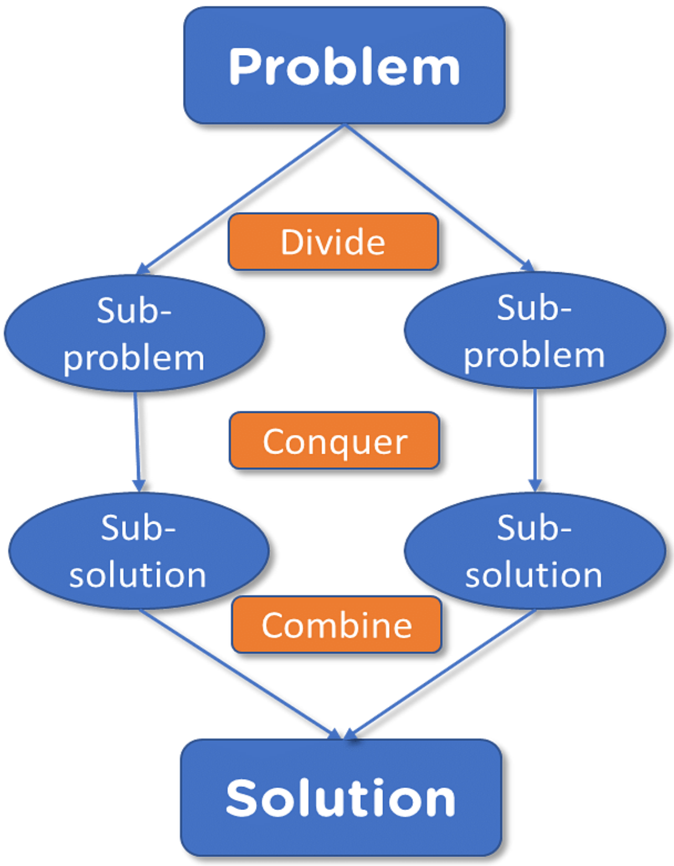
Divide and Conquer Algorithms

Divide and Conquer Algorithm breaks a complex problem into smaller subproblems, solves them independently, and then combines their solutions to address the original problem effectively.

A divide-and-conquer algorithm has three parts:

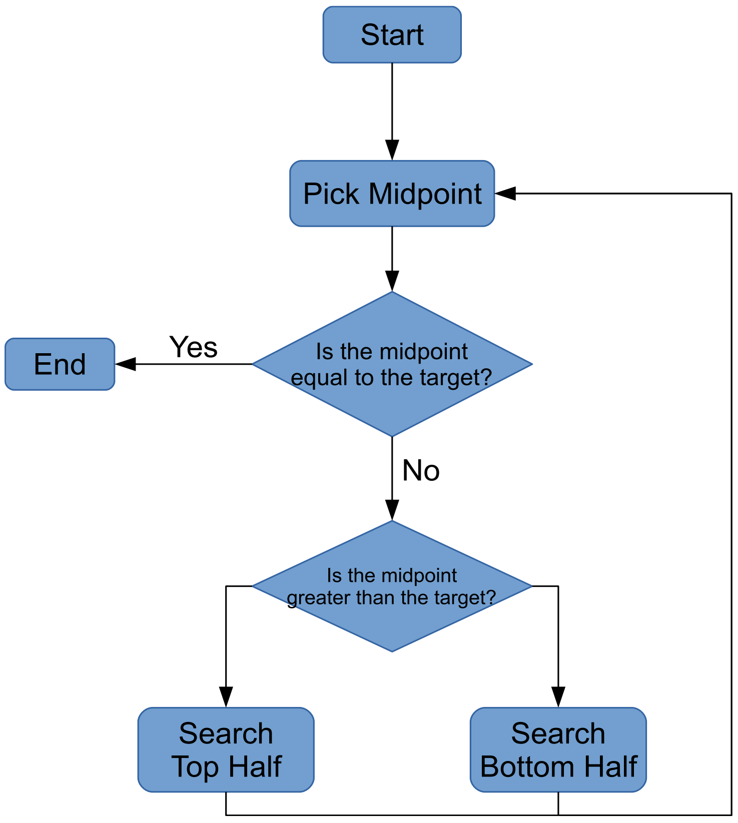
- Divide up the problem into a lot of smaller pieces of the same problem.
- Conquer the subproblems by recursively solving them. Solve the subproblems as base cases if they're small enough.
- To find the solutions to the original problem, combine the solutions of the subproblems.



Binary Search Algorithm:

Binary searches are efficient algorithms based on the concept of “divide and conquer” that improves the search by recursively dividing the array in half until you either find the element or the list gets narrowed down to one piece that doesn’t match the needed element.
Binary searches work under the principle of using the sorted information in the array to reduce the time complexity to zero (Log n). Here are the binary search approach’s basic steps:

- Begin with an interval that covers the entire array
- If the search key value is less than the middle-interval item, narrow the interval to that lower half. Otherwise, narrow the interval to the upper half.
- Keep checking the chosen interval until either the value is found or the interval’s empty


Start
Pick Midpoint
Is the midpoint equal to the target?
Yes
End
No
Is the midpoint greater than the target?
Search Top Half
Search Bottom Half

Implementation of a Binary Search

There are two forms of binary search implementation: Iterative and Recursive Methods. The most significant difference between the two methods is the Recursive Method has an O(logN) space complexity, while the Iterative Method uses O(1). So, although the recursive version is easier to implement, the iterative approach is more efficient.

- Iterative algorithms repeat a specific statement set a given number of times. The algorithms uses looping statements (e.g., loop, while loop, do-while loop) to repeat the same steps several times.
- On the other hand, recursive algorithms rely on a function that calls itself repeatedly until it reaches the base condition (also called the stopping condition).

Here Is a Sample of Iterative Algorithm Coding

```
class BinarySearch
{
public static int binarySearchIterative(int[] list, int first, int last, int key)
{
    while(first <= last)
    {
        int middle = first + (last - first) / 2;
          if(list[middle] ==  key)
        {
           return middle;
        }
        else if(list[middle] < key)
        {
           first = middle + 1;
        }
        else
        {
           last  = middle - 1;
        }

    }
    return -1;
}
public static void main(String[] args)
{
    int[] list = {10, 14, 19, 26, 27, 31, 33, 35, 42, 44};
    int element = 31;
    int result = binarySearchIterative(list, 0, list.length - 1, element);
    if(result == -1)
    {
          System.out.println("The element does not exist in the list");
    }
    else
    {
          System.out.println("The element found at index : " + result);
    }
}
}
```

Here Is a Sample of Recursive Algorithm Coding

```
class BinarySearch
{
public static int binarySearchRecursive(int[] list, int first, int last, int key)
{
    if(first >= last)
    {
        return -1;
    }
    int middle = first + (last - first) / 2;
    if(list[middle] == key)
    {
       return middle;
    }
    else if(list[middle] < key)
    {
        first = middle + 1;
        return binarySearchRecursive(list, first, last, key);
    }
    else
    {
        last  = middle - 1;
       return binarySearchRecursive(list, first, last, key);
    }
}
public static void main(String[] args)
{
    int[] list = {10, 14, 19, 26, 27, 31, 33, 35, 42, 44};
    int element = 31;
    int result = binarySearchRecursive(list, 0, list.length - 1, element);
    if(result == -1)
    {
          System.out.println("The element does not exist in the list");
    }
    else
    {
          System.out.println("The element found at index : " + result);
    }
}
}
```

Here is one everyday binary search example: Dictionaries
So, you somehow find yourself without Internet access, and you need to look up the definition of the word “wombat.” That means behaving like our primitive ancestors would and reaching for an actual physical dictionary! If you wanted to do a linear search, you would start at the “A” words and work your way through the dictionary until you got to “wombat.” Good luck with that!
However, most of us are cleverer than that, and we instinctively employ the binary search method. We consult the “W” listings and go to the middle of that section. If “wombat” is alphabetically smaller than the word on that middle page, we ignore the rest of the pages on the right side. If “wombat” is larger, then we ignore the left-hand pages. We then keep repeating the process until we find the word.

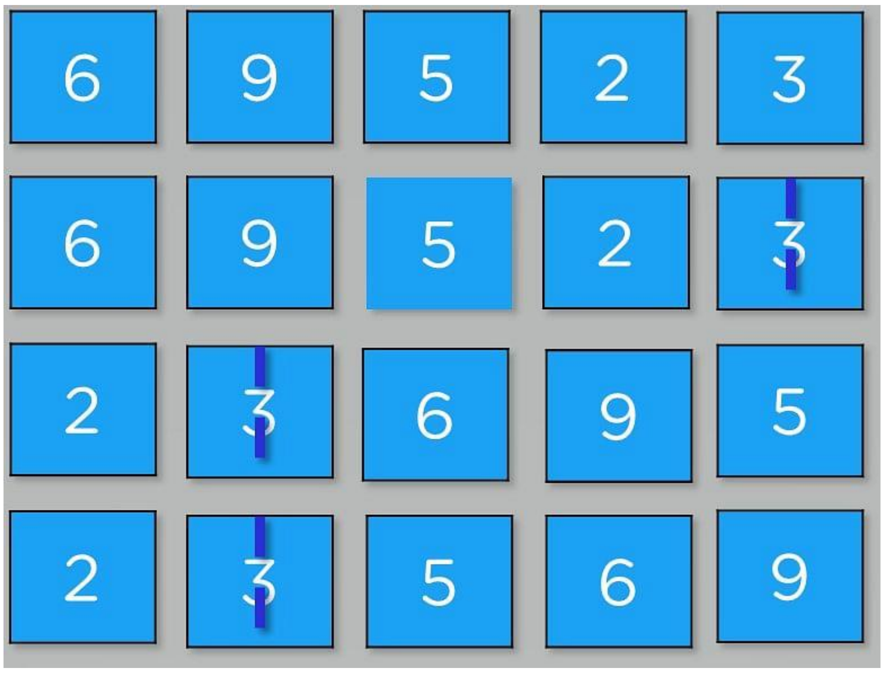
What is Quick Sort Algorithm?

Quicksort is a highly efficient sorting technique that divides a large data array into smaller ones. A vast array is divided into two arrays, one containing values smaller than the provided value, say pivot, on which the partition is based. The other contains values greater than the pivot value.

How Does Quick Sort Work?

To sort an array, you will follow the steps below:

1. You will make any index value in the array as a pivot.
2. Then you will partition the array according to the pivot.
3. Then you will recursively quicksort the left partition
4. After that, you will recursively quicksort the correct partition.



Let's have a closer look at the partition bit of this algorithm:

1. You will pick any pivot, let's say the highest index value.
2. You will take two variables to point left and right of the list, excluding pivot.
3. The left will point to the lower index, and the right will point to the higher index.
4. Now you will move all elements which are greater than pivot to the right.
5. Then you will move all elements smaller than the pivot to the left partition.

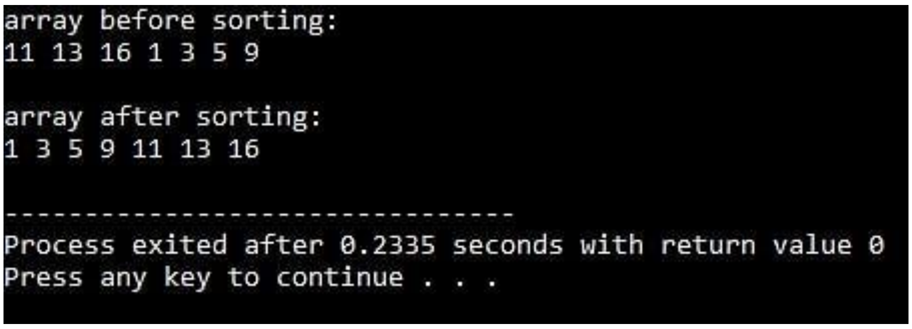
How to Implement the Quick Sort Algorithm?

You will be provided with an array of elements {10, 7, 8, 9, 1, 5}. You have to write a code to sort this array using the QuickSort algorithm. The final array should come out to be as {1, 5, 7, 8, 9, 10}.

Code:

```
// C++ implementation of QuickSort
#include <bits/stdc++.h>
using namespace std;
```

```
// A utility function to swap two elements
void swap(int* a, int* b)
{
int t = *a;
*a = *b;
*b = t;
}
/* This function takes the final pivot element, puts the pivot element in an ordered array, and places all smaller elements on the left side of the pivot, as well as all larger elements on the right of the pivot. */
int partition (int arr[], int l, int h)
{
int pivot = arr[h]; // pivot
int i = (l - 1); // Index of smaller element and indicates the right position of pivot found so far
for (int k = l; k <= h - 1; k++)
{
// When the actual element is less than the pivot
if (arr[k] < pivot)
{
i++; // increment index of smaller element
swap(&arr[i], &arr[k]);
}
}
swap(&arr[i + 1], &arr[h]);
return (i + 1);
}
//A function to implement quicksort
void quickSort(int arr[], int l, int h)
{
if (l < h)
{
//pi is a partitioning index, and
//arr[p] is now in the correct location.
int pi = partition(arr, l, h);
// Separately sort elements before
// partition and after partition
quickSort(arr, l, pi - 1);
quickSort(arr, pi + 1, h);
}
}
/* Function to print an array */
void print_array(int arr[], int size)
{
int i;
for (i = 0; i < size; i++)
cout << arr[i] << " ";
cout << endl;
}
int main()
{
int arr[] = {11, 13, 16, 1, 3, 5, 9};
int n = sizeof(arr) / sizeof(arr[0]);
quickSort(arr, 0, n - 1);
cout << "Sorted array: \n";
printArray(arr, n);
return 0;
}
```



```
array before sorting:
11 13 16 1 3 5 9

array after sorting:
1 3 5 9 11 13 16

--------------------------------
Process exited after 0.2335 seconds with return value 0
Press any key to continue . . .
```

Quicksort Complexity

Quicksort is popular for its impressive average-case performance, making it a popular choice for sorting large datasets. It is a powerful sorting algorithm with a favorable average-case time complexity and versatile applications in various fields of computer science. Understanding its complexity and having code implementations in multiple programming languages can be valuable for students and professionals, allowing them to leverage Quicksort's efficiency in their projects and applications. To understand its complexity, we'll examine both time and space complexities.

a. Time Complexity

1. Average Case: O(n*log(n))
2. Worst Case: O(n^2)
3. Best Case: O(n*log(n))

b. Space Complexity

Quicksort has a space complexity of O(log(n)) in the average case. This arises from the recursive function calls and the partitioning process. It can be O(n) due to an unbalanced partitioning leading to a deep recursion stack in the worst case.

Quicksort Applications

1. Sorting Algorithms: Quicksort is frequently used as a building block for hybrid sorting algorithms, such as Timsort (used in Python's built-in sorting function).

2. Database Systems: Quicksort plays a vital role in database management systems for sorting records efficiently.
3. Computer Graphics: Rendering and graphics applications often involve sorting operations, where Quicksort can be employed to optimize rendering performance.
4. Network Routing: Quicksort can be utilized in various networking algorithms, particularly routing tables.
5. File Systems: File systems use Quicksort to manage and organize files efficiently.

Quicksort code in C

```
#include <stdio.h>
void swap(int* a, int* b) {
   int t = *a;
   *a = *b;
   *b = t;
}
int partition(int arr[], int low, int high) {
   int pivot = arr[high];
   int i = (low - 1);
   for (int j = low; j <= high - 1; j++) {
      if (arr[j] < pivot) {
         i++;
         swap(&arr[i], &arr[j]);
      }
   }
   swap(&arr[i + 1], &arr[high]);
   return (i + 1);
}
void quicksort(int arr[], int low, int high) {
   if (low < high) {
      int pivotIndex = partition(arr, low, high);
      quicksort(arr, low, pivotIndex - 1);
      quicksort(arr, pivotIndex + 1, high);
   }
}
int main() {
   int arr[] = {3, 6, 8, 10, 1, 2, 1};
   int n = sizeof(arr) / sizeof(arr[0]);
   quicksort(arr, 0, n - 1);
   for (int i = 0; i < n; i++) {
      printf("%d ", arr[i]);
   }
   return 0;
}
```

What Are the Advantages of Quick Sort?

- It is an in-place algorithm since it just requires a modest auxiliary stack.
- Sorting n objects takes only n (log n) time.
- Its inner loop is relatively short.
- After a thorough mathematical investigation of this algorithm, you can make a reasonably specific statement about performance issues.

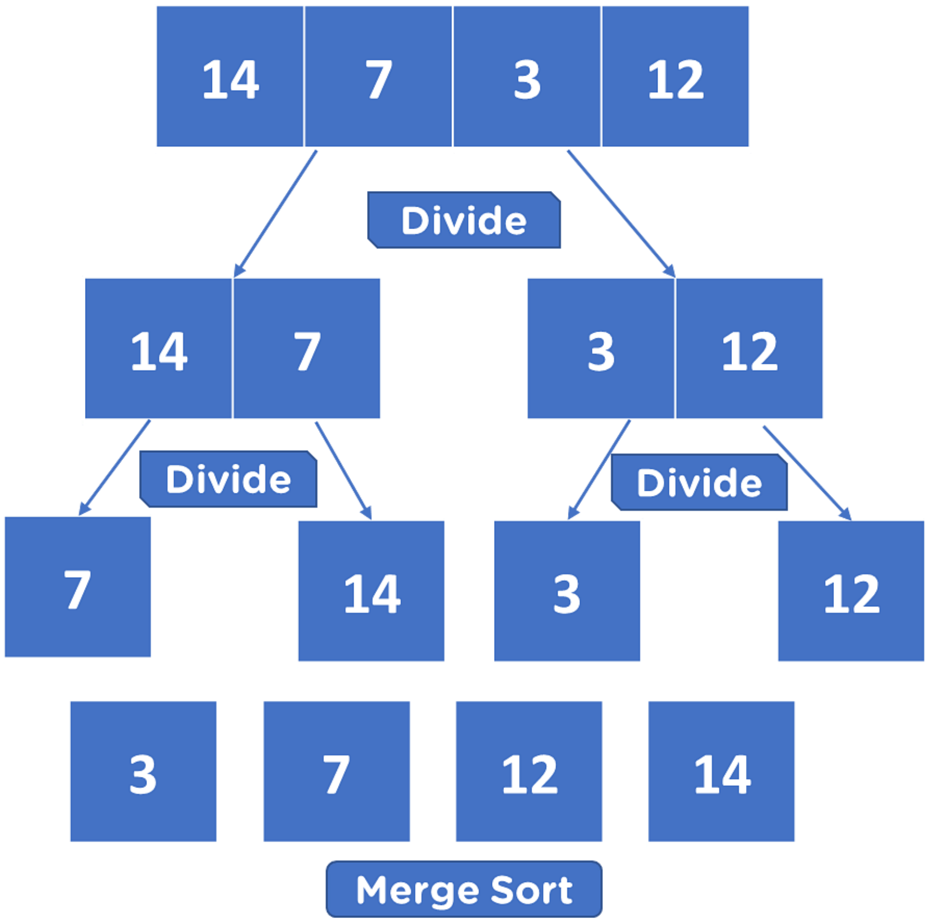
What Are the Disadvantages of Quick Sort?

- It is a recursive process. The implementation is quite tricky, mainly if recursion is not provided.
- In the worst-case scenario, it takes quadratic (i.e., n2) time.
- It is fragile in the sense that a slight error in implementation can go unreported and cause it to function poorly.

What is Merge Sort Algorithm?

The "Merge Sort" uses a recursive algorithm to achieve its results. The divide-and-conquer algorithm breaks down a big problem into smaller, more manageable pieces that look like the initial problem. It then solves these subproblems recursively and puts their solutions together to solve the original problem.

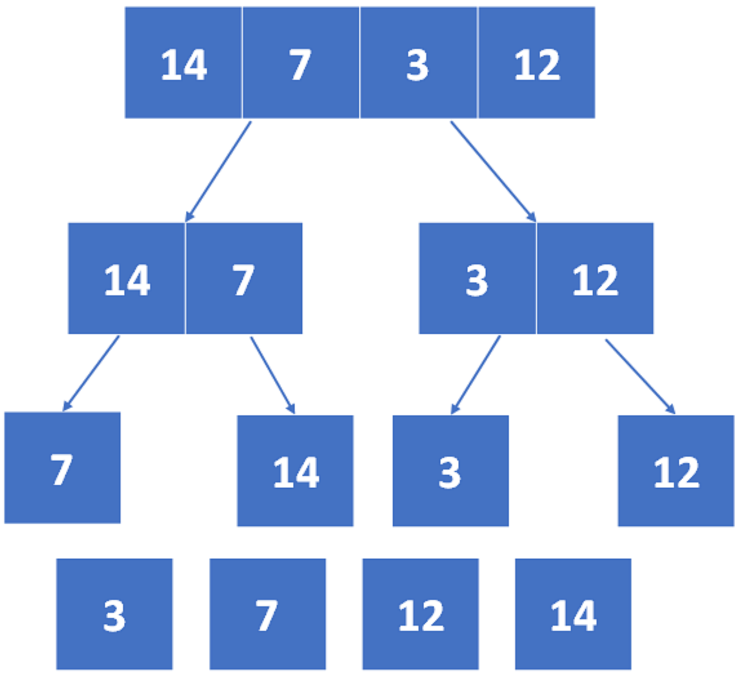
Merge sort is one of the most efficient sorting algorithms. It is based on the divide-and-conquer strategy. Merge sort continuously cuts down a list into multiple sublists until each has only one item, then merges those sub lists into a sorted list.



How Does the Merge Sort Algorithm Work?

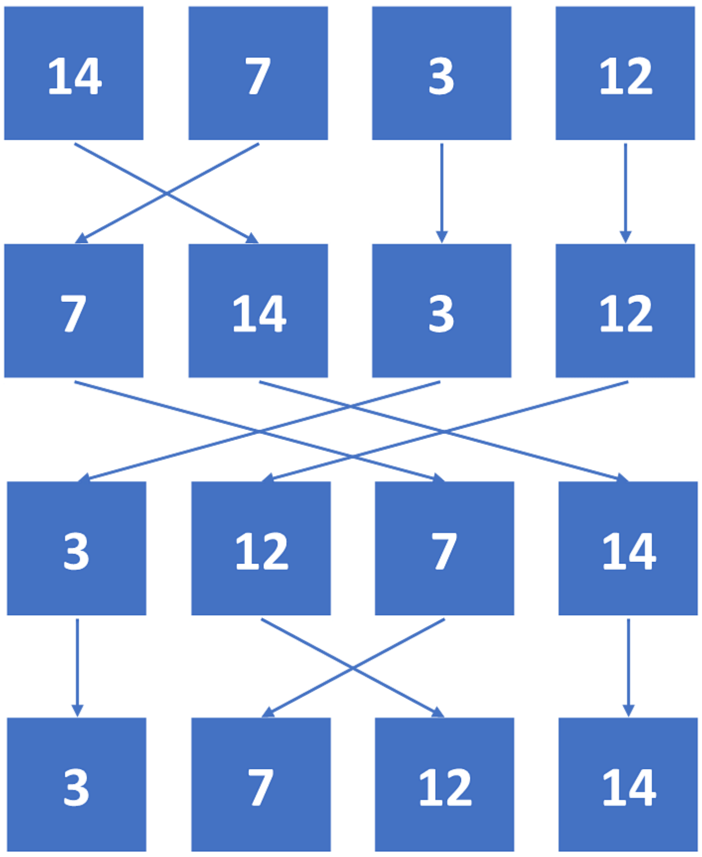
Merge sort algorithm can be executed in two ways:

- Top-down Approach



It starts at the top and works its way down, splitting the array in half, making a recursive call, and merging the results until it reaches the bottom of the array tree.

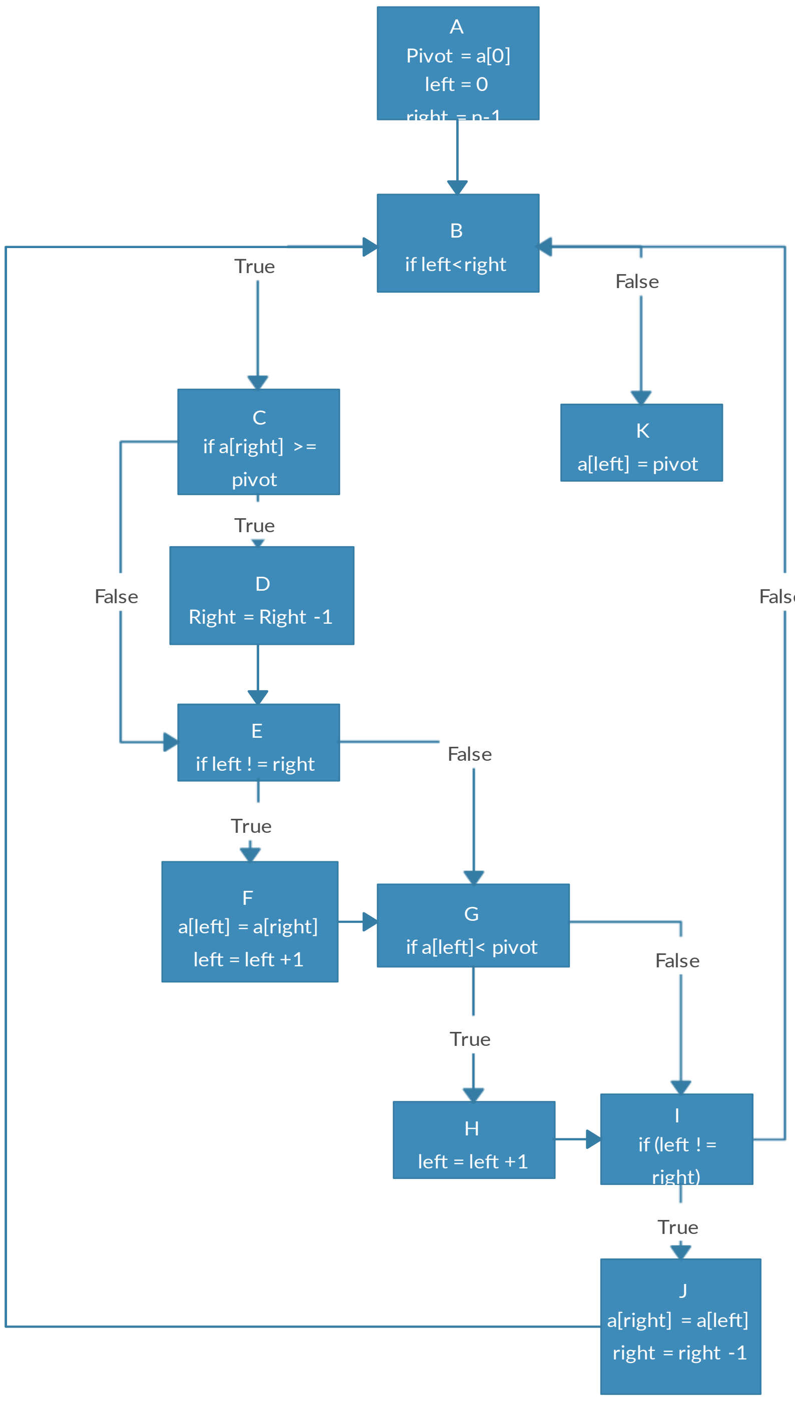
- Bottom-Up Approach



The iterative technique is used in the Bottom-Up merge sort approach. It starts with a "single-element" array and then merges two nearby items while also sorting them. The combined-sorted arrays are merged and sorted again until only one single unit of the sorted array remains.

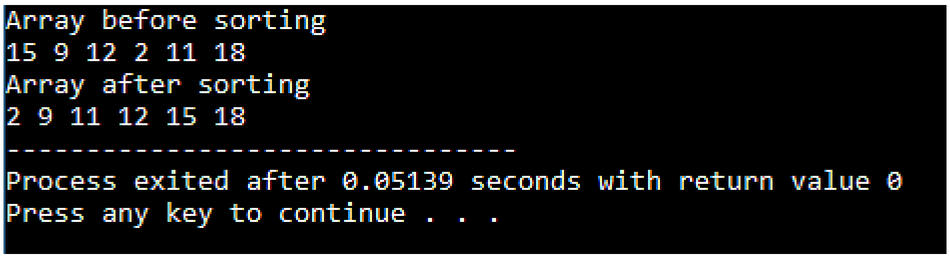
How to Implement the Merge Sort Algorithm?

It splits the input array in half, calls itself for each half, and then combines the two sorted parts. Merging two halves is done with the merge() method. Merge (array[], left, mid, right) is a crucial process that assumes array[left..mid] and array[mid+1..right] are both sorted sub-arrays and merges them into one.



Code:

```c
#include <stdio.h>
// Function to merge two sorted subarrays
void merge(int arr[], int left, int mid, int right) {
    int n1 = mid - left + 1; // Size of the left subarray
    int n2 = right - mid;    // Size of the right subarray
    // Create temporary arrays
    int L[n1], R[n2];
    // Copy data to temporary arrays L[] and R[]
    for (int i = 0; i < n1; i++)
        L[i] = arr[left + i];
    for (int j = 0; j < n2; j++)
        R[j] = arr[mid + 1 + j];
    // Merge the temporary arrays back into arr[left..right]
    int i = 0, j = 0, k = left;
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k] = L[i];
            i++;
        } else {
            arr[k] = R[j];
            j++;
        }
        k++;
    }
    // Copy the remaining elements of L[], if any
    while (i < n1) {
        arr[k] = L[i];
        i++;
        k++;
    }
    // Copy the remaining elements of R[], if any
    while (j < n2) {
        arr[k] = R[j];
        j++;
        k++;
    }
}
// Function to implement Merge Sort
void mergeSort(int arr[], int left, int right) {
    if (left < right) {
        // Find the middle point
        int mid = left + (right - left) / 2;
        // Sort the first and second halves
        mergeSort(arr, left, mid);
        mergeSort(arr, mid + 1, right);
        // Merge the sorted halves
        merge(arr, left, mid, right);
    }
}
// Function to print the array
void printArray(int arr[], int size) {
    for (int i = 0; i < size; i++)
        printf("%d ", arr[i]);
    printf("\n");
}
// Main function
int main() {
    int arr[] = {12, 11, 13, 5, 6, 7};
    int size = sizeof(arr) / sizeof(arr[0]);
    printf("Original array: \n");
    printArray(arr, size);
    // Sort the array
    mergeSort(arr, 0, size - 1);
    printf("Sorted array: \n");
    printArray(arr, size);
    return 0;
}
```



```
Array before sorting
15 9 12 2 11 18
Array after sorting
2 9 11 12 15 18
--------------------------------
Process exited after 0.05139 seconds with return value 0
Press any key to continue . . .
```

What Are the Advantages of the Merge Sort?

- Merge sort can efficiently sort a list in O(n*log(n)) time.
- Merge sort can be used with linked list without taking up any more space.
- A merge sort algorithm is used to count the number of inversions in the list.
- Merge sort is employed in external sorting.

What Are the Drawbacks of the Merge Sort?

- For small datasets, merge sort is slower than other sorting algorithms.
- For the temporary array, merge sort requires an additional space of O(n).
- Even if the array is sorted, the merge sort goes through the entire process.

What Is Selection Sort Algorithm in Data Structures?

Selection sort is a simple comparison-based sorting algorithm that divides the input list into a sorted part at the beginning and an unsorted part at the end. The algorithm repeatedly selects the smallest (or largest, depending on the order) element from the unsorted part and swaps it with the first unsorted element, gradually growing the sorted portion until the entire list is sorted.
In the selection sort, the cost of swapping is irrelevant. Swapping refers to interchanging the positions of two elements in the array. All components must be checked in the selection sort and the cost of writing to memory matters. This is like flash memory, where the number of writes/swaps is O(n) instead of O(n2) in bubble sort.

- Selection sort is a straightforward and efficient comparison-based sorting algorithm ideal for small datasets.
- Each iteration incrementally builds a sorted section by adding one element at a time.
- The algorithm works by identifying the smallest element in the unsorted portion of the array and swapping it with the first unsorted element, effectively moving it to the front.
- Alternatively, depending on the sorting order desired, the largest element can be selected and placed at the end of the array.
- During each iteration, the selection sort identifies the correct element and positions it appropriately within the sorted section of the array.

How Does the Selection Sort Algorithm Work?
Selection sort takes the smallest element in an unsorted array and brings it to the front. You'll review each item (left to right) until you find the smallest one. The first item in the array is now sorted, while the rest of the array is unsorted. Here's the step-by-step process:

1. Initialization

- Start with an unsorted array or list.
- The array is conceptually divided into two parts: the sorted part, which is initially empty, and the unsorted part, which contains all the elements.

2. Iteration Over the Array

- The algorithm iterates over the array, one element at a time, looking for the smallest element in the unsorted portion during each iteration.

3. Finding the Minimum Element

- In each iteration, assume that the first element of the unsorted portion is the smallest.
- Compare this element with the rest of the elements in the unsorted portion.
- If a smaller element is found, update the smallest element to this new value.

4. Swapping Elements

- After the smallest element in the unsorted portion is found, swap it with the first element of the unsorted portion. This effectively moves the smallest element to its correct position in the sorted portion.

5. Reduce the Unsorted Portion

- After each swap, the boundary between the sorted and unsorted portions moves one element to the right. The sorted portion grows by one element, and the unsorted portion decreases by one.

6. Repeat the Process

- The process is repeated for the remaining unsorted elements. Each time, the next smallest element is selected and moved to the end of the sorted portion.

7. Completion

- The algorithm continues until the entire array is sorted. The array is fully sorted when the unsorted portion is reduced to a single element.

Let's take a simple example to illustrate how selection sort works. Consider the array: [29, 10, 14, 37, 14].

1. First Iteration

- The unsorted array is [29, 10, 14, 37, 14].
- Find the smallest element in the unsorted portion: 10.
- Swap 10 with the first element (29).
- The array now looks like this: [10, 29, 14, 37, 14].

2. Second Iteration

- The unsorted array is [29, 14, 37, 14].
- Find the smallest element in the unsorted portion: 14.
- Swap 14 with the first element of the unsorted portion (29).
- The array now looks like this: [10, 14, 29, 37, 14].

3. Third Iteration

- The unsorted array is [29, 37, 14].
- Find the smallest element in the unsorted portion: 14.
- Swap 14 with the first element of the unsorted portion (29).
- The array now looks like this: [10, 14, 14, 37, 29].

4. Fourth Iteration

- The unsorted array is [37, 29].
- Find the smallest element in the unsorted portion: 29.
- Swap 29 with the first element of the unsorted portion (37).
- The array now looks like this: [10, 14, 14, 29, 37].

5. Final Array

- The array is now fully sorted: [10, 14, 14, 29, 37].

Time Complexity

Selection sort has a time complexity of $O(n^2)$, where n is the number of elements in the array. For each n element, the algorithm makes n-1 comparisons to find the minimum element. This quadratic time complexity makes selection sort less efficient for large datasets than more advanced algorithms like quick sort or merge sort.

Space Complexity

Selection sort is an in-place sorting algorithm, meaning it doesn't require additional storage space apart from the input array. Its space complexity is O(1), as it only uses constant extra memory.

Algorithm of the Selection Sort Algorithm

Step 1: Set Min to location 0 in Step 1.

Step 2: Look for the smallest element on the list.

Step 3: Replace the value at location Min with a different value.

Step 4: Increase Min to point to the next element

Step 5: Continue until the list is sorted.

Pseudocode of Selection Sort Algorithm

```
void selectionSort(int arr[], int n) {
   int i, j, minIndex, temp;
   // Move the boundary of the unsorted portion one element to the right after each iteration
   for (i = 0; i < n-1; i++) {
      // Assume the first element is the minimum
      minIndex = i;
      // Find the minimum element in the unsorted portion of the array
      for (j = i+1; j < n; j++) {
         if (arr[j] < arr[minIndex]) {
            minIndex = j;
         }
      }
      // Swap the found minimum element with the first element of the unsorted portion
      temp = arr[minIndex];
      arr[minIndex] = arr[i];
      arr[i] = temp;
   }
}
```