**EECS 376: Foundations of Computer Science** **Due 8:00pm, January 24**
**University of Michigan, Winter 2024** **(accepted until 9:59 pm, no credit after)**

# Homework 2

This homework has 7 questions, for a total of 100 points and 8 extra-credit points.
Unless otherwise stated, each question requires *clear*, *logically correct*, and *sufficient* justification to convince the reader.
For bonus/extra-credit questions, we will provide very limited guidance in office hours and on Piazza, and we do not guarantee anything about the difficulty of these questions.
We strongly encourage you to typeset your solutions in LaTeX.
If you collaborated with someone, you must state their name(s). You must *write your own solution* for all problems and *may not use any other student's write-up*.

(0 pts) 0. **Before you start; before you submit.**

(a) Carefully review Sections 1.2-1.3 (Induction for Reasoning about Algorithms) of Handout 1 before starting this assignment, and apply it to the solutions you submit.

(b) If applicable, state the name(s) and uniqname(s) of your collaborator(s).

**Solution:**

(10 pts) 1. **Self assessment.**

Carefully read and understand the posted solutions to the previous homework. Identify one part for which your own solution has the most room for improvement (e.g., has unsound reasoning, doesn't show what was required, could be significantly clearer or better organized, etc.). Copy or screenshot this solution, then in a few sentences, explain what was deficient and how it could be fixed.

(Alternatively, if you think one of your solutions is significantly *better* than the posted one, copy it here and explain why you think it is better.)

**Solution:**

2. **Potential potential functions, for a different sort of sort.**

The following algorithm computes a *topological sort* of a given directed acyclic graph (DAG). In essence, it works as follows; see below for precise pseudocode. Repeatedly do the following until we cannot do so any longer: choose an arbitrary vertex $u$ that has no incoming edges, append $u$ to the output list, and delete $u$ along with all its outgoing edges.

In this problem we are concerned with showing that this algorithm must terminate, using the potential method. (We are not concerned with any other aspects of correctness or running time.)

```
function TopSort($G = (V, E)$)                          ▷ $G$ is a directed acyclic graph
    initialize $i \leftarrow 0$, array $L[1, \dots, |V|]$
    $S \leftarrow \{u \in V : u$ has no incoming edge$\}$
    while $S$ is not empty do
        $S \leftarrow S \setminus \{u\}$ for some arbitrary $u \in S$
        $i \leftarrow i + 1$, $L[i] \leftarrow u$
        for each outgoing edge $(u, v) \in E$ from $u$ do
            $E \leftarrow E \setminus \{e\}$
            if $v$ has no incoming edge then
                $S \leftarrow S \cup \{v\}$
    return $L$
```

(5 pts) (a) Observe that $i$ is initialized to zero, and that each iteration of the while loop increments $i$. For each $i \geq 0$, let $S_i$ denote the corresponding value of the set $S$ at the start of the loop. (So, $S_0$ is the initial value of $S$; $S_1$ is the value of $S$ after one iteration, etc.)

Consider using the cardinality $|S_i|$ as a candidate potential function. Briefly explain why this is *not* a valid choice, i.e., why it does not meet the requirements of a potential function.

**Solution:**

(5 pts) (b) For each $i \geq 0$, let $N_i$ denote the set of vertices that have *not yet* been added to $S$, at the start of the loop for the corresponding value of $i$. Consider using the cardinality $|N_i|$ as a candidate potential function. Briefly explain why this is *not* a valid choice.

**Solution:**

(6 pts) (c) Prove that the algorithm halts on every valid input (no matter what internal choices the algorithm makes), by defining a valid potential function and proving that it meets the needed requirements.

**Solution:**

3. **Euclid's algorithm, extended.**

Given two integers $x, y$ (that are not both zero), one may wonder what values can be obtained by summing integer multiples of $x$ and $y$, i.e., expressed as $ax + by$ for some (not necessarily positive) integers $a, b$. It is not too hard to see that it is imposisble to obtain any positive integer *smaller* than their GCD $g = \gcd(x, y)$, because $ax + by$ must be divisible by $g$. A less obvious fact is that the GCD *can* be obtained in this way; this theorem is known as *Bézout's identity* (pronounced "BAY-zoo").

In this problem, you will modify the standard Euclidean algorithm to output not only $g = \gcd(x, y)$ itself, but also a pair $(a, b)$ of integers for which $ax + by = g$. (As we will see later, this is a very important tool for cryptography.) Such integers are called *Bézout coefficients* for $x, y$. We give most of the algorithm below:

**Input:** integers $x \geq y \geq 0$, not both zero
**Output:** a triple $(g, a, b)$ of integers where $g = \gcd(x, y) = ax + by$

```
function ExtendedEuclid($x, y$)
    if $y = 0$ then
        return $(x, 1, 0)$                    ▷ Base case: $1x + 0y = x = \gcd(x, 0)$
    else
        Divide $x$ by $y$, writing $x = qy + r$ for integer quotient $q$ and remainder $0 \leq r < y$
        $(g, a', b') \leftarrow$ ExtendedEuclid$(y, r)$
        [FOR YOU TO DETERMINE: compute appropriate $a$, $b$]
        return $(g, a, b)$
```

(10 pts) (a) State what $a$ and $b$ should be on Line 7, and prove that the output is correct, i.e., that (1) $g = \gcd(x, y)$, and (2) $ax + by = g$.

*Hint*: by recursion/induction, we know that $g = \gcd(y, r)$ and $a'y + b'r = g$.

**Solution:**

(10 pts) (b) Run the Extended Euclid algorithm by hand to find Bézout coefficients for the input $(x, y) = (295, 204)$, and show that the output is correct. (You may use a calculator/computer only for the division steps.) Fill in the table below with a 'trace' of the execution, i.e., all the variables' values in all the iterations. Also include the potential values $s = x+y$, the ratios (as fractions) of potentials $s_{i+1}/s_i$ for adjacent iterations, and Y/N indications of whether $s_{i+1}/s_i \leq 2/3$.

The entries labeled 'input', 'division', $s$, and $s_{i+1}/s_i$ should be filled from top to bottom, corresponding to the recursive calls; the 'recursive answer' and 'output' entries will need to be filled from bottom to top, corresponding to the 'post-processing' (Line 7) of each recursive call's results. The entries for $a, b$ should match those of $a', b'$ one row above. Put '–' for any entries that are not defined due to the base case.

**Solution:**

| | input | | division | | rec ans | | output | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| $i$ | $x$ | $y$ | $q$ | $r$ | $a'$ | $b'$ | $a$ | $b$ | $s_i$ | $s_{i+1}/s_i$ | $\leq 2/3$ ? |
| 0 | 295 | 204 | ?? | ?? | ?? | ?? | ?? | ?? | ?? | ?? | Y or N |

4. **Divide and conquer, and the Master Theorem.**

For each of the following recursive algorithms, give (with brief justification) a recurrence for the algorithm's running time $T(n)$ as a function of the input array size $n$. State whether the Master Theorem is applicable to the recurrence, and if so, use it to give the closed-form solution; if not, explain why not.

(8 pts) (a)

```
function Func($A[1, \ldots, n]$)
    if $n \le 3$ then
        return 0
    Func($A[1, \ldots, \lfloor n/2 \rfloor]$)
    $i \leftarrow 1$
    while $i < n$
        $i \leftarrow i + 1$
        Func($A[1, \ldots, n-3]$))
    $j \leftarrow 2$
    $z \leftarrow i$
    while $j < i$
        $z \leftarrow z - 1$
        $j \leftarrow j + 1$
    return $i$
```

**Solution:**

(8 pts) (b)

```
function Func($A[1, \ldots, n]$)
    if $n \le 1$ then
        return 0
    $x \leftarrow$ Func($A[1, \ldots, \lfloor n/2 \rfloor]$)
    $y \leftarrow$ Func($A[\lfloor n/2 \rfloor + 1, \ldots, n]$)
    $z \leftarrow$ Func($A[1, \ldots, \lfloor n/2 \rfloor]$)
    if $x = 0$ or $y = 0$ or $z = 0$ then
        result $\leftarrow 0$
        for $i = 1$ to $n$ do
            for $j = 1$ to $n$ do
                result $\leftarrow$ result $+ 1$
        return result
    $h \leftarrow$ MergeSort($A[1, \ldots, \lfloor n/2 \rfloor]$)
    $g \leftarrow$ MergeSort($A[\lfloor n/2 \rfloor + 1, \ldots, n]$)
    return $h[0] - g[0]$
```

**Solution:**

(8 pts) (c)

```
function StoogeSort($A[1, \dots, n]$)
    if $n = 1$ then
        return $A$
    if $n = 2$ and $A[1] > A[n]$ then
        swap $A[1]$ and $A[n]$
    if $n > 2$ then
        $t \leftarrow \lceil 2n/3 \rceil$
        StoogeSort($A[1, \dots, t]$)
        StoogeSort($A[n - t + 1, \dots, n]$)
        StoogeSort($A[1, \dots, t]$)
    return $A$
```

**Solution:**

(3 EC pts) (d) *Optional extra credit:* Prove that StoogeSort from the previous part is a *correct* sorting algorithm.

**Solution:**

5. **Sorting out complaints.**

A group of $n$ students, all of whom have distinct heights, line up in a single-file line for a group photo. Any student who stands somewhere in front of a shorter student will conceal the shorter student, who will not appear in the picture. (The taller student need not be *directly* in front of the shorter one.) For this reason, each student makes one complaint to the photographer for *each* taller student in front of them.

In this problem we are concerned with algorithms that determine the number of complaints that will be made. The input is an array $A[1, \dots, n]$ of the students' heights, from the front of the line to the back. (So, $A[1]$ is the height of the student in the very front, and $A[n]$ is the height of the student in the very back.) The desired output is the total number of complaints. (Throughout this question, assume that two heights can be compared in constant time, independent of $n$.)

(5 pts) (a) Describe briefly, clearly, and precisely (in English) a simple brute-force algorithm for this problem; do not give pseudocode. State, with brief justification, a $\Theta(\cdot)$ bound on its (worst-case) running time as a function of $n$. You do not need to give a formal proof.

**Solution:**

(10 pts) (b) Suppose that both the front half and back half of the line (i.e., the two halves of the array $A$) happen to be sorted in ascending order, though the line as a whole may not be. Describe clearly and precisely (in English or in pseudocode, as you prefer) an $O(n)$-time algorithm that outputs the number of complaints in this scenario, and briefly justify its correctness and running time.

**Solution:**

(15 pts) (c) Give a $O(n \log n)$-time divide-and-conquer algorithm for this complaint-counting problem. Briefly and clearly describe (in English) how the algorithm works, then give clear pseudocode.

*Hint*: Enhance the MergeSort algorithm to both sort *and* count. Think about how sorting the two halves of the array affects, or doesn't affect, whether a specific student will be concealed by a particular other student.

**Solution:**

(5 EC pts) 6. **Optional extra credit: trophy testing.** (This is a good problem for those who want a challenge. It will be graded with high standards and not much partial credit, and no regrades will be done.)

This past weekend, during the National Championship Celebration at Crisler Center, J.J. McCarthy decided to toss the AFCA Coaches' Trophy to Mike Sainristil. Luckily, Mike caught it (as he does with most footballs that come in his vicinity), but the Michigan athletic department was very concerned, since this trophy is made of Waterford Crystal and valued at more than \$34,130. As a result, it has asked researchers at the University's Materials Science and Engineering department to conduct research on Waterford Crystal.

To test its strength, blocks of Waterford Crystal will be dropped from various heights between 1 and $n$ inches (inclusive), for some $n$ of interest. Waterford Crystal is said to have *least breaking height* (LBH) $k$ if a block of it will break when dropped from a height of $k$ inches or more, but will not break if dropped from any fewer number of inches. If it does not break even when dropped from a height of $n$ inches, we say that the LBH is "more than $n$," denoted "$> n$" for convenience. All blocks have the same LBH; the LBH is a property of the material itself, and doesn't vary from block to block.

Your task is to determine the LBH of Waterford Crystal. Since it's a very expensive material, you are given just two blocks. Fortunately, any two blocks of Waterford Crystal have the same LBH. Unfortunately, once a block breaks, it is unusable for future testing.

It is easy to see that if you had only one block, you would have no choice but to drop the block from 1 inch, then from 2, and so on, until the block breaks (or it doesn't even break from a height of $n$ inches). This is because if you were to skip some height, then you would risk prematurely breaking the block without knowing the exact height from which it would have first broken.

But, with two blocks, you can do better! In this problem you will determine the best way to utilize two blocks to determine the LBH, using the fewest number of drops. Instead of expressing the solution as "if I want to determine the LBH among the $n+1$ possibilities, I can find it using at most $d = d(n)$ drops," it will be cleaner to express the solution *inversely*, as: "With $d$ drops, I can determine the LBH among $n(d)+1$ different possibilities," where $n = n(d)$ is a function of $d$. (Recall that "$> n$" is the extra case representing "greater than $n$.")

(a) Suppose you are given two blocks of Waterford Crystal and allowed no more than $d$ drops, from heights of your choice (which can depend on the results of previous drops). What is the largest value of $n = n(d)$ for which you can determine the LBH among the $n+1$ different possibilities? Describe your method clearly and concisely in English, give a recurrence and base case for $n(d)$, and give an exact (not asymptotic) closed-form solution.

Your algorithm should be a recursive divide-and-conquer one, inspired by the following reasoning. When you first drop a block from a certain height, this will effectively divide the problem into two cases: if the block breaks, the LBH is at most that height (and you have only one block left); otherwise, the LBH is strictly larger than that height (and you still have two blocks). So, the first drop serves to divide the problem, and then subsequent drop(s) conquer either lesser heights or greater heights.

**Solution:**

(b) Give a short explanation why your algorithm is *optimal*; i.e., why *any algorithm* that uses 2 blocks and at most $d$ drops cannot be guaranteed to work for a value larger than $n = n(d)$, for the $n(d)$ you gave in the previous part. A few sentences of intuitive but logically valid explanation suffice here, but if you wish, you're welcome to prove this rigorously.

**Solution:**