CMPT 125 Assignment 5

- Due at 15:20:00 on Friday, Feb. 15
- Please make your submission on CourSys by uploading the following:
  - A .pdf document of your answers
  - A .c file containing all your code
- You may write or type your answers; take clear pictures of your work if you write.

Question 1 (25 marks)
In assignment 3 we learned recursive implementations of selection sort. In this question, we explore divide and conquer technique to implement quick sort.

In quicksort we pick an element as pivot and partitions the given array into two sub arrays under the condition of each elements of the array greater and less than the pivot. There are many different versions of quicksort that pick pivot in different ways.

1. Always pick first element as pivot.
2. Always pick last element as pivot
3. Pick a random element as pivot.
4. Pick median as pivot.

We will take option 2 to solve this problem. We start from the leftmost element under consideration and keep track of index of last the element smaller the pivot as mid. We then scan the array, and while scanning, if we find an element smaller than the pivot, we swap current element with arr[mid+1]. Otherwise we ignore current element and keep scanning. Like assignment 3, we will break our implementation into 3 separate functions and use them together at the end.

a) First, write a function called swap, which simply swaps the current element with arr[i].

```c
void swap(int arr[], int i, int j) {
    // swaps arr[i] and arr[j]
}
```

Solution:

```c
void swap(int arr[], int i, int j) {
    // swaps arr[i] and arr[j]
    int tmp = arr[i];
    arr[i] = arr[j];
    arr[j] = tmp;
}
```

b) The key process in quick sort is partition(). This function takes last element as pivot places the pivot element at its correct position in the sorted array and places all elements smaller than the pivot to left of pivot and all greater elements to right.

```c
int partition(int arr[], int first, int last) {
    // pivot (element to be placed at the "middle" position);
    int pivot = arr[last];
    // your code here to partition
}
```
Solution:

```c
int partition(int arr[], int first, int last) {
    // pivot (Element to be placed at the "middle" position);
    int pivot = arr[last];

    // initialize and keep track of the last processed element smaller than pivot;
    // use the mid variable from lecture slides on the partition function
    int mid = first - 1;

    // go through array element by element, and swap with mid+1
    // if current element is smaller than pivot
    for (int sweep = first; sweep < last; sweep++) {
        if (arr[sweep] < pivot) {
            swap(arr, mid+1, sweep);
            mid++;
        }
    }

    swap(arr, mid+1, last);
    // Careful with this swap, since pivot is at the end of array,
    // instead of the beginning as it was in class
    return mid+1;
}
```

c) Finally, write a recursive quick sort function, called quickSortR(). This function would find the partitioning index using the partition function from step 2. And separately sort elements before partition and after partition.

```c
void quickSortR(int arr[], int first, int last) {
    // See code from the lecture on quicksort
}
```

```c
void main() {
    int arr[14] = {488888, 3, 5, 0, 23, 12124, 6, 7, 2, 1121, 0, 92, 5, 8};
    quickSortR(arr, 0, 13);
}
```
for (int i = 0; i<14; i++) {
    printf("arr[%d] = %d\n", i, arr[i]);
}

Solution: (Basically the same as lecture slides)

// Post:  arr[first..last] are sorted
void quickSortR(int arr[], int first, int last) {
    // Base case
    if (last <= first) {
        return;
    }

    // Split array
    int mid = partition(arr, first, last);

    // Recursively sort
    quickSortR(arr, first, mid-1);
    quickSortR(arr, mid+1, last);
}
Question 2 (25 marks)

Given an array of N elements, you notice that there are some duplicate values. Implement a modified merge function such that the mergesort code we saw in class both sorts the array and removes all duplicates from the array. Your algorithm should run in O(N log N) time; briefly argue/justify that your program runs in O(N log N) time.

Note: You are not allowed to simply call mergesort, and then remove duplicates by scanning the array after sorting it. You must modify the merge function.

Note: You may assume all elements in the original array are positive, and fill in “empty” spots with -1.

Example input:

\[
\begin{array}{cccccccc}
5 & 2 & 5 & 9 & 1 & 2 & 8 & 4 & 3 & 6
\end{array}
\]

Example correct outputs from example input:

\[
\begin{array}{ccccccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 8 & 9 & -1 & -1
\end{array}
\]

\[
\begin{array}{ccccccccccc}
1 & -1 & 2 & 3 & 4 & 5 & 6 & 8 & 9 & -1
\end{array}
\]

\[
\begin{array}{ccccccccccc}
1 & -1 & 2 & 3 & 4 & 5 & 6 & -1 & 8 & 9
\end{array}
\]

Any of the above is correct. As long as there are no duplicate positive numbers, and as long as positive numbers are in sorted order if all -1 elements are ignored, the output would correct. You don’t need to sort the -1’s.
Solution: Modifications from the lecture slides are bolded

```c
// Pre: arr[first..mid] and arr[mid+1..last] are sorted
// Post: arr[first..last] are sorted without duplicate positive elements
void merge(int arr[], int first, int mid, int last) {
    int len = last - first + 1;
    int newArr[len];

    // Initialize the temporary array with -1's, so that when
    // it is copied back to the original array, empty spaces
    // are filled with -1's
    for (int i = 1; i < len; i++) {
        newArr[i] = -1;
    }

    int left = first; int right = mid + 1; int newPos = 0;

    while (left <= mid && right <= last) {
        if (arr[left] <= arr[right]) {
            // Note the "<=" instead of "<" in the if statement
            // In the case that the two "heads" of subarrays are
            // equal, we also increment the head of the right
            // subarray, so that we skip over duplicates
            if (arr[left] == arr[right]) {
                right++;
            }
            newArr[newPos++] = arr[left++];
        } else {
            newArr[newPos++] = arr[right++];
        }
    }

    // Flush non empty piece
    arrCpy(newArr + newPos, arr + left, mid - left + 1);
    arrCpy(newArr + newPos, arr + right, last - right + 1);

    arrCpy(arr + first, newArr, len);
}
```