HW 5

Due on **Wednesday, February 27, 2019 at 10am**

CS/ECE 374: Algorithms & Models of Computation, Spring 2019 Version: **1.2**

Submission instructions as in previous homeworks.

13 (100 PTS.) Not a sorting network.

You are given an array $\texttt{A}[1 \ldots n]$, but you can not access it directly (or even read the values in it, or compare them directly). Instead, you are give some procedures that can access the array and do certain operations. Your task is to sort the array.

13.A. (30 PTS.) You are given a procedure **sortBlock**(i), which sorts (in increasing order and in place), all the elements in $A[i \ldots, i+u]$, where $u \geq 1$ is a prespecified fixed parameter (i.e., u is a known fixed number between 1 and n, but it is not under your control). For example, for the following array:

2.71828	18284	59045	23536	02874	7.1352	66249
1	2	3	4	5	6	7

With $u = 3$, the call **sortBlock**(2) would result in:

2.71828	02874	18284	23536	59045	7.1352	66249
1	2	3	4	5	6	7

Describe an algorithm, that uses $O\big((n/u)^2\big)$ calls to **sortBlock**, and sorts the array.
What is the running time of your algorithm if calling **sortBlock** takes $O(1)$ time?

13.B. (50 PTS.) Congratulations! You just got a better sorting primitives **bMerge**, **copy**, and a work array $W[1, \ldots, n]$.

(i) **copy** can copy any block of length at most $u+1$ between the two arrays (or inside them).

(ii) **bMerge** is weirder. It takes two blocks L and U (both with at most $u+1$ elements), treat them as a single block, sort the unified block, and writes the smaller $|L|$ elements (in sorted order) to L, and the other (larger) $|U|$ elements in sorted order to U (the two blocks do not have to be of the same length).

For example, for

2.71828	18284	59045	23536	02874	7.1352	66249
1	2	3	4	5	6	7

With $u = 1$, the call **bMerge**$(A[2 \ldots 3], A[6 \ldots 7])$ would result in:

2.71828	7.1352	18284	23536	02874	59045	66249
1	2	3	4	5	6	7

bMerge also returns the number of elements in original block L that are still in L after the operation is completed. In this example, since 18284 was in L before the operation, and it is in L after the operation is completed, the returned value would be 1.

Note that the blocks given to **bMerge** can not overlap.
Assume that $A[1 \ldots n/2]$ and $A[n/2+1, \ldots n]$ are already sorted (n is even). Describe an algorithm that performs a minimal total number of calls to **sortBlock**, **copy** and **bMerge**, and sorts the array A. What is the running time of your algorithm if calling **sortBlock**, **copy** and **bMerge** takes $O(1)$ time? (Prove your bound.)

13.C. (20 PTS.) Building on (B) and expanding on it, describe a sorting algorithm using these primitives that sort the given array A (that is initially not sorted). What is the running time of your algorithm if calling **sortBlock**, **copy** and **bMerge** takes $O(1)$ time? Naturally, the faster the better (Prove your bound).

14 (100 PTS.) Not a sorting question.

Consider an array $A[0 \,..\, n-1]$ with n distinct elements. Each element is an ℓ bit string representing a natural number between 0 and $2^\ell - 1$ for some $\ell > 0$. The only way to access any element of A is to use the function **FetchBit**(i, j) that returns the jth bit of $A[i]$ in $O(1)$ time.

14.A. (20 PTS.) Suppose $n = 2^\ell - 1$, i.e. exactly one of the ℓ-bit strings does not appear in A. Describe an algorithm to find the missing bit string in A using $\Theta(n \log n)$ calls to **FetchBit** without converting any of the strings to natural numbers.

14.B. (40 PTS.) Suppose $n = 2^\ell - 1$. Describe an algorithm to find the missing bit string in A using only $O(n)$ calls to **FetchBit**.

14.C. (40 PTS.) Suppose $n = 2^\ell - k$, i.e. exactly k of the $l-$bit strings do not appear in A. Describe an algorithm to find the k missing bit strings in A using only $O(n \log k)$ calls to **FetchBit**.

15 (100 PTS.) Don't want to walk too much.

You are given a set of n distinct points on a line with x-coordinates $x_1, x_2, \,..\, x_n$. The points are not sorted and their values are stored in an array X where $X[i] = x_i$. Each point is associated with a positive weight w_i such that $\sum w_i = 1$. The weights are also stored in an array W where $W[i] = w_i$. Our goal is to find x_j that minimizes the weighted distance given by: $\sum_i w_i |x_j - x_i|$.

15.A. (10 PTS.) (Easy.) Show that if all the weights are equal, x_j is the median of X.

15.B. (20 PTS.) In the general case, show that x_j is the point that satisfies the following property:

$$\sum_{x_i < x_j} w_i < \frac{1}{2} \qquad \text{and} \qquad \sum_{x_i > x_j} w_i \leq \frac{1}{2}.$$

15.C. (20 PTS.) Given, X and W, describe in few lines an algorithm to find x_j. What is the running time of your algorithm. (Your algorithm for this part should be simpler than the algorithm for the next part.)

15.D. (50 PTS.) Given, X and W, describe an algorithm to find x_j in $O(n)$ time. Prove the bound on the running time of your algorithm.