CS 624: Analysis of Algorithms

Assignment 2

Due: Tuesday, October 8, 2024

Topics: Heaps, Sorting

1. The Split3-Sort algorithm is defined as follows:

Algorithm 1 Split3-Sort(A,p,r)

```
1: if (A[p] > A[r]) then
2:    Swap A[p] with A[r]
3: end if
4: if (p + 1 < r) then
5:    k = ⌊(r − p + 1)/3⌋ // Round down
6:    Split3 − sort(A, p, r − k) // First two thirds
7:    Split3 − sort(A, p + k, r) // Last two thirds
8:    Split3 − sort(A, p, r − k) // First two thirds again
9: end if
```

   (a) Prove that the call to Split3-Sort(A, 1, n) correctly sorts the array $A[1..n]$ (**Hint:** I found the best way is to use induction, but be careful with the base case - notice line 4. What is the minimum difference between p and r?)

   (b) Write the recurrence formula for Split3-Sort and give the asymptotic bound on the run time (Θ notation).

   (c) Compare the run time from (b) to the run time of HeapSort, MergeSort and QuickSort. Is it better? Worse? Same?

2. Let $\{f_n : n = 0, 1, \dots\}$ be the Fibonacci sequence (where by convention $f_0 = 0$ and $f_1 = 1$).

   (a) This question is based on material from lecture notes 2. Show that $\sum_{n=1}^{\infty} \frac{nf_n}{2^{n-1}} = 20$. Do this by using a generating function as shown in the last section of the Lecture 2 notes, and differentiating. **Hint:** The derivative of $\frac{x}{1-x-x^2}$ is $\frac{1+x^2}{(1-x-x^2)^2}$.

   (b) Show why (in the same way as you proved the first part of this problem) you might think that $\sum_{n=1}^{\infty} nf_n = 2$. Then show why this could not possibly be true (it doesn't have to be a long answer, but it has to be convincing).

3. Argue the correctness of HEAPSORT using the following loop invariant:

   At the start of each iteration of the for loop of lines 2–5, the subarray $A[1..i]$ is a max-heap containing the i smallest elements of $A[1..n]$ and the subarray $A[i + 1..n]$ contains the n-i largest elements of $A[1..n]$ sorted.

4. 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.

5. The procedure `Max-Heap-Delete(A,i)` deletes the item in node i from heap A. Give an implementation of `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.

6. Show that there is an algorithm that produces the k smallest elements of an unsorted set of n elements in time O(n + k log n). Be careful: To do this problem correctly, you have to state the algorithm carefully and prove that it does what it is supposed to do. (The proof can be very simple.) Then you have to prove that the algorithm runs in time $O(n + k \log n)$.

7. Based on exercises 8.2-1 - 8.2-3 in the 4^{th} edition (slightly edited).

   (a) Show the run of counting-Sort on the array $A = [6, 0, 2, 0, 1, 3, 4, 6, 1, 3, 2]$. No need to draw out every stage. Just show the array C after line 5, C after line 8 and the first three stages of the final sorting (similar to figure 8.2)

   (b) Prove that Counting-Sort is stable. In other words, equal elements appear in the sorted array in the same order as they were in the original input – if we have two equal elements $A[i]$ and $A[j]$ such that $i < j$, $A[i]$ will appear before $A[j]$ in the sorted array.

   (c) Show that if we rewrite the for loop header in line 11 of the Counting-Sort pseudo code as `for i=1 to n` (going from the start to the end), the algorithm still works properly, but it is not stable. Then rewrite the pseudocode for counting sort so that elements with the same value are written into the output array in order of increasing index and the algorithm is stable.

8. Based on a question from the textbook: You have n black buckets and n white buckets, all of different shapes and sizes. All the black buckets hold different amount of water, as do the white buckets. You cannot tell from the shape or size how much water each bucket can hold. For every black bucket there is a white bucket that can hold the same amount of water. Your task is to group the buckets into pairs of black and white buckets that hold the same amount of water. The only operation you are allowed to do is to pick a pair of buckets, one black and one white. Fill the black bucket and pour the water from the black to the white bucket. This way you can tell whether the black bucket has a later volume, smaller volume or equal volume to the white bucket. You may not directly compare two black or two white buckets.

   (a) Describe a $O(n^2)$ algorithm that groups the buckets into pairs.

   (b) Describe a probabilistic algorithm that uses an expected $O(n \log n)$ number of comparisons to group the buckets into pairs (hint: It's a bit similar to randomized-quicksort). What is the worst case number of comparisons? Prove your answers.