Problem Set 2

Assigned: June 3
Due: June 10

PROBLEM 1.

Suppose that you are given the problem of returning in sorted order the K smallest elements in an array of size N, where K is much smaller than log(N), but much larger than 1.

a. Describe how selection sort, insertion sort, mergesort, and heapsort can be adapted to this problem. Your description need not give the pseudo-code for the modified algorithms; it is enough simply to describe what changes can be made, as long as your description is clear. You may use the recursive version of mergesort.

   Ans: Selection Sort:
   1. Add input parameter K and return parameter as an array of K elements.
   2. Check if K is greater than log(N) or is smaller than 1 then return error.
   3. Change the first for Loop to (1 to K) instead of (1 to N).
   4. At the end of sorting, return the first K elements of the array sorted till Kth element.

   Insertion Sort:
   1. Add input parameter K and return parameter as an array of K elements.
   2. Check if K is greater than log(N) or is smaller than 1 then return error.
   3. At the end of sorting, return the first K elements of the sorted array.

   Merge Sort:
   1. Add input parameter K and return parameter as an array of K elements.
   2. Check if K is greater than log(N) or is smaller than 1 then return error.
   3. At the end of sorting, return the first K elements of the sorted array.

   Heap Sort:
   1. Add input parameter K and return parameter as an array of K elements.
   2. Check if K is greater than log(N) or is smaller than 1 then return error.
   3. Do DELETEMIN only K times instead of N times, i.e. change the second for loop to 1 to K instead of I to N.
   4. At the end of sorting, return the first K elements of the array sorted till Kth element.

b. Find the worst-case running times of these algorithms as functions of K and N.
   Ans: Selection Sort: O(KN)
   Insertion Sort: O(N^2)
   Merge Sort: O(N log(N))
   Heap Sort: O(N log(N) + K log(N)) = O(N log(N))

PROBLEM 2.

Give an O(N · log(K)) time algorithm to merge K sorted lists into one sorted list, where N is the total number of elements in all the input lists. (Hint: Use a heap for K-way merging.)

Ans: As there are K sorted arrays and N is the total number of elements. We would assume that

each sorted list has N/K elements.
And we pass a matrix containing the K sorted arrays each having N/K elements (assumed) as an input to the algorithm.
We would use MINHEAP for the merging the K sorted lists.
Let each node in the heap contains the value and the index for the sorted list to which the corresponding value belongs to and index of the value in that list.

```
ADD_TO_HEAP(X, H, I, J) {
        N = Put new node where it goes
        Put X into N.VAL
        Put I into N.ListIndex
        Put J into N.ValIndex
        While( N.VAL < PARENT(N).VAL){
                SWAP(N, PARENT(N))
                N = PARENT(N)
                IF(N == ROOT) RETURN
        }
}

DELETEMIN(H) return R{
        Small = Root(H);
        Root.VAL = Last.VAL;
        Delete Last;
        N = Root;
        While(N is not a leaf){
                C = SmallerChild of N;
                IF(C.VAL < N.VAL){
                        SWAP(C,N);
                        N = C;
                }
        }
        R = Small;
}

Merge_K_Lists( A[1..K][1..N/K]) return B[1..N]{

        For( I = 1 to K){                               -------------------------       O(K)
                ADDHeap(A[I][1].value, H, I, 0);        -------------------------       O(Log K)
        }

        For(J = 1 to N){                                -------------------------       O(N)
                M = DELETEMIN(H);                       -------------------------       O(Log K)
                B[J].VAL = M.VAL;
                If(J <= (N – K)){
                  ListIndex = M.ListIndex;
                  ValIndex = M.ValIndex + 1;
                  ADDHeap(A[ListIndex][ValIndex].value, H, ListIndex, ValIndex); ---- O(Log(K))
                }
        }
}
```

The worst case complexity for the algorithm is O(K log(K) + N log(K) + (N – K) log(K))
= O(2N log (K)) = O (N log(K)).

The algorithm can also be implemented using an array of pointers to each array and the elements of the array being the nodes in the heap. And increment the pointer for each sorted list as an element of the same gets deleted. With this we will have to make the assumption that each list contains N/K elements.

PROBLEM 3.

(Siegel 5.23) Consider the following sorting problem. The input is a sequence of N integers with many duplications, such that the number of distinct integers in the sequence is O(log(N)). Design a sorting algorithm to sort such sequences using at most O(N log log N) comparisons in the worst case.

Ans:

```
BinarySearch(X;A) return POS{
        L = 1;
        U = |A|;
        Loop While (U > L){
                M = (L + U)/2
                If (A[M] == X)
                        Return M;
                If (A[M] < X) L = M + 1;
                Else U = M – 1;
        }
        Return 0;
}

Sort(A[1…N]){
        B[1..log(N)] // array of to store the distinct values
        C[1..log(N)] // array to the store the frequency of the element at the corresponding index in
                    // array B
        K = 0 // last index for B
        For(I = 1 to N){                  -------------------------------------------- O(N)
                If(B.Length == 0]{
                        B[I].val = A[I];
                        C[I].freq = 1;
                        Increment K;
                }
                Else{
                        P = BinarySearch(A[I],B);          --------------------- O(Log Log (N))
                        If(P == 0){
                                B[K].val = A[I];
                                C[K].freq = 1;
                                Increment K;
                        }
                        Else{
                                C[P].freq = C[P].freq + 1;
                        }
                }
        }
        MERGESORT(B);                                      --------------------- O(Log (N) Log Log (N))
        L = 1;
        For(J = 1 to log(N)){
                For(q = 1 to C[J]){
                        A[L] = B[J];
                        Increment L;
                        }
        }
}
```

The worst case complexity for the above algorithm would be O(N Log Log (N) + Log(N) Log Log(N) + N) = O(N Log Log N).

PROBLEM 4.

(Siegel 5.24) Design an efficient algorithm to determine whether two unsorted sets of M and N integers are disjoint. Assume that M < N. Full credit will be given for an algorithm that runs in time O(N log M); half credit will be given for an algorithm that runs in time O(N log N + M log M) (which is the same thing as O(N log N).

Ans:

```
CHECK_DISJOINT(A[1..M], B[1..N]) Return Bool{

        MERGESORT(A);               ----------------------------------      M log(M)
        For( I = 1 to N){                                -------------      N
                If ( BINARYSEARCH (B[I]; A) ) {         --------------      log(M)
                        Return False;
                }
        }
        Return True;
}
```

The worst case running time for this algorithm is O(M log(M) + N log(M)) = O(N log (M)).