BİL 031/133 Combinatorics and Graph Theory

HOMEWORK 6 (40 Points)

Due Date: August 7, 2020

1 [6 POINTS] FINDING MODE OF A UNIMODAL SEQUENCE

Assume that you are given an array $A[1 \dots n]$ of distinct numbers. You are told that the sequence of numbers in the array is *unimodal*, in other words, there is an index i such that the sequence $A[1 \dots i]$ is increasing ($A[j] < A[j+1]$ for $1 \leq j < i$), and the sequence $A[i \dots n]$ is decreasing. The index i is called the *mode* of A.

[3 POINTS] Give an $O(\log n)$ algorithm that find the *mode* of A.

[3 POINTS] Prove your algorithm (correctness and termination). Show that your algorithm achieves the requested time bound.

2 [6 POINTS] DIVIDE-AND-CONQUER PARADIGM

Given an array A of n integers, we want to determine the minimum and maximum elements of A. So, you are going to design an algorithm, which takes an array A of size n (and possibly other parameters) as input, and returns a pair (min, max) of 2 integers. In this problem, our concern is not the asymptotic behavior of the run-time of the algorithm, but the **number of comparisons made by your algorithm**.
A naive iterative algorithm can find the maximum and minimum elements by making two passes over the array. In the first pass it will make $(n-1)$ comparisons and find the maximum element, and in the second pass it will find the minimum element by making another $(n-1)$ comparisons. And then it will return the values of these two elements. This naive algorithm makes a total of $2(n-1)$ comparisons. **You will design a recursive algorithm that follows the divide-and-conquer paradigm, and substantially beats the naive algorithm.**

Before getting started, think a bit on the base cases. How many comparisons do you need to make if $n = 1$? What if $n = 2$?

- (2 Points) Design a recursive algorithm for finding the minimum and maximum elements of the array. Recall that a recursive algorithm calls itself unless the instance size is sufficiently small (base case). You should treat both $n = 1$, and $n = 2$ as your base cases.
- (2 Points) Let $T(n)$ denote the number of comparisons made by your algorithm, in the worst case, for an array of size n. Write a recurrence for $T(n)$. In order not to deal with floor and ceiling functions, you can assume that $n = 2^i$ for some positive integer i. Thus, you can take $n = 2$ as the base case of your recurrence.
- (2 Points) Prove that the solution to your recurrence is $T(n) = \frac{3 \cdot n}{2} - 2$ by mathematical induction. (If we did not make simplifying assumptions, the solution of the recurrence would be $T(n) = \lceil \frac{3 \cdot n}{2} \rceil - 2$.)

3 [8 Points] Devil is in the Details

In class, we have seen how to build a max-heap in $\Theta(n)$-time by using the BUILD-MAX-HEAP algorithm. The pseudocode for the BUILD-MAX-HEAP algorithm and its helper procedure MAX-HEAPIFY are provided below for your convenience.

```
BUILD-MAX-HEAP(A)
    heap-size[A] = length[A];

    for i = ⌊length[A]/2⌋ downto 1
        MAX-HEAPIFY(A, i);


MAX-HEAPIFY(A, i)
    l = LEFT(i);
    r = RIGHT(i);

    if l ≤ heap-size[A] and A[l] > A[i]
        largest = l;
    else
        largest = i;

    if r ≤ heap-size[A] and A[r] > A[largest]
        largest = r;

    if largest ≠ i
        exchange A[i] ↔ A[largest];
        MAX-HEAPIFY(A, largest);
```

In this question, you are given a different algorithm to build a max-heap, namely BUILD-MAX-HEAP-2 and asked to analyze it and state its differences from the BUILD-MAX-HEAP. The pseudocode for BUILD-MAX-HEAP-2 is given below.

```
BUILD-MAX-HEAP-2(A)
    heap-size[A] = 1;

    for i = 2 to length[A]
        MAX-HEAP-INSERT(A, A[i]);


MAX-HEAP-INSERT(A, key)
        heap-size[A] = heap-size[A] + 1;
        A[heap-size[A]] = − ∞;
        HEAP-INCREASE-KEY(A, heap-size[A], key);

    HEAP-INCREASE-KEY(A, i, key)
        if key < A[i]
            error "new key is smaller than current key";

        A[i] = key;
        while i > 1 and A[PARENT[i]] < A[i]
            exchange A[i] ↔ A[PARENT[i]];
            i = PARENT[i];
```

- **[4 Points]** Do the procedures BUILD-MAX-HEAP and BUILD-MAX-HEAP-2 always create the same heap when run on the same input array? Prove that they do, or provide a counterexample.
- **[4 Points]** Give a tight upper bound for the worst case asymptotic running time of the BUILD-MAX-HEAP-2.

4 [4 POINTS] O VS O

You are given two *similar looking* propositions below. However, one of them is true, and the other one is false. You are asked to prove the true one, and disprove the false one.
Let $f(n)$ and $g(n)$ be two functions defined over the set of positive integers such that $\lg f(n)$ and $\lg g(n)$ are strictly increasing.

- [2 **Points**] Prove that if $f(n) = O(g(n))$, then $\lg(f(n)) = O(\lg(g(n)))$.
- [2 **Points**] Disprove that if $f(n) = o(g(n))$, then $\lg(f(n)) = o(\lg(g(n)))$.

5 [16 POINTS] EFFICIENT ALGORITHM DESIGN FOR SPECIAL CASES OF SUBSET SUM

Subset Sum is a classical problem in computer science. In this problem, you are given a multiset (array) A of integers, and an integer k. The problem is to determine whether there exists a subset S of the integers in A such that the sum of the integers in S is exactly k.

[0 Points] Try to come up with an efficient (polynomial-time) algorithm for the **Subset Sum** problem. Never mind, if you could not, since nobody achieved to do that yet! This problem is **NP**-complete, and does not admit an efficient algorithm unless **P** = **NP**.

In this question, we will examine special cases of the **Subset Sum** problem. Specifically we will consider the cases where $|S|$ is bounded.

Consider the special case of the **Subset Sum** problem, where $|S| = 1$. In this special case, we are given an array A of integers, and an integer k, and asked to decide whether there exists a subset $|S| = 1$ of integers in A such that the sum of the integers in S is exactly k. Since $|S| = 1$, S is composed of just one integer. Thus, the problem is equivalent to checking whether k is one of the integers in A or not. This can be done by using the **linear search** algorithm in time **O(n)**.

[2 Points] Give an $O(n^2)$-algorithm for the special case of the Subset Sum problem, where $|S| = 2$, i.e., your algorithm should determine whether there exist integers i and j in A such that $i + j = k$.

Notice that you can immediately obtain an algorithm for the special case of the Subset Sum problem, where $|S| \leq 2$, that runs in $O(n^2)$-time. The algorithm should first make a call to the linear search algorithm, and then a call to the algorithm you designed. If any of them returns true, it should return true. This algorithm will be correct only if the algorithm you designed is correct. Thus, you need to prove your algorithm before we proceed.

[3 Points] Prove your $O(n^2)$-algorithm for the special case of the Subset Sum, where $|S| = 2$. You need to show that your algorithm terminates and returns a correct answer for all possible instances of the problem.

[3 Points] Let us now generalize your algorithm to handle the cases, where $|S|$ is larger. Show that for any integer i, you can give an $O(n^i)$-algorithm for the special case of the Subset Sum problem, where $|S| = i$.

Since, for any i, you can give a $O(n^i)$ algorithm for the special case of the Subset Sum problem with $|S| = i$, you can immediately give an $O(n^i)$-algorithm for the special case of the Subset Sum problem with $|S| \leq i$. This algorithm will first run the $O(n)$ linear search algorithm, then $O(n^2)$ algorithm you designed for the case $|S| = 2$, then the $O(n^3)$ algorithm for the case $|S| = 3$, ... , and the $O(n^i)$-algorithm for the special case $|S| = i$, and return true, if any of these

algorithms returns true. Notice that the run-time of this algorithm is $\sum_{j=1}^{i} O(n^j) = O(n^i)$. Thus, you have proven the Theorem stated below.

Theorem 1. *For any integer i, there exists an $O(n^i)$-algorithm that decides the special case of the Subset Sum problem, where $|S|$ is bounded above by i.*

Let us now elaborate the special case, where $|S| = 2$. An electrical engineer trying to get a job in computer industry could as well come up with your $O(n^2)$-algorithm. Competitive computer scientists should be able to do a lot better than that.

[3 **Points**] Give an $O(n \cdot \lg n)$-algorithm for the special case of the Subset Sum problem, where $|S| = 2$. (Hint: The nice thing with trying to design an algorithm that runs in $O(n \cdot \lg n)$-time is that sorting comes for free.)

[5 **Points**] Now, we will assume that the array we are given is **already sorted** in ascending order and try to reduce the time-bound to $O(n)$ for the special case $|S| = 2$. Give an $O(n)$-algorithm that takes a sorted (in ascending order) array of integers, and an integer k; and returns true if the sum of the any two distinct elements of the array is k, and returns false otherwise. Recall that you need to prove that your algorithm terminates with the correct output in $O(n)$ time.