms can take input such that they can be solved using the divide and conquer technique. Arrays are one of them. In algorithms that require input to be in the form of a list, like various sorting algorithms, array data structures are most commonly used.

In the input for a sorting algorithm below, the array input is divided into subproblems until they cannot be divided further.



Then, the subproblems are sorted (the conquer step) and are merged to form the solution of the original array back (the combine step).

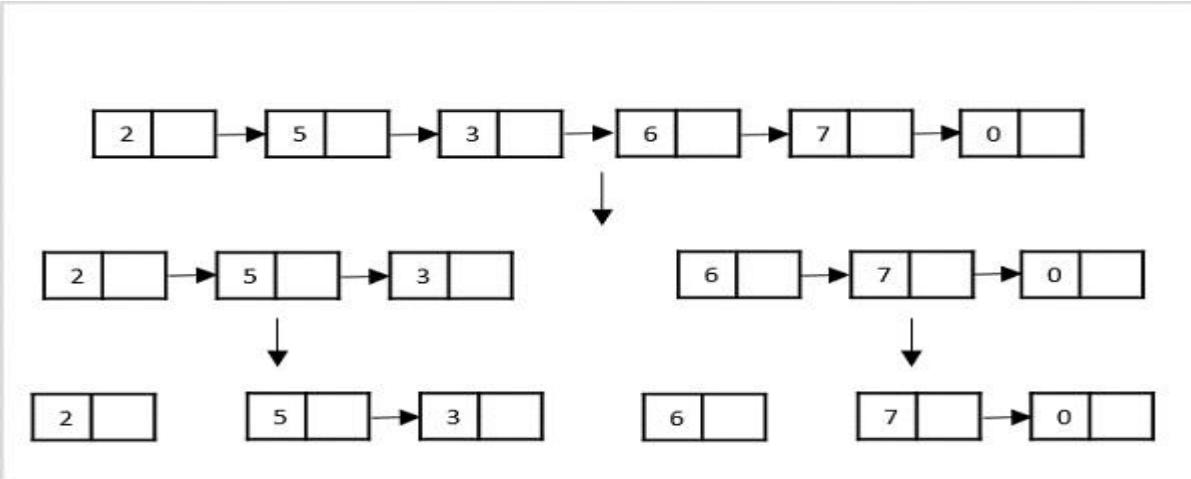


Since arrays are indexed and linear data structures, sorting algorithms most popularly use array data structures to receive input.

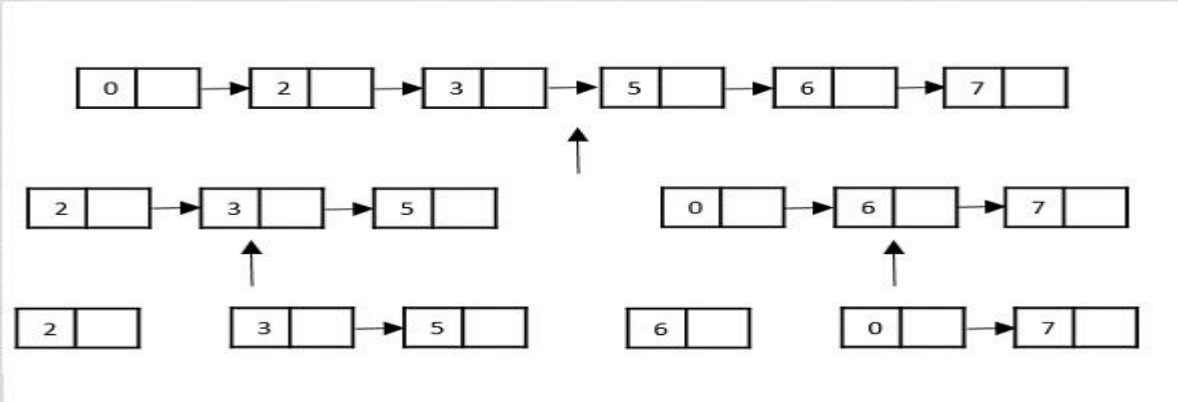
Linked Lists as Input

Another data structure that can be used to take input for divide and conquer algorithms is a linked list (for example, merge sort using linked lists). Like arrays, linked lists are also linear data structures that store data sequentially.

Consider the merge sort algorithm on linked list; following the very popular tortoise and hare algorithm, the list is divided until it cannot be divided further.



Then, the nodes in the list are sorted (conquered). These nodes are then combined (or merged) in recursively until the final solution is achieved.



Various searching algorithms can also be performed on the linked list data structures with a slightly different technique as linked lists are not indexed linear data structures. They must be handled using the pointers available in the nodes of the list.

Examples

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

- Merge Sort
- Quick Sort
- Binary Search
- Strassen's Matrix Multiplication
- Closest Pair

There are various ways available to solve any computer problem, but the mentioned are a good example of divide and conquer approach.

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 conquer. For this algorithm to work properly, the data collection should be in a sorted form.

Implementation in C

```
#include <stdio.h>

#define MAX 20

// array of items on which linear search will be conducted.
int intArray[MAX] = {1,2,3,4,6,7,9,11,12,14,15,16,17,19,33,34,43,45,55,66};

void printline(int count) {
   int i;

   for(i = 0;i <count-1;i++) {
      printf("=");
   }
```

```
   printf("=\n");
}

int find(int data) {
   int lowerBound = 0;
   int upperBound = MAX -1;
   int midPoint = -1;
   int comparisons = 0;
   int index = -1;

   while(lowerBound <= upperBound) {
      printf("Comparison %d\n" , (comparisons +1) );
      printf("lowerBound : %d, intArray[%d] = %d\n",lowerBound,lowerBound,
         intArray[lowerBound]);
      printf("upperBound : %d, intArray[%d] = %d\n",upperBound,upperBound,
         intArray[upperBound]);
      comparisons++;

      // compute the mid point
      // midPoint = (lowerBound + upperBound) / 2;
      midPoint = lowerBound + (upperBound - lowerBound) / 2;

      // data found
      if(intArray[midPoint] == data) {
         index = midPoint;
         break;
      } else {
         // if data is larger
         if(intArray[midPoint] < data) {
            // data is in upper half
            lowerBound = midPoint + 1;
         }
         // data is smaller
         else {
            // data is in lower half
            upperBound = midPoint -1;
         }
      }
   }
   printf("Total comparisons made: %d" , comparisons);
   return index;
}

void display() {
   int i;
   printf("[");

   // navigate through all items
   for(i = 0;i<MAX;i++) {
      printf("%d ",intArray[i]);
   }

   printf("]\n");
}

void main() {
   printf("Input Array: ");
   display();
   printline(50);

   //find location of 1
```

```
   int location = find(55);

   // if element was found
   if(location != -1)
      printf("\nElement found at location: %d" ,(location+1));
   else
      printf("\nElement not found.");
}
```

Output

If we compile and run the above program then it would produce following result −

```
Input Array: [1 2 3 4 6 7 9 11 12 14 15 16 17 19 33 34 43 45 55 66 ]
==================================================
Comparison 1
lowerBound : 0, intArray[0] = 1
upperBound : 19, intArray[19] = 66
Comparison 2
lowerBound : 10, intArray[10] = 15
upperBound : 19, intArray[19] = 66
Comparison 3
lowerBound : 15, intArray[15] = 34
upperBound : 19, intArray[19] = 66
Comparison 4
lowerBound : 18, intArray[18] = 55
upperBound : 19, intArray[19] = 66
Total comparisons made: 4
Element found at location: 19
```

Sorting

Sorting is the process of arranging the elements either in ascending (or) descending order.

Types of sorting

C language provides five sorting techniques, which are as follows −

- Bubble sort (or) Exchange Sort
- Selection sort
- Insertion sort (or) Linear sort
- Quick sort (or) Partition exchange sort
- Merge Sort (or) External sort

Quick sort

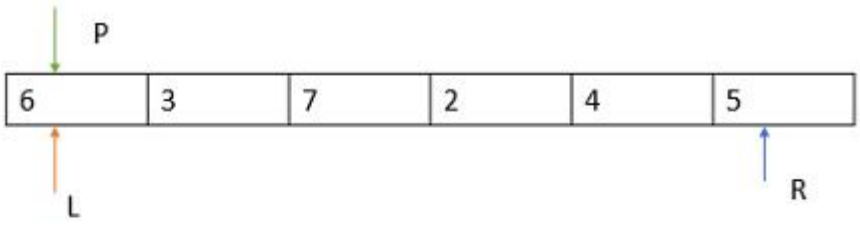
It is a divide and conquer algorithm.

- Step 1 − Pick an element from an array, call it as pivot element.
- Step 2 − Divide an unsorted array element into two arrays.
- Step 3 − If the value less than pivot element come under first sub array, the remaining elements with value greater than pivot come in second sub array.

Consider an example given below, wherein

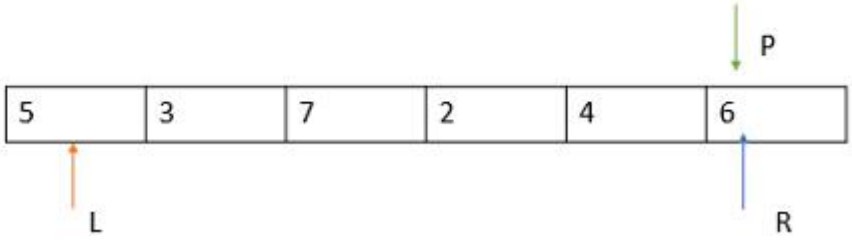
- P is the pivot element.
- L is the left pointer.
- R is the right pointer.

The elements are 6, 3, 7, 2, 4, 5.



Case 1: P=6 {Right side P is greater and Left side of is Leese}

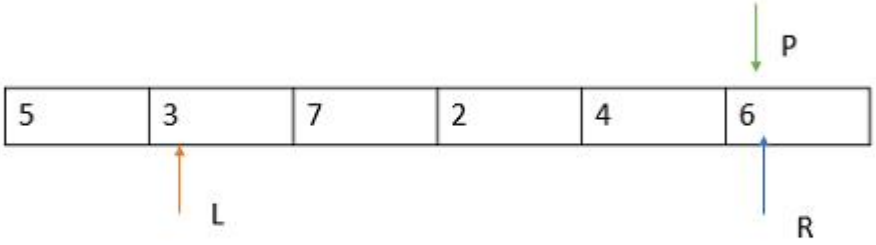
Is P<R => 6<5 {wrong}
So, swap P with R



Case 2: P=6 , L=5 {Right side P is greater and Left side of is Lesser}

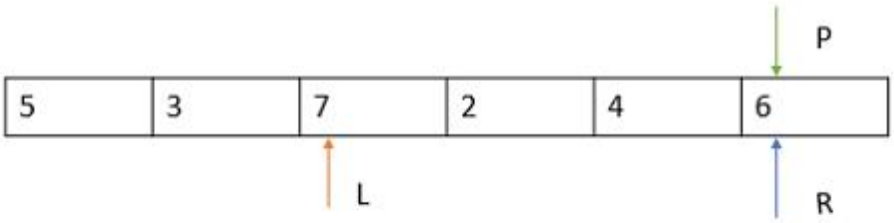
Is P>L => 6> 5 {right}
Move L towards right

Case 3:



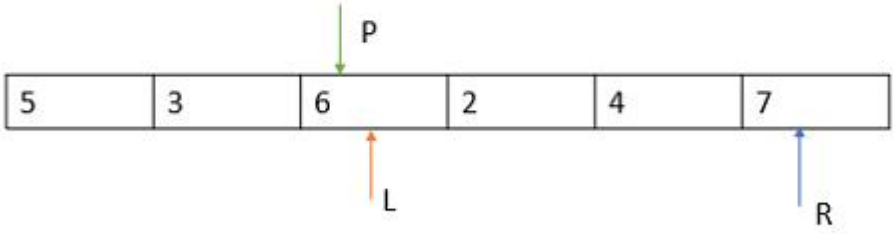
Is P>L, 6>3, Yes
So, move L towards right

Case 4:



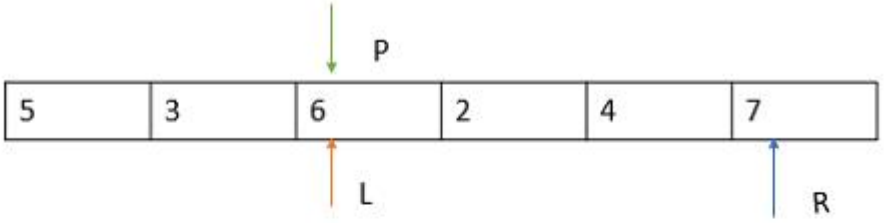
Is P>L => 6> 7 {wrong}
then swap P and L

Case5 :



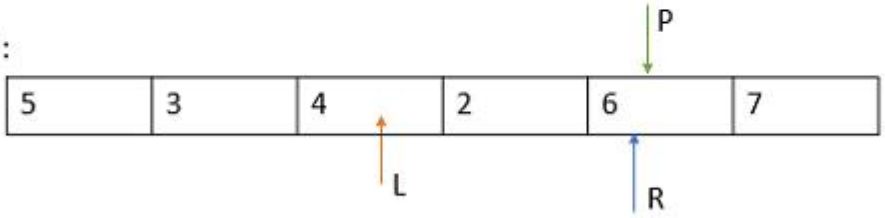
Is P<R => 6<7, => Yes
Decrement R

Case 6:



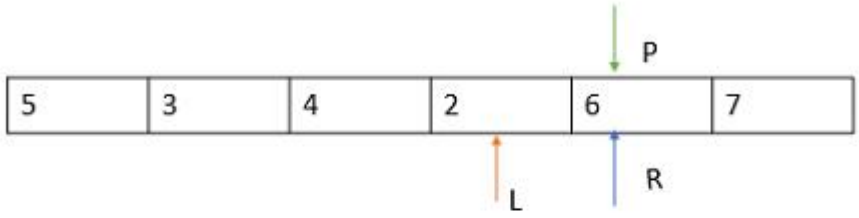
Is P<R => 6<4 {wrong}
then swap

Case 7 :



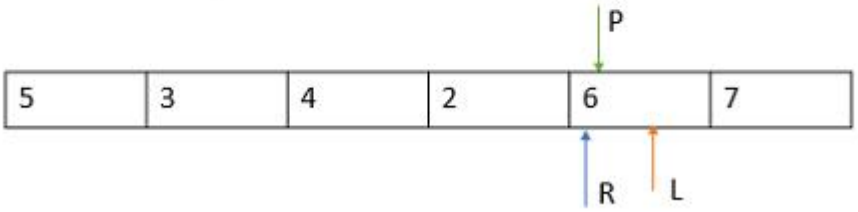
Is P>L => 6>4, => Yes
Move L to right

Case 8:



Is P>L => 6>2 {right}
move L to right

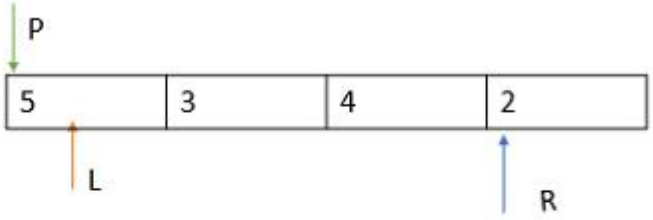
Case 9 :



Now,

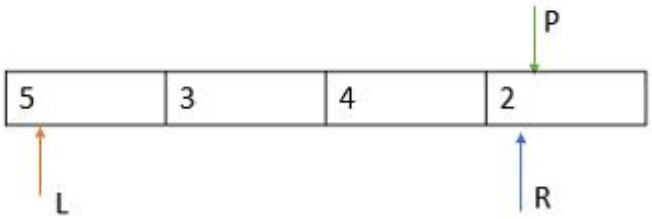
- The pivot is in fixed position.
- All the left elements are less.
- The right elements are greater than pivot.
- Now, divide the array into 2 sub arrays left part and right part.
- Take left partition apply quick sort.

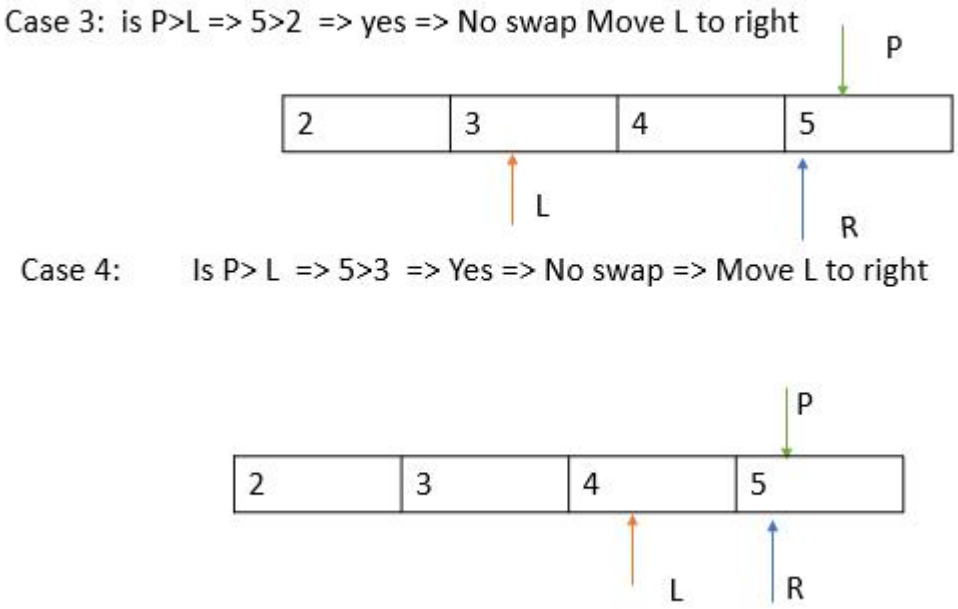
Case 1:



Is P<R => 5<2 {wrong} so, swap

Case 2 :





Now,

- The pivot is in fixed position.
- All the left elements are less and sorted
- The right elements are greater and are in sorted order.
- The final sorted list is combining two sub arrays is 2, 3, 4, 5, 6, 7

Example

Following is the C program to sort the elements by using the quick sort technique −

```
#include<stdio.h>
void quicksort(int number[25],int first,int last){
   int i, j, pivot, temp;
   if(first<last){
      pivot=first;
      i=first;
      j=last;
      while(i<j){
         while(number[i]<=number[pivot]&&i<last)
         i++;
         while(number[j]>number[pivot])
         j--;
         if(i<j){
            temp=number[i];
            number[i]=number[j];
            number[j]=temp;
         }
      }
      temp=number[pivot];
      number[pivot]=number[j];
      number[j]=temp;
      quicksort(number,first,j-1);
```

```
      quicksort(number,j+1,last);
   }
}
int main(){
   int i, count, number[25];
   printf("How many elements are u going to enter?: ");
   scanf("%d",&count);
   printf("Enter %d elements: ", count);
   for(i=0;i<count;i++)
   scanf("%d",&number[i]);
   quicksort(number,0,count-1);
   printf("Order of Sorted elements: ");
   for(i=0;i<count;i++)
   printf(" %d",number[i]);
   return 0;
}
```

Output

When the above program is executed, it produces the following output −

```
How many elements are u going to enter?: 10
Enter 10 elements: 2 3 5 7 1 9 3 8 0 4
Order of Sorted elements: 0 1 2 3 3 4 5 7 8 9
```