1 Group Sorting (66 points)

We say that an array $A$ of size $n$ is $k$-group sorted if it can be divided into $k$ consecutive groups, each of size $n/k$, such that the elements in each group are larger than the elements in earlier groups, and smaller than elements in later groups. The elements within each group need not be sorted.

For example, the following array is 4-group sorted:

\[
\begin{bmatrix}
1 & 2 & 4 & 3 & 7 & 6 & 8 & 5 & 10 & 11 & 9 & 12 & 15 & 13 & 16 & 14
\end{bmatrix}
\]

Note that every array is 1-group-sorted, and only sorted arrays are $n$-group sorted. For the rest of this problem we will only care about deterministic algorithms (and lower bounds against deterministic algorithms). You may assume that all elements are distinct, and if you want to you may assume that $n$ and $k$ are powers of 2.

(a) Describe an algorithm that $k$-group-sorts an array in $O(n \log k)$ (i.e., in at most $O(n \log k)$ time it must turn an array which is not $k$-group sorted into one that is). Prove correctness and running time.

Solution. We use the following algorithm:

1: \textbf{procedure} \textsc{Group-Sort}(A, k)
2: \hspace{1em} Let $n = |A|$
3: \hspace{1em} If $k = 1$ return $A$
4: \hspace{1em} $p \leftarrow \text{BPFRT}(A, (n/k)\lceil k/2 \rceil)$
5: \hspace{1em} Compare all elements to $p$: Let $L$ be all elements of $A$ that are at most $p$ (including $p$), let $G$ be all elements that are larger than $p$
6: \hspace{1em} $L' \leftarrow \text{Group-Sort}(L, \lfloor k/2 \rfloor)$
7: \hspace{1em} $G' \leftarrow \text{Group-Sort}(G, \lfloor k/2 \rfloor)$
8: \hspace{1em} Return $(L', G')$
9: \textbf{end procedure}

To prove correctness, we proceed by induction. For the base case, suppose that $k = 1$. Then the algorithm returns $A$ itself, which is indeed 1-group-sorted. For the inductive step, consider some $k$. By the correctness of BPFRT, $p$ is the $(n/k)\lceil k/2 \rceil$th smallest element of $A$, and thus by the construction of $L$ and $G$ we know that $L$ consists of the $(n/k)\lceil k/2 \rceil$ smallest elements
and $G$ consists of the largest $(n/k)\lceil k/2 \rceil$ elements. Thus everything in $G$ is greater than everything in $L$. By induction, $L'$ is $\lceil k/2 \rceil$-group-sorted, and so consists of $\lceil k/2 \rceil$ groups each of size $n/k$ so that the elements in each group are all larger than all elements of earlier groups. Similarly, by induction $G'$ is $\lceil k/2 \rceil$-group-sorted, and so consists of $\lceil k/2 \rceil$ groups each of size $n/k$ so that the elements in each group are all larger than all elements of earlier groups. Since everything in $G$; is larger than everything in $L'$, this implies that $(L', G')$ is $k$-group sorted.

To analyze the running time, let $T(n, k)$ be the worst-case running time on an array of size $n$. Then $T(n, 1) = c$ for some constant $c$, since we can just return $A$. If $k > 1$, then line 4 takes $O(n)$ time (by the running time of BPFRT), and by definition line 6 takes $T((n/k)\lceil k/2 \rceil, \lceil k/2 \rceil)$ time and line 7 takes $T((n/k)\lceil k/2 \rceil, \lceil k/2 \rceil)$ time. Thus the running time is

$$T(n, k) = T((n/k)\lceil k/2 \rceil, \lceil k/2 \rceil) + T((n/k)\lceil k/2 \rceil, \lceil k/2 \rceil) + cn.$$

If we for simplicity forget the ceilings and floors (think of $n$ and $k$ as powers of 2), then this is $T(n, k) = T(n/2, k/2) + T(n/2, k/2) + cn$, and now a simple induction proves that $T(n, k) \leq cn \log k$, since we have that $T(n, k) = 2T(n/2, k/2) + cn = 2c(n/2) \log(k/2) + cn = cn(\log k - 1) + cn = cn \log k$ as claimed.

(b) Prove that any comparison-based $k$-group-sorting algorithm requires $\Omega(n \log k)$ comparisons in the worst case.

**Solution.** We will assume without loss of generality that $k \geq 4$, since if $k \leq 4$ then $\Omega(n \log k) = \Omega(n)$ and it is trivial to see that $n - 1$ is a lower bound on the number of comparisons (or else there will be an element that is not compared at all, and so we cannot be sure that we have correct $k$-group-sorted).

Note that an equivalent definition to $k$-group-sorted is that the $n/k$ smallest elements are in the first group (in any order), then the next $k/2$ smallest elements are in the next group (in any order), etc. So consider the following problem: given an array, for every $i \in [n/k]$ identify which of the $n/k$ elements are in the $i$’th group. By using the same “tagging” trick as in lecture, it is easy to see that if we can solve the $k$-group-sorting problem in $x$ comparisons then we can also solve this problem in $x$ comparisons. So we will prove a lower bound for this problem.

Each comparison we make has two possible outcomes, so any algorithm for this problem can be modeled as a binary decision tree. The number of leaves is the the number of possible answers. This is a little trickier to calculate, but isn’t too bad to bound: there are $n/k$ choices out of the original $n$ positions for the first group, then $n/k$ choices out of the $n(1 - 1/k)$ remaining for the second group, then $n/k$ choices out of the $n(1 - 2/k)$ remaining for the third group, etc. Hence the number of leaves (number of possible valid solutions) is

$$\prod_{i=0}^{k-1} \left( \begin{array}{c} n(1 - i/k) \\ n/k \end{array} \right) \geq \prod_{i=0}^{k/2} \left( \begin{array}{c} n(1 - i/k) \\ n/k \end{array} \right) \geq \prod_{i=0}^{k/2} \left( \begin{array}{c} n/2 \\ n/k \end{array} \right) \geq \prod_{i=0}^{k/2} \left( \frac{n/2}{n/k} \right)^{n/k} \geq \prod_{i=1}^{k/2} \left( \frac{k}{2} \right)^{n/k} = \left( \frac{k}{2} \right)^{n/2}.$$
Since that is a lower bound on the number of leaves and the decision tree is binary, the depth of the tree (and thus the number of comparisons in the worst-case) must be at least
\[ \log \left( \left( \frac{k}{2} \right)^{n/2} \right) = \frac{n}{2} \log \left( \frac{k}{2} \right) = \Omega(n \log k) \]

(c) Describe an algorithm that completely sorts an already \( k \)-group-sorted array in \( O(n \log(n/k)) \) time. Prove correctness and running time.

**Solution.** Consider the following simple algorithm: for each group, we use mergesort to sort it.

**Correctness:** We need to prove that the array \( A \) is sorted after doing this. Consider two elements \( i < j \) in \( A \). If \( i \) and \( j \) were in different groups then the group of \( i \) must have been before the group of \( j \) (by the definition of \( k \)-group-sorted), so after running mergesort on each group it is still true that \( i \) is before \( j \) in \( A \). On the other hand, if \( i \) and \( j \) were in the same group then the correctness of mergesort implies that \( i \) is before \( j \) in \( A \). Thus \( i \) and \( j \) are in the correct order, and so \( A \) is sorted.

**Running Time:** We know that mergesort takes time \( O(n \log n) \) on an array of size \( n \). We run it \( k \) times on arrays of size \( n/k \), and hence the total running time is \( O(k \cdot (n/k) \log(n/k)) = O(n \log(n/k)) \) as claimed.

(d) Prove that any comparison-based algorithm to completely sort a \( k \)-group-sorted array requires \( \Omega(n \log(n/k)) \) comparisons in the worst case.

**Solution.** As usual, each comparison has two possible outcomes and thus any algorithm can be modeled as a binary decision tree. Since we want to end up with a fully sorted list, as in class every leaf is a permutation. However, since the array we start with is already \( k \)-group-sorted, not every permutation is possible. By definition of \( k \)-group-sorted we know that everything in a group belongs somewhere in one of those \( n/k \) places for the group: we just don’t know the permutation inside of each group. Hence the number of leaves (number of possible permutations) is
\[ \prod_{i=1}^{k} (n/k)! = ((n/k)!)^k. \]

This implies that the depth (worst-case number of comparisons) is at least
\[ \log \left( ((n/k)!)^k \right) = k \log((n/k)!) = \Theta(k(n/k) \log(n/k)) = \Theta(n \log(n/k)) \]

2 Sorting Different-Length Items (34 points)

We saw in class some fast sorting algorithms where we assumed that the elements were integers with the same number of digits. In this problem we change those assumptions slightly.
(a) Suppose we are given an array of integers, but instead of all integers having the same length they can each have a different number of bits. So, e.g., the number 0 or 1 takes one bit, the numbers 2, 3 take 2 bits, the numbers 4, 5, 6, 7 take three bits, etc. However, the total number of bits over all of the integers in the array is equal to \( n \). Show how to sort the array in \( O(n) \) time (prove correctness and running time).

Solution. Let \( k \) be the maximum number of digits of any of the input integers (which can clearly be computed in \( O(n) \) time). We can use the following algorithm to sort the array of integers:

1. Divide the array of integers into buckets \( B_1, B_2, B_3, \ldots, B_k \), where \( B_i \) contains all the numbers with \( i \) digits.
2. Sort each \( B_i \) using radix sort.
3. Concatenate the sorted \( B_i \)'s in order from 1 to \( k \).

For correctness, note that for any \( i < j \), every number in \( B_i \) is less than all the numbers in \( B_j \). Since the radix sort correctly sort the integers in every bucket, the concatenation of \( B_1, B_2, \ldots, B_n \) remains sorted.

To analyze the running time, observe that Steps 1 and 3 take \( O(n) \) time each because the number of integers is at most the number of total digits. For Step 2, let the number of integers in bucket \( B_i \) be \( n_i \). For bucket \( B_i \) the radix sort runs in \( O(in_i) \) time (using the simple version of radix sort, where for each of the \( i \) digits we do a single bucket sort, which has 2 buckets because the numbers are in base 2). Therefore Step 2 runs in \( \sum_{i=1}^{k} O(in_i) = O\left(\sum_{i=1}^{k} in_i\right) \). Since the total number of bits over all of the integers in the array is equal to \( n \), we have \( \sum_{i=1}^{k} in_i = n \). Hence the total running time is \( O(n) \).

(b) Suppose now that we are given an array of strings (over some finite alphabet, say the letters \( \text{a-z} \)). Each string can have a different number of characters, but the total number of characters in all the strings (i.e., the sum over the strings of the length of the string) is equal to \( n \). Show to sort the strings lexicographically in \( O(n) \) time. Here lexicographic order is the standard alphabetic order, so for example \( \text{a} < \text{ab} < \text{b} \).

Solution. In this case, we can just use a slight modification of ordinary radix sort. Specifically, we'll use the version where we start by sorting the most significant digit, and then recurse on the buckets. We'll assume that all the strings are “left-aligned”, and we add a end-of-string character to each string. (i.e. If \( s=\text{“string”} \), then \( s[1]=\text{‘s’}, s[4]=\text{‘i’}, \) and \( s[7]=0 \).)

The algorithm is as follows:
sort_strings(S, d):
    create 27 string lists B[0], B['a'], ..., B['z']
    for each string s in S:
        put s in B[s[d]]
    for 'a' ≤ i ≤ 'z':
        if B[i] is not empty:
            B[i] = sort_strings(B[i], d+1)
            concatenate B[i] at the end of B[0]
    return B[0]

As you can see, this is just ordinary radix sort, except that we use an extra bucket to keep track of which strings have ended, and we don’t recurse on that bucket. Here is an example of the bucketing performed by a run of this algorithm:

We can prove the correctness by induction on d. We claim that the algorithm returns a sorted string list of S if all the strings have the same first d − 1 characters. The base case is d = ℓ, which is obviously true because only the last characters of the strings can be different. Assume the claim holds for d + 1 and we consider d. Since all the buckets B[i] are sorted by inductive assumption, and B[0] can only have one string initially. Concatenating B[0], B['a'], ..., B['z'] gives us a sorted string list.

For each string s, we will compare, move, and concatenate it for at most length(s) + 1 times, because we visit each character (including the end-of-string character) of s only once. Since length(s) + 1 ≤ 2 · length(s) for all s, the algorithm runs in time $O(\sum_s \text{length}(s)) = O(n)$. 

5