HW Solution #5

Only P1-P5 will be graded.

P1:

Merge sort:

1. [254] [564] [425] [628] [614] [021] [213] [117]
2. [254, 564] [425, 628] [021, 614] [117, 213]
3. [254, 425, 564, 628] [021, 117, 213, 614]
4. [021, 117, 213, 254, 425, 564, 614, 628]

Quick sort:

1. [254, 564, 425, 628, 614, 021, 213, 117]
2. [021, 117] 254 [564, 425, 628, 614, 213]
3. 021, 117, 254, [425, 213], 564, [628, 614]
4. 021, 117, 213, 254, 425, 564, 614, 628

Radix sort:

1. 021, 213, 254, 564, 614, 425, 117, 628
2. 213, 614, 117, 021, 425, 628, 254, 564
3. 021, 117, 213, 254, 425, 564, 614, 826

P2: Merge sort and insertion sort are stable.

P3:

1. 5, 13, 2, 25, 7, 17, 20, 8, 4
2. 5, 13, 20, 25, 7, 17, 2, 8, 4
3. 5, 25, 20, 13, 7, 17, 2, 8, 4

4. 25, 5, 20, 13, 7, 17, 2, 8, 4
5. 25, 13, 20, 5, 7, 17, 2, 8, 4
6. 25, 13, 20, 8, 7, 17, 2, 5, 4 (Heapification done)
7. 4, 13, 20, 8, 7, 17, 2, 5, 25
8. 20, 13, 4, 8, 7, 17, 2, 5, 25
9. 20, 13, 17, 8, 7, 4, 2, 5, 25
10. 5, 13, 17, 8, 7, 4, 2, 20, 25
11. 17, 13, 5, 8, 7, 4, 2, 20, 25
12. 2, 13, 5, 8, 7, 4, 17, 20, 25
13. 13, 2, 5, 8, 7, 4, 17, 20, 25
14. 13, 8, 5, 2, 7, 4, 17, 20, 25
15. 4, 8, 5, 2, 7, 13, 17, 20, 25
16. 8, 4, 5, 2, 7, 13, 17, 20, 25
17. 8, 7, 5, 2, 4, 13, 17, 20, 25
18. 4, 7, 5, 2, 8, 13, 17, 20, 25
19. 7, 4, 5, 2, 8, 13, 17, 20, 25
20. 2, 4, 5, 7, 8, 13, 17, 20, 25
21. 5, 4, 2, 7, 8, 13, 17, 20, 25
22. 2, 4, 5, 7, 8, 13, 17, 20, 25
23. 4, 2, 5, 7, 8, 13, 17, 20, 25
24. 2, 4, 5, 7, 8, 13, 17, 20, 25

Or with a min-heap:

1. 5, 13, 2, 25, 7, 17, 20, 8,4
2. 5, 13, 2, 4, 7, 17, 20, 8, 25
3. 5, 4, 2, 8, 7, 17, 20, 13, 25
4. 2, 4, 5, 8, 7, 17, 20, 13, 25 (Heapification done)
5. 4, 7, 5, 8, 25, 17, 20, 13, 2
6. 5, 7, 13, 8, 25, 17, 20, 4, 2

7. 7, 8, 13, 20, 25, 17, 5, 4, 2
8. 8, 17, 13, 20, 25, 7 , 5, 4, 2
9. 13, 17, 25, 20, 8, 7, 5, 4, 2
10. 17, 20, 25, 13, 8, 7, 5, 4, 2
11. 20, 25, 17, 13, 8, 7, 5, 4, 2
12. 25, 20, 17, 13, 8, 7, 5, 4, 2

```
// This function sorts the
// input array and returns the
// number of inversions in the array
int mergeSort(int arr[], int array_size)
{
    int temp[array_size];
    return _mergeSort(arr, temp, 0, array_size - 1);
}

// An auxiliary recursive function
// that sorts the input array and
// returns the number of inversions in the array.
int _mergeSort(int arr[], int temp[], int left, int right)
{
    int mid, inv_count = 0;
    if (right > left) {
        // Divide the array into two parts and
        // call _mergeSortAndCountInv()
        // for each of the parts
        mid = (right + left) / 2;

        // Inversion count will be sum of
        // inversions in left-part, right-part
        // and number of inversions in merging
        inv_count += _mergeSort(arr, temp, left, mid);
        inv_count += _mergeSort(arr, temp, mid + 1, right);

        // Merge the two parts
        inv_count += merge(arr, temp, left, mid + 1, right);
    }
    return inv_count;
}

// This function merges two sorted arrays
```

```
// and returns inversion count in the arrays.
int merge(int arr[], int temp[], int left, int mid,
          int right)
{
    int i, j, k;
    int inv_count = 0;

    i = left;
    j = mid;
    k = left;
    while ((i <= mid - 1) && (j <= right)) {
        if (arr[i] <= arr[j]) {
            temp[k++] = arr[i++];
        }
        else {
            temp[k++] = arr[j++];

            // this is tricky -- see above
            // explanation/diagram for merge()
            inv_count = inv_count + (mid - i);
        }
    }

    // Copy the remaining elements of left subarray
    // (if there are any) to temp
    while (i <= mid - 1)
        temp[k++] = arr[i++];

    // Copy the remaining elements of right subarray
    // (if there are any) to temp
    while (j <= right)
        temp[k++] = arr[j++];

    // Copy back the merged elements to original array
    for (i = left; i <= right; i++)
        arr[i] = temp[i];

    return inv_count;
}

// Driver code
int main()
{
    int arr[] = { 1, 20, 6, 4, 5 };
    int n = sizeof(arr) / sizeof(arr[0]);
```

```
    int ans = mergeSort(arr, n);
    cout << " Number of inversions are " << ans;
    return 0;
}
```

P5: The naive approach is do a comparison-based sort, which will yield $O(N \log(N))$. Counting sort can give a $O(N)$ solution, but the maximum citation can be very large. A trick is to just make the maximum bucket as N. The h-index is at most N, so we can just reserve N buckets. All the citations above N can only contribute to the last entry.

```
int hIndex(vector<int>& citations) {
    const int n = citations.size(); // N
    vector<int> count(n + 1, 0); // The buckts
    for (const auto& x : citations) {
        // Put all x >= n in the same bucket.
        if (x >= n) {
            ++count[n]; // If the citation for one paper is beyond N
        } else {
            ++count[x];
        }
    }

    int h = 0;
    for (int i = n; i >= 0; --i) {
        h += count[i];
        if (h >= i) {
            return i;
        }
    }
    return h;
```

P6: The idea is based on the observation that to minimize the difference, we must choose consecutive elements from a sorted packet. We first sort the array arr[0..n-1], then find the subarray of size m with the minimum difference between the last and first elements.

1. Initially sort the given array. Declare a variable to store the minimum difference, and initialize it to INT_MAX. Let the variable be min_diff.
2. Find the subarray of size m such that the difference between the last (maximum in case of sorted) and first (minimum in case of sorted) elements of the subarray is minimum.
3. We will run a loop from 0 to (n-m), where n is the size of the given array and m is the size of the subarray.

4. We will calculate the maximum difference with the subarray and store it in diff = arr[highest index] – arr[lowest index]
5. Whenever we get a diff less than the min_diff, we will update the min_diff with diff.