CS 624: Analysis of Algorithms
Assignment 2
Due: Wednesday, October 4, 2023

1. Show that \( \sum_{k=0}^{n} \binom{n}{k} k = n2^{n-1} \). Hint: use a generating function and differentiate.

2. (a) Show how to implement a stack (Last in first out) using a Priority Queue.
   (b) Show how to implement a queue (First in first out) using a Priority Queue.
   In both cases, do not worry about an \( O(1) \) run time.

3. Describe an \( O(n \log k) \) algorithm for merging \( k \) sorted lists into one sorted list, where \( n \) is the total number of elements. **Hint:** Think of the merging part of MergeSort and extend it to multiple lists. Remember that the lists are not necessarily the same size.

4. The procedure \( \text{Max-Heap-Delete}(A, i) \) deletes the item in node \( i \) from heap \( A \). Give an implementation of \( \text{Max-Heap-Delete} \) that runs in \( O(\log n) \) for an \( n \)-element Max-Heap. Assume that the heap elements are mapped into indices, so you have access to the \( i^{th} \) node. Note that the problem asks you to give an algorithm that runs in \( O(\log n) \) time. So you not only have to give the algorithm, you also have to show that it really does run in \( O(\log n) \) time.

5. Suppose you start with a rectangular array of numbers. Perform the following operations:
   - First sort each row (smallest to largest).
   - Then sort each column (smallest to largest).

   Show that after sorting the columns, each row is still sorted. (Hint: Prove by contradiction)

6. Exercise 3.1 in the Lecture 3 handout (on page 7 of the handout). Don’t be sloppy here! I’m looking for a precise explanation.

7. Exercise 6.1 in the Lecture 3 handout (on page 13 of the handout).

8. Exercise 3.1 from the Lecture 4 handout (page 7).

9. Show that there is no comparison-based sort whose running time is linear for at least half of the \( n! \) inputs of length \( n! \). What about a fraction of \( 1/n \) of the inputs of length \( n! \)? What about a fraction of \( 1/2^n \)? Be careful here. The only hard part of this problem is understanding exactly what it is asking. Once you understand that, the solution should be pretty short. Here’s some help in understanding what the problem is asking: "The \( n! \) inputs" is just the book’s way of referring to the \( n! \) possible permutations of the \( n \) input numbers. Each of those permutations is thought of as one input to the sorting algorithm we are considering. We proved in class that any decision tree modeling a sorting algorithm for \( n \) numbers would have depth at least \( Cn \log n \) (for some \( C \)). This was based on the fact that it had to have \( n! \) leaves (each leaf corresponding to one of the "inputs").
But that’s only the worst case. Maybe it’s the case that there is some comparison sort (modeled by a binary decision tree) which has the property that even though the tree (necessarily) has maximum depth $Cn \log n$, still there is some constant $A$ (independent of $n$) such that half the leaves are at depth $A n$ or less. The first question in the problem is asking if this is possible. (And there are two other questions in the problem as well.)


11. Assume that $c \geq 0$. Assume you had some kind of super-hardware that, when given two lists of length $n$ that are sorted, merges them into one sorted list, and takes only $n^c$ steps.

(a) Write down a recursive algorithm that uses this hardware to sort lists of length $n$.

(b) Write down a recurrence to describe the run time.

(c) For what values of $c$ does this algorithm perform substantially better than $O(n \log n)$? Why is it highly implausible that this kind of super-hardware could exist for these values of $c$?