Sorting Algorithm

A sorting algorithm is used to arrange elements of an array/list in a specific order. For example,

Here, we are sorting the array in ascending order.

There are various sorting algorithms that can be used to complete this operation. And, we can use any algorithm based on the requirement.

Sorting Algorithms Code:

This C++ program demonstrates several sorting algorithms (Bubble Sort, Selection Sort, Merge Sort) and searching algorithms (Binary Search both iterative and recursive). Let's explain each function step by step.

swap(int *a, int *b)

This function swaps two integers using pointers. It temporarily stores the value of the first variable, assigns the value of the second variable to the first, and then assigns the stored value to the second variable. This is a common utility function used in sorting algorithms where swapping elements is frequent.

display(int size, int arr[])

This function prints the elements of an array. It iterates over the array using a loop and outputs each element followed by a space. This is useful for visualizing the contents of the array after sorting.

bubbleSort(int size, int arr[])

Bubble Sort is a simple sorting algorithm. This function iterates over the array, repeatedly comparing and swapping adjacent elements if they are in the wrong order. This process is repeated until the array is sorted. The number of comparisons is also tracked and displayed at the end. The function finally calls display to show the sorted array.

For a more detailed explanation and visualization go through the article:

https://www.programiz.com/dsa/bubble-sort

insertionSort(int size, int arr[])

Insertion Sort is another simple sorting algorithm where the array is built one element at a time. It involves comparing each new element with already sorted elements and inserting it into the correct position. This function doesn't include a display or comparison count functionality.

For a more detailed explanation and visualization go through the article:

https://www.programiz.com/dsa/insertion-sort

selectionSort(int size, int arr[])

Selection Sort improves on the bubble sort by making only one exchange for every pass through the list. It looks for the smallest (or largest) element in the array and swaps it with the first unsorted element. The number of comparisons is counted and displayed, similar to Bubble Sort.

For a more detailed explanation and visualization go through the article:

https://www.programiz.com/dsa/selection-sort

merge(int arr[], int left, int mid, int right)

This function is a part of the Merge Sort algorithm. It merges two subarrays of arr[]. The first subarray is arr[left..mid] and the second subarray is arr[mid+1..right]. Merge Sort is

a divide and conquer algorithm that divides the input array into two halves, calls itself for the two halves, and then merges the two sorted halves.

mergeSort(int arr[], int l, int r)

Merge Sort algorithm is implemented here. If the left index is smaller than the right index, the array is divided and mergeSort is called recursively for the two halves, and then the merge function is used to merge the sorted halves.

For a more detailed explanation and visualization go through the article:

https://www.programiz.com/dsa/merge-sort

binarySearchRecursive(int arr[], int left, int right, int key, int &comparisons)

This function implements the Binary Search algorithm recursively. It searches for a key in a sorted array arr[] between indices left and right. The number of comparisons made during the search is also counted.

For a more detailed explanation and visualization go through the article:

https://www.programiz.com/dsa/binary-search

binarySearchIterative(int arr[], int size, int key, int &comparisons)

This is an iterative version of the Binary Search algorithm. It repeatedly divides the portion of the array under consideration in half until the key is found or the remaining portion is empty. The number of comparisons is also tracked.

For a more detailed explanation and visualization go through the article:

https://www.programiz.com/dsa/binary-search

main()

The main function provides a menu-driven interface to execute these algorithms. It allows the user to choose among Bubble Sort, Selection Sort, Merge Sort, and both recursive and iterative Binary Search. The array size and elements are input by the user. After executing the chosen algorithm, the results are displayed along with the number of comparisons made during the sorting or searching.

Code:

```
#include<iostream>
using namespace std;
void swap(int *a, int *b)
{
        int temp=*a;
        *a=*b;
        *b=temp;
}
void display(int size, int arr[]) {
   for (int i = 0; i < size; i++) {
      cout << arr[i] << "  ";
   }
   cout << endl;
}

void bubbleSort(int size, int arr[]) {
   int comparisons = 0;
   for (int i = 0; i < size - 1; i++) {
      for (int j = 0; j < size - i - 1; j++) {
         comparisons++;
         if (arr[j] > arr[j + 1]) {
            // Swap elements if they are in the wrong order
            swap(&arr[j], &arr[j + 1]);
         }
      }
   }
   display(size, arr);
   cout << "Bubble Sort Comparisons: " << comparisons << endl;
}

void insertionSort(int size, int arr[])
{
   int i, key, j;
   for (i = 1; i < size; i++) {
      key = arr[i];
      j = i - 1;
```

```
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
    }
}
void selectionSort(int size, int arr[]) {
    int comparisons = 0;
    for (int i = 0; i < size - 1; i++) {
        int min_index = i;
        for (int j = i + 1; j < size; j++) {
            comparisons++;
            if (arr[j] < arr[min_index]) {
                // Update the index of the minimum element
                min_index = j;
            }
        }
        // Swap the found minimum element with the first element
        swap(&arr[i], &arr[min_index]);
    }
    cout << "Selection Sort Comparisons: " << comparisons << endl;
    display(size, arr);
}

// Merge two subarrays L and M into arr
void merge(int arr[], int left, int mid, int right) {

    // Create L ← A[p..q] and M ← A[q+1..r]
    int n1 = mid - left + 1;
    int n2 = right - mid;

    int L[n1], M[n2];

    for (int i = 0; i < n1; i++)
        L[i] = arr[left + i];
    for (int j = 0; j < n2; j++)
```

```
        M[j] = arr[mid + 1 + j];

    // Maintain current index of sub-arrays and main array
    int i, j, k;
    i = 0;
    j = 0;
    k = left;

    // Until we reach either end of either L or M, pick larger among
    // elements L and M and place them in the correct position at A[p..r]
    while (i < n1 && j < n2) {
        if (L[i] <= M[j]) {
            arr[k] = L[i];
            i++;
        } else {
            arr[k] = M[j];
            j++;
        }
        k++;
    }

    // When we run out of elements in either L or M,
    // pick up the remaining elements and put in A[p..r]
    while (i < n1) {
        arr[k] = L[i];
        i++;
        k++;
    }

    while (j < n2) {
        arr[k] = M[j];
        j++;
        k++;
    }
}
```

```
void mergeSort(int arr[], int l, int r) {
  if (l < r) {
    // m is the point where the array is divided into two subarrays
    int m = l + (r - l) / 2;

    mergeSort(arr, l, m);
    mergeSort(arr, m + 1, r);

    // Merge the sorted subarrays
    merge(arr, l, m, r);
  }
}


int binarySearchRecursive(int arr[], int left, int right, int key, int &comparisons) {
    if (left <= right) {
        int mid = left + (right - left) / 2;
        comparisons++;
        if (arr[mid] == key) {
            return mid; // Key found
        } else if (arr[mid] < key) {
            return binarySearchRecursive(arr, mid + 1, right, key, comparisons);
        } else {
            return binarySearchRecursive(arr, left, mid - 1, key, comparisons);
        }
    }
    return -1; // Key not found
}

int binarySearchIterative(int arr[], int size, int key, int &comparisons) {
    int left = 0, right = size - 1;
    while (left <= right) {
        int mid = left + (right - left) / 2;
        comparisons++;
        if (arr[mid] == key) {
            return mid; // Key found
        } else if (arr[mid] < key) {
```

```
            left = mid + 1;
        } else {
            right = mid - 1;
        }
    }
    return -1; // Key not found
}

int main() {
    int size;
    cout << "Enter the size of the array: ";
    cin >> size;

    int arr[size];
    cout << "Enter the elements of the array:\n";
    for (int i = 0; i < size; i++) {
        cin >> arr[i];
    }

    // Display the menu
    cout << "\nMenu:\n";
    cout << "1. Bubble Sort\n";
    cout << "2. Selection Sort\n";
    cout << "3. Merge Sort\n";
    cout << "4. Binary Search (Recursive)\n";
    cout << "5. Binary Search (Iterative)\n";
    cout << "Enter your choice: ";

    int choice;
    cin >> choice;

    switch (choice) {
        case 1:
            bubbleSort(size, arr);
            break;
        case 2:
            selectionSort(size, arr);
```

```
            break;
        case 3:
            mergeSort(arr, 0, size-1);
            display(size, arr);
            break;
        case 4: {
            int key;
            cout << "Enter the value to find: ";
            cin >> key;
            int comparisons = 0;
            int result = binarySearchRecursive(arr, 0, size - 1, key, comparisons);
            if (result != -1) {
                cout << "Element found at index " << result << endl;
            } else {
                cout << "Element not found\n";
            }
            cout << "Comparisons: " << comparisons << endl;
            break;
        }
        case 5: {
            int key;
            cout << "Enter the value to find: ";
            cin >> key;
            int comparisons = 0;
            int result = binarySearchIterative(arr, size, key, comparisons);
            if (result != -1) {
                cout << "Element found at index " << result << endl;
            } else {
                cout << "Element not found\n";
            }
            cout << "Comparisons: " << comparisons << endl;
            break;
        }
        default:
            cout << "Invalid choice\n";
    }
```

```
    return 0;
}
```