EECS 376: Foundations of Computer Science
University of Michigan, Winter 2024

Solutions to 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.)

Solutions to Homework 2

```
function TopSort(G = (V, E))                    ▷ G is a directed acyclic graph
    initialize i ← 0, array L[1, ..., |V|]
    S ← {u ∈ V : u has no incoming edge}
    while S is not empty do
        S ← S \ {u} for some arbitrary u ∈ S
        i ← i + 1, L[i] ← u
        for each outgoing edge e = (u, v) ∈ E from u do
            E ← E \ {e}
            if v has no incoming edge then
                S ← S ∪ {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: While the cardinality of S decreases at the start of each iteration when an arbitrary vertex is removed from it, it can also increase at the end of the iteration, if some neighbors of that vertex are added. Because $|S|$ is not strictly decreasing with each (full) iteration, this does not meet the requirements of a potential function.

(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: A new vertex v is added to the set S if v has an edge from the removed vertex u, and has no other incoming edge. If u has no such neighbors v, then $|N_i|$ does not decrease in that iteration of the loop. Thus, $|N_i|$ is not a valid choice.

(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: We claim that $|V| - i$ is a valid potential function (for any valid input), where $|V|$ is the number of vertices in the input graph. (Note that V itself is unchanged through the entire run of the algorithm, so we do not need to index it by i.)

In each iteration of the loop, i increments by 1, which means that $|V| - i$ decreases by 1 with each iteration, as required. We now show that $|V| - i \geq 0$ throughout the algorithm, i.e., that i never exceeds $|V|$. Observe that no vertex can be put into S more than once: adding it a first time means it has no incoming edges, so it cannot be reached from another vertex later on, and in the loop only reachable vertices are added to S. (Note that the algorithm does not introduce any edges to the graph.)

Thus, we put a total of at most $|V|$ vertices into S over the entire algorithm. Since each iteration removes a vertex from S, the algorithm runs for at most $|V|$ iterations, so $i \leq |V|$ as needed. Therefore, our potential function is a valid choice because it is initially finite, strictly decreasing with each iteration, and bounded from below by zero.

Another valid potential function would be $|S_i| + |N_i|$, combining the potentials from parts a and b. At each iteration, an element is removed from S, which decrements the value of the potential. Then, for every neighbor of that vertex that is added to S, $|N_i|$ decreases by 1, so the potential remains unchanged. The potential is also bounded by 0, as both quantities are sets that cannot have less than 0 elements. Furthermore, $|S_i| + |N_i| = 0$ implies that $|S_i| = 0$, which means the algorithm will terminate.

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 impossible 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 $g = \gcd(x, y) = ax + by$

```
1: function ExtendedEuclid(x, y)
2:     if y = 0 then
3:         return (x, 1, 0)                    ▷ Base case: 1x + 0y = x = gcd(x, 0)
4:     else
5:         Divide x by y, writing x = qy + r for integer quotient q and remainder 0 ≤ r < y
6:         (g, a', b') ← ExtendedEuclid(y, r)
7:         [FOR YOU TO DETERMINE: compute appropriate a, b]
8:         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: (1) Recall from the analysis of the "plain" Euclid algorithm (from class) that $\gcd(x, y) = \gcd(y, r) = g$, where $r = x \bmod y$ is the remainder of x when divided by y.
(2) We can derive good choices for a, b as follows. By induction (i.e., correctness of the recursive call), we know that $a'y + b'r = g$. We can rearrange the equation from line 5 to get $r = x - qy$. We want a and b to satisfy $ax + by = g$, i.e., we want to make

$$ax + by = a'y + b'r = a'y + b'(x - qy) \ .$$

To do this, we match up the coefficients of x and y (respectively) from the leftmost and rightmost expressions, which suggests that we should set $a = b'$ and $b = a' - b'q$. We can then verify that these values are correct:

$$\begin{aligned} ax + by &= b'x + (a' - b'q)y \\ &= a'y + b'(x - qy) \\ &= a'y + b'r = g \ , \end{aligned}$$

as required.

(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	s_{i+1}/s_i	$\leq 2/3$?
0	295	204	1	91	29	-65	-65	94	499	295/499	Y
1	204	91	2	22	-7	29	29	-65	295	113/295	Y
2	91	22	4	3	1	-7	-7	29	113	25/113	Y
3	22	3	7	1	0	1	1	-7	25	4/25	Y
4	3	1	3	0	1	0	0	1	4	1/4	Y
5	1	0	–	–	–	–	1	0	1	–	–

Running the algorithm, we find that ExtendedEuclid$(295, 204)$ gives us $(g, a, b) = (1, -65, 94)$, and we can verify that indeed $(-65) \cdot 295 + 94 \cdot 204 = 1$.

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)

```
1: function FUNC(A[1, ..., n])
2:     if n ≤ 3 then
3:         return 0
4:     FUNC(A[1, ..., ⌊n/2⌋])
5:     i ← 1
6:     while i < n
7:         i ← i + 1
8:         FUNC(A[1, ..., n − 3]))
9:     j ← 2
10:    z ← i
11:    while j < i
12:        z ← z − 1
13:        j ← j + 1
14:    return i
```

Solution: The runtime recurrence is $T(n) = T(n/2) + (n-1)T(n-3) + O(n)$. Line 4 makes a recursive call with input size $n/2$. The loop on line 6 runs $n-1$ times and makes a recursive call of size $n-3$ at each iteration. Then, the loop on line 11 runs $n-2$ times, which is $O(n)$.

The Master Theorem does not apply here for any one of multiple reasons: the $T(n-3)$ term, which is not of the form $T(n/b)$ for a constant b; its coefficient $(n-1)$, which is not constant; and the mismatched sizes $n/2, n-3$ for the recursive calls.

(8 pts) (b)

```
1: function FUNC(A[1, ..., n])
2:     if n ≤ 1 then
3:         return 0
4:     x ← FUNC(A[1, ..., ⌊n/2⌋])
5:     y ← FUNC(A[⌊n/2⌋ + 1, ..., n])
6:     z ← FUNC(A[1, ..., ⌊n/2⌋])
7:     if x = 0 or y = 0 or z = 0 then
8:         result ← 0
9:         for i = 1 to n do
10:            for j = 1 to n do
11:                result ← result + 1
12:        return result
13:    h ← MERGESORT(A[1, ..., ⌊n/2⌋])
14:    g ← MERGESORT(A[⌊n/2⌋ + 1, ..., n])
15:    return h[0] − g[0]
```

Solution: The runtime recurrence is $T(n) = 3T(n/2) + O(n^2)$. This is because there are 3 recursive calls on inputs of size $n/2$ made on lines 4-6, followed by an $O(n^2)$-time loop from lines 9-12. At first glance it might look like this loop can be entered only when $\lfloor n/2 \rfloor = 1$, however that is not the case. For example, $x = 0$ if the first and second halves of the array $A[1, \ldots, \lfloor n/2 \rfloor]$, when sorted, have the same first element (because the return value is $h[0] - g[0]$). In this case, we enter the $O(n^2)$-time loop. Lastly, there are two calls to MERGESORT that each take $O(n \log n)$ time, which can be absorbed into the $O(n^2)$ term. (As an exercise, formally prove that this is correct reasoning, according to the definition of big-O notation. That is, show that if $f(n) = O(n^2)$ and $g(n) = O(n \log n)$, then it must be the case that $(f+g)(n) = O(n^2)$.) Thus, the Master Theorem applies with $k = 3$, $b = 2$, and $d = 2$, hence $k < b^d$, which yields a solution of $T(n) = O(n^2)$.

(8 pts) (c)

```
1: function STOOGESORT(A[1, ..., n])
2:     if n = 1 then
3:         return A
4:     if n = 2 and A[1] > A[n] then
5:         swap A[1] and A[n]
6:     if n > 2 then
7:         t ← ⌈2n/3⌉
8:         STOOGESORT(A[1, ..., t])
9:         STOOGESORT(A[n − t + 1, ..., n])
10:        STOOGESORT(A[1, ..., t])
11:    return A
```

Solution: The runtime recurrence is $T(n) = 3T(2n/3) + O(1)$. Lines 8-10 make three recursive calls on arrays of size $2n/3$. The remaining lines take constant time, i.e., $O(1)$. Thus, the Master Theorem applies with $k = 3$, $b = 3/2$ (careful!: not $2/3$), and $d = 0$, hence $k > b^d$, which yields a solution of $T(n) = O(n^{\log_{3/2} 3})$, where $\log_{3/2} 3 \approx 2.71$. (Notice that this is an even worse bound than the one for insertion sort!)

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

Solution: We prove this by strong induction on n. The base case of $n \leq 2$ is trivial. To prove the inductive step, we assume that three recursive calls correctly sort the input subarrays, and show that the algorithm is correct on any input of size n.

Assume for simplicity of notation that 3 divides n. The first key claim is the following: if a number x has rank at least $2n/3$ (i.e., it is in the "top third" of the values in the array), then after the first recursive call the position of x in the array is at least $n/3$. For if not, there are more than $n/3$ numbers larger than x in the sorted subarray after the first recursive call. But this means that the rank of x is less than $2n/3$.

Given this claim, correctness follows fairly straightforwardly. Let S_{high} be the set of elements whose ranks are at least $2n/3$, and let S_{low} be its complement. By the key claim, all numbers in S_{high} will be considered by the second recursive call and so, after the second call, they will be correctly sorted and placed at their correct positions. At this point, S_{low} might not be sorted yet, but the position of any number in S_{low} must be at most $2n/3$ because S_{high} are sorted and placed correctly. Therefore, the third recursive call will exactly sort S_{low}, which sorts the whole array.

For better understanding, you should convince yourself that if the fraction $2/3$ is changed to anything smaller, then the algorithm is incorrect on some inputs.

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 $A[1, \ldots, n]$ 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: The brute-force approach is to check all pairs of students and count how many pairs result in a complaint being filed. For n students, this would check all $\binom{n}{2} = n(n-1)/2$ pairs, which gives an asymptotic runtime of $\Theta(n^2)$.

(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: The main idea is to enhance the MERGE subroutine of the MERGESORT algorithm, to both merge the two halves of the array, *and* count the number of complaints between the two halves. We call this enhanced version MERGEANDCOUNT.

In addition to all the steps done by MERGE, the algorithm does the following. It maintains a counter i_{LR} for the total number of complaints. When considering a student s from the front half by comparing to students in the back half, it keeps track of how many of those students will complain about s. Notice that this is exactly the number of students in the back half who are shorter than s, which is given by the index of the pointer into the back half. When s is finally added to the merged list (either because a taller student in the back half is reached, or the end of the back half is reached), the number of complaints registered against it is added to the counter. Moreover, when considering the next student s' from the front half, the count of complaints against s' starts where the count against s left off, because every student in the back half who complains about s also complaints about s' (since s' is taller than s, and is in front of all the back-half students).

Although not required, we now provide the pseudocode for the algorithm we described.

Algorithm:

```
1: MergeAndCount(L[1...ℓ], R[1...r]) :
2: initialize C[1...ℓ + r] and i_LR ← 0
3: initialize x_L = x_R = 1
4: while x_L ≤ ℓ and x_R ≤ r do
5:     if L[x_L] < R[x_R] then
6:         C[x_L + x_R − 1] ← L[x_L]
7:         x_L ← x_L + 1
8:         i_LR ← i_LR + x_R − 1            ▷ This tracks the number of complaints
9:     else
10:        C[x_L + x_R − 1] ← R[x_R]
11:        x_R ← x_R + 1
12: if x_L ≤ ℓ then
13:    C[(x_L + r)...(ℓ + r)] ← L[x_L...ℓ]
14:    i_LR ← i_LR + (ℓ − x_L + 1)r          ▷ This tracks the number of complaints
15: else
16:    C[(x_R + r)...(ℓ + r)] ← R[x_R...r]
17: return (C, i_LR)
```

Correctness: Define the following function for any person at x_L index in the first array.

$NSS(x_L)$ is the Number of Students Shorter in the second array than the person at index x_L from the first array.

For any x_L we increment x_R in line 11 until we find the smallest x_R that satisfies $R[x_R] > L[x_L]$ greater than the height of x_L. $NSS(x_L)$ would be exactly $x_R - 1$ as the second array is sorted. Thus we increment the number of complaints on x_L in line 8. Anyone from $NSS(x_L) + 1$ to r both inclusive doesn't have complaints on x_L as they all are taller. Then we move on to next person in the first array.

The next important property is $NSS(x_L + 1) \geq NSS(x_L)$. Notice that the person at $x_L + 1$ is taller than the person at x_L as the arrays are sorted. So all the students shorter than $L[x_L]$ would also be shorter than $L[x_L + 1]$ thus, we need not restart the counter x_R and we are guaranteed to find $NSS(x_L + 1)$ correctly. Thus x_R never decrements. We do this until either of the index x_L, x_R reaches their respective end.

If x_L reaches the end of the array then we know that all the complaints are accounted for, however if it doesn't i.e. $x_L \leq \ell$ then the remaining persons $\ell - x_L + 1$ (including x_L) in the first array are taller than everyone in the second array i.e $NSS(i) = r$ for $i \in [x_L, \ell]$. Thus we have $(\ell - x_L + 1)r$ more complaints that we add in line 14. This completes the complaints on remaining people from x_L in the first array.

Since the number of complaints on every person in the first array is accounted by finding $NSS(.)$ and there are no complaints on anyone in the second array as it is sorted. Thus we correctly find all the complaints.

Running Time: We iterate over the while loop at most $\ell + r = n$ times as both indices start from 1 and always increment. Since each iteration does $O(1)$ operations

we do atmost $O(n)$ operations in the loop and $O(1)$ more operations outside the loop. Hence it runs in $O(n)$ time.

(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: Algorithm: The key idea is to devise a divide-and-conquer algorithm that simultaneously sorts *and* counts complaints in a given array. Observe that each complaint is between two students where exactly one of the following conditions holds: (1) both students are in the front half, (2) both are in the back half, or (3) the taller student is in the front half and the shorter student is in the back half.

So, our algorithm is an enhancement of MergeSort that works as follows: it recursively sorts-and-counts-complaints (of the first type) for the front half; then does the same for the back half (and complaints of the second type); then merges-and-counts-complaints (of the third type) for the two sorted halves, using the procedure from the previous part. It finally returns the fully sorted array, together with the sum of the three subcounts. The pseudocode is as follows.

```
1: MergeSortAndCount(A[1...n]) :
2: if n ≤ 1 return (A, 0)
3: (L, i_L) ← MergeSortAndCount(A[1...n/2])
4: (R, i_R) ← MergeSortAndCount(A[n/2+1...n])
5: (C, i_LR) ← MergeAndCount(L, R)          ▷ from the previous part
6: return (C, i_L + i_R + i_LR)
```

Correctness: It is clear that this algorithm correctly sorts, because it reorders the students exactly as MergeSort does. It is less obvious to show that it returns a correct count of complaints. The issue we need to address is that rearranging students (when we recursively sort the two halves) can *change which pairs of students result in complaints*. This could potentially affect the counts returned by later subroutine calls, resulting in an inaccurate count for the *original* array (which is what we care about).

To address this issue, we make the crucial observation that while sorting one half eliminates all complaints within that half, it *does not affect* the complaints between students in the other half, nor those between students in different halves. (That is, it does not eliminate any of those complaints, nor does it introduce any new ones.) This is because a complaint is between just two students, and is determined merely by their positions *relative to each other*. Rearranging students in the front half does not change the positions of any students in the back half, nor does it change the *relative* position of a student in the front half versus one in the back half (the front-half student remains

in front of the back-half one). Similar reasoning applies when rearranging students in the back half.

Therefore, the subcounts we obtain from recursing on the second half (after sorting the front half), and from merging the two halves (after sorting them), are the *same* as the subcounts for the original input array. As already argued, every complaint is from exactly one of the three categories, so the sum of the subcounts is the total number of complains in the original array, as needed.

Running Time: As this algorithm is a lightly modified version of MergeSort, it has running time $O(n \log n)$. More specifically, the runtime obeys the recurrence relation $T(n) = 2T(n/2) + O(n)$: Lines 3 and 4 comprise the $2T(n/2)$ portion of the recurrence relation, and MergeAndCount runs in $O(n)$ time, as shown in the previous part. Using the Master Theorem, we get the desired runtime of $O(n \log n)$.

(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: Using d drops, we can claim that we can handle $n = n(d) = 1+2+\cdots+d = d(d+1)/2$, using the following algoritm.

The algorithm works as follows: if we have d drops left, the first drop is from height d. If the block breaks, we know that the LBH is at most d. So, with our one remaining block and $d-1$ drops, we start from a height of 1 inch and move upwards to no higher than $d-1$ inches, until the block breaks; this determines the LBH. If the block doesn't break from height d, we know that the LBH is strictly greater than d. So, we consider heights $d+1$ to $n(d)$ to be a new smaller range, which we can think of as a "shifted" range of 1 to $n(d) - d$. We then start over (recurse) on this shifted range, still with two blocks, but now with $d-1$ drops left.

Therefore, our algorithm handles an $n(d)$ that obeys the recurrence $n(d) = n(d-1)+d$. The base case is $n(1) = 1$, because given only one drop, we can only distinguish between an LBH of 1 or greater than 1. Using repeated substitution (to be formal, induction), the recurrence expands to $n(d) = 1 + 2 + \cdots + d = d(d+1)/2$, as claimed.

(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: The key idea is that *any optimal* algorithm must agree with our method on the first drop. If an optimal algorithm attempts to make the first drop from a height higher than d, then if the block breaks, there is only one left, and the algorithm cannot be guaranteed to find the LBH (by the reasoning given in the problem setup). On the other hand, if the first drop is from a height less than d, then it can't work for a value of n as large our algorithm's, because it can only be guaranteed to work for an optimal range above that too-low height. Thus, an optimal algorithm must agree with our algorithm on the first drop and, repeating this logic, must agree on every successive drop.

We can more rigorously prove this claim as follows. We prove that for any algorithm

that finds (with certainty) the LBH for some n using two blocks and d drops, we must have $n \leq n(d) = d(d+1)/2$.

In the problem setup we argued informally that with *one block* and d drops, the optimal value of n is $n = d$; now we make this rigorous. Define $n_1(d)$ to be the largest value of n for which it is possible find (with certainty) the LBH with one block and d drops. The crucial observation is that the first drop must be from a height of one inch. For if the first drop is from $j > 1$ inches, and the block breaks, we cannot determine the LBH since it could be any positive integer less than or equal to j. If we first drop from one inch and the block does not break, then we must solve the same problem, but "shifted up by one inch" and with $d-1$ drops, so $n_1(d) \leq 1 + n_1(d-1)$. Since $n_1(1) = 1$, by induction all this implies that $n_1(d) \leq d$ for all d.

Similarly, let $n_2(d)$ denote the largest value of n for which it is possible to find (with certainty) the LBH with two blocks and d drops. We observe that the first drop must be from no higher than d inches: if we drop the first block from $j > d$ inches and it breaks, then we are left with the problem of determining the LBH with $n = j - 1$ (because the outcome "$> n = j - 1$" implies that the LBH is exactly j) and only one block and $d-1$ drops. By the previous result, this requires $j - 1 \leq n_1(d-1) = d - 1$, so $j \leq d$. So, the first drop is necessarily from some height $j \leq d$. If the block does not break, we are left with the same problem, but "shifted up by j inches" and with $d-1$ drops. Thus $n_2(d) \leq n_2(d-1) + j \leq n_2(d-1) + d$. Since $n_2(1) = 1$, this recurrence solves to $n_2(d) \leq d(d+1)/2 = n(d)$, as claimed.