

Lab 2: Implementation of divide and conquer algorithms(Quicksort, Merge sort) using C++

Objective:

To implement divide and conquer algorithms to implement quicksort and merge sort using C++.

Theory:

- 1) Divide and Conquer: Divide and conquer is an algorithmic paradigm that involves solving a problem by breaking it down into smaller, more manageable subproblems. Each subproblem is solved independently, often recursively, and then the solutions to the subproblems are combined to form the solution to the original problem. This approach is particularly effective for problems that can be divided into similar subproblems, such as sorting algorithms (e.g., quicksort and mergesort) and searching algorithms (e.g., binary search).
 - a) Divide: Break the original problem into smaller subproblems, which are typically of the same type as the original problem. This step reduces the complexity by simplifying the problem.
 - b) Conquer: Solve the subproblems independently. If the subproblems are still too large, apply the divide and conquer approach recursively until the subproblems are simple enough to be solved directly.
 - c) Merge: Merge the solutions of the subproblems to form a solution to the original problem. This step typically involves combining the results in a way that addresses the structure of the original problem.
- 2) Applications of Divide and Conquer Algorithm:
 - a) Quicksort: It is a very efficient sorting algorithm, which is also known as a partitionexchange sort. It starts by selecting a pivot value from an array followed by dividing the rest of the array elements into two sub-arrays. The partition is made by comparing each of the elements with the pivot value. It compares whether the element holds a greater value or lesser value than the pivot and then sort the arrays recursively.

Algorithm:

 - i) Consider the first element of the array as pivot.
 - ii) Define two variables left and right. Set left and right to first and last elements of the array respectively.
 - iii) Increment left until array[left] > pivot then stop.
 - iv) Decrement right until array[right] < pivot then stop.

- v) If $\text{left} < \text{right}$ then exchange $\text{array}[\text{left}]$ and $\text{array}[\text{right}]$.
- vi) Repeat steps 3,4 & 5 until $\text{left} > \text{right}$.
- vii) Exchange the pivot element with $\text{array}[\text{right}]$ element.

b) Merge Sort: It is a sorting algorithm that follows the divide and conquer approach. It works by recursively dividing the input array into smaller subarrays and sorting those subarrays then merging them back together to obtain the sorted array. In simple terms, we can say that the process of merge sort is to divide the array into two halves, sort each half and merge the sorted halves back together. The process is repeated until the array is sorted.

Algorithm:

- i) Create two pointers, one for each sorted half.
- ii) Initialize an empty temporary array to hold the merged result.
- iii) Compare the elements at the pointers of the two halves.
- iv) Copy the smaller element into the temporary array.
- v) Move the pointer of the sub-list with the smaller element forward.
- vi) Repeat step 3 until one of the sub-list is empty.
- vii) Copy the remaining elements from the non-empty sub-list to the temporary array.
- viii) Copy the elements back from the temporary array to the original list.

Observation:

```
#include <iostream>
#include <functional>

using namespace std;

class Sort{

public:
void swap(int *a, int *b){
    int temp = *a;
    *a = *b;
    *b = temp;
}

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 pi = partition(arr, low, high);
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}

void merge(int arr[], int mid, int left, int right){
```

```

int subArr1 = mid - left + 1;
int subArr2 = right - mid;

auto *leftArr = new int[subArr1];
auto *rightArr = new int[subArr2];

for (int i = 0; i < subArr1; i++){
    leftArr[i] = arr[left + i];
}
for (int i = 0; i < subArr2; i++){
    rightArr[i] = arr[mid + 1 + i];
}

auto indexofSubArr1 = 0;
auto indexofSubArr2 = 0;

int indexofMergedArr = left;

while(indexofSubArr1 < subArr1 && indexofSubArr2 < subArr2){
    if(leftArr[indexofSubArr1] <= rightArr[indexofSubArr2]){
        arr[indexofMergedArr] = leftArr[indexofSubArr1];
        indexofSubArr1++;
    }
    else{
        arr[indexofMergedArr] = rightArr[indexofSubArr2];
        indexofSubArr2++;
    }
    indexofMergedArr++;
}

while(indexofSubArr1 < subArr1){
    arr[indexofMergedArr] = leftArr[indexofSubArr1];
    indexofSubArr1++;
    indexofMergedArr++;
}

while (indexofSubArr2 < subArr2){
    arr[indexofMergedArr] = rightArr[indexofSubArr2];
    indexofSubArr2++;
    indexofMergedArr++;
}

```

```

    }
    delete[] leftArr;
    delete[] rightArr;
}

void mergeSort(int arr[], int left, int right){
    if(left >= right){
        return;
    }
    int mid = left + (right - left) / 2;
    mergeSort(arr, left, mid);
    mergeSort(arr, mid + 1, right);
    merge(arr, mid, left, right);
}

void toString(int arr[], int size){
    for (int i = 0; i < size; i++){
        cout << arr[i] << " ";
    }
    cout << endl;
}

};

long long getTime(std::function<void()> f){
    auto start = clock();
    f();
    auto end = clock();
    long double duration = end - start;
    return (duration/CLOCKS_PER_SEC) * 1000000000;
}

int main(){
    Sort sort;
    int size = 100000;
    int arr[size];
    for (int i = 0; i < size; i++){
        int v = rand() % size;
        arr[i] = v;
    }
    auto sorted = [&]() {

```

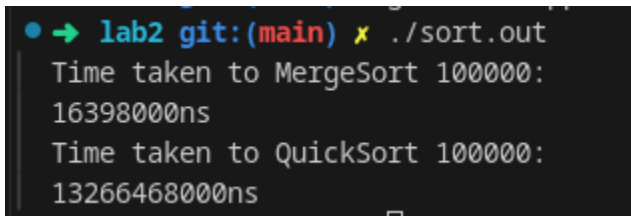
```

    sort.mergeSort(arr, 0, size - 1);
};

auto sorted2 = [&]() {
    sort.quickSort(arr, 0, size - 1);
};

cout << "Time taken to MergeSort " << size << ": \n" << getTime(sorted) << "ns" <<
endl;
cout << "Time taken to QuickSort " << size << ": \n" << getTime(sorted2) << "ns" <<
endl;
return 0;
}

```



```

lab2 git:(main) x ./sort.out
Time taken to MergeSort 100000:
16398000ns
Time taken to QuickSort 100000:
13266468000ns

```

Conclusion:

In this way, we implemented divide and conquer algorithms(quicksort, merge sort) using C++.