Problem Set 1

This first problem set is designed to help you gain a familiarity with big-O notation and the basics of algorithmic analysis. By the time you're done, you'll have a better sense for how to design and analyze algorithms and how to use big-O notation in the process.

Please read over the "Problem Set Advice" handout before starting this problem set. It contains information about our grading policies, procedures, and expectations for the assignments. In particular, **please be sure to write your answers different problems on separate pages** to make it easier for us to grade.

As always, please feel free to drop by office hours or send us emails if you have any questions. We'd be happy to help out.

This problem set has 36 possible points. It is weighted at 10% of your total grade. The earlier questions serve as a warm-up for the later problems, so the difficulty of the problems increases over the course of this problem set.

Good luck, and have fun!

Due Wednesday, July 3 at 2:15 PM

Problem One: Counting Sort (2 Points)

The *counting sort* algorithm is an algorithm for sorting an array of integers, each of which happens to fall in the range [0, *U*) for some number *U*. Here is pseudocode for counting sort:

```
procedure countingSort(array A, int U)
    1. let counts = new array of size U
    2. for i = 0 to U - 1:
    3.    counts[i] = 0
    4. for i = 0 to length(A) - 1:
    5.    counts[A[i]] = counts[A[i]] + 1
    6. let index = 0
    7. for i = 0 to U - 1:
    8.    for j = 0 to counts[i] - 1:
    9.        A[index] = i
   10.        index = index + 1
```

We suggest tracing through this algorithm on some small arrays to get a sense for how it works.

One correct but loose analysis of counting sort's runtime is the following:

> The loop on line 2 runs O(*U*) times and does O(1) work on each iteration, so it does a total of O(*U*) work. The loop on line 4 runs O(*n*) times (where *n* is the length of the input array) and does O(1) work on each iteration, so it does a total of O(*n*) work. Thus the initial setup takes O(*n* + *U*) time.
>
> Loop 7 has a nested loop within it. The nested loop (loop 8) can execute at most O(*n*) times (because `counts[i]` ≤ *n*) and does O(1) work on each iteration, so it does at most O(*n*) work. Therefore, since Loop 7 runs O(*U*) times, the work done by the loop is O(*nU*).
>
> Therefore, the overall work done is O(*nU*).

While this analysis of the runtime is correct, it overestimates the amount of work done by this counting sort. Prove that counting sort actually runs in time Θ(*n* + *U*).

(When doing this analysis, you do not need to use the formal definition of Θ notation. You can use an intuitive analysis along the lines of the above.)

The next question didn't fit nicely on this page, so here's a picture of a happy puppy instead!

Problem Two: O, Ω, and Θ notations (10 Points)

Over the course of the quarter, we will use O, Ω, and Θ notation to quantify properties of algorithms – their runtimes, their memory usages, etc. This question asks you to prove various properties about these notations or to show that certain properties do not necessarily hold.

For each of these problems, if you want to prove a result, you should offer a formal mathematical proof using the rigorous definition O, Ω, or Θ notation. If you want to disprove a result, you can give a counterexample, but should prove why your counterexample is valid.

Assume all functions listed below have ℕ as their domain and codomain.

i. Prove or disprove: For any functions f and g, $f(n) + g(n) = \Theta(\max\{f(n), g(n)\})$.

ii. Prove or disprove: If $f(n) = O(g(n))$, then $2^{f(n)} = O(2^{g(n)})$.

iii. Prove or disprove: If $f_1(n) = \Theta(g_1(n))$ and $f_2(n) = \Theta(g_2(n))$, then $f_2(f_1(n)) = \Theta(g_2(g_1(n)))$.

iv. Prove or disprove: For any functions f and g, $f(n) = O(g(n))$ or $g(n) = O(f(n))$. (This is an inclusive OR.)

v. Prove or disprove: $n! = O(2^n)$.

Problem Three: Signaling for Help (5 Points)

Suppose that you are stranded on a desert island. You have a radio and a battery with you, and the radio is capable of transmitting at different integer power levels (e.g. 1W, 2W, 3W, 4W, …). As soon as you transmit a distress signal with enough power for someone to receive it, you will get confirmation that the message was received and help will be sent to you. Unfortunately, you don't know how close the nearest rescue crew is, so you don't know how much power to feed into the radio.

Suppose that you need to feed at least n watts of power into the radio for your distress signal to be heard; that is, if you transmit at any power level greater than or equal to n watts, your signal will be received. However, you do not know what n is. Your goal is to design an algorithm for sending a distress signal that does not use much more power than is necessary.

For example, you could try transmitting the distress signal at powers 1W, 2W, 3W, 4W, … until you reach power nW, at which point you will get confirmation that the message was received. Unfortunately, this approach will use total power $1W + 2W + 3W + \ldots + nW = \Theta(n^2)W$, which can be problematic if n is large. Fortunately, though, it's possible to send a distress signal using only $\Theta(n)$W power.

Design an algorithm that will successfully transmit a distress signal, but which uses only $\Theta(n)$W total power. Remember that n is unknown to you. Then:

- Describe your algorithm.
- Prove that your algorithm is correct.
- Prove that your algorithm uses $\Theta(n)$W power.

Problem Four: Searching a Grid (9 Points)

Suppose that you are given an $m \times n$ grid of integers where each row and each column are in sorted order (we'll call this a *sorted grid*). For example:

10	12	13	21	32	34	43	51	67	69	90	101	133
16	21	23	26	40	54	65	67	68	71	99	110	150
21	23	31	33	54	58	74	77	97	98	110	111	150
32	46	59	65	74	88	99	103	113	125	137	149	159
53	75	96	115	124	131	132	136	140	156	156	157	161
84	86	98	145	146	151	173	187	192	205	206	208	219
135	141	153	165	174	181	194	208	210	223	236	249	258
216	220	222	225	234	301	355	409	418	446	454	460	541

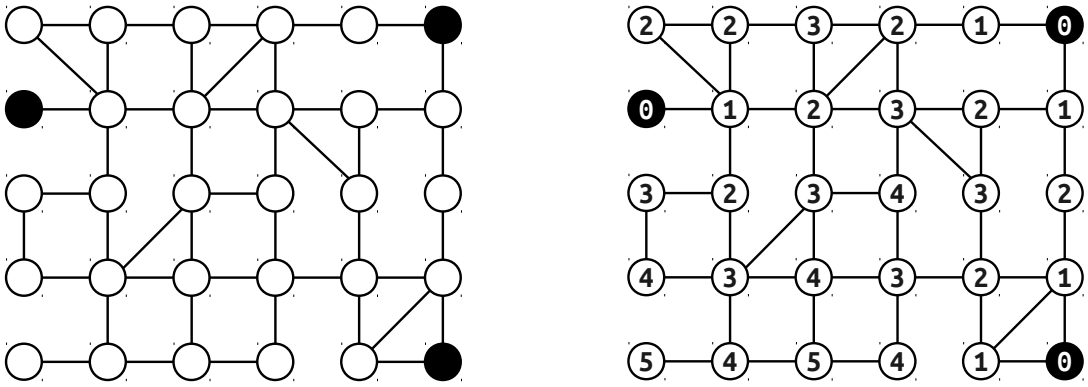
Design an $O(m + n)$-time algorithm that, given as input a sorted $m \times n$ grid and an integer, returns whether or not that integer is contained in the grid. Your algorithm should use only O(1) additional space. Then:

- Describe your algorithm.
- Prove that your algorithm is correct.
- Prove that your algorithm runs in time $O(m + n)$ and uses O(1) additional space.

Problem Five: Emergency Route Planning (9 Points)

Suppose that you have an undirected graph representing a city's transportation network. Each edge represents a street (which for now we'll assume is a two-way street), and each node represents an intersection.

Certain intersections in the city are hospitals, and you are interested in finding, for each intersection, the distance that intersection is from the nearest hospital, as measured by the number of edges in the path from that intersection to the nearest hospital. For example, given the following transportation network, where black nodes represent hospitals, the distances are as follows:



(Continued on the next page)

Let n be the number of nodes in the transportation network and let m be the number of edges. Although in the above example there were exactly three hospitals, any number of nodes in the grid can be hospitals.

i. Design an $O(m + n)$-time algorithm for computing the distance from each intersection to the closest hospital. Note that your algorithm's asymptotic runtime should not depend on the total number of hospitals. Then:

   - Describe your algorithm.
   - Prove that your algorithm is correct.
   - Prove that your algorithm runs in time $O(m + n)$.

   (Hint: Try using one of the algorithms we've covered so far as a subroutine.)

ii. Suppose that the city's transportation graph has some one-way streets, meaning that some of the edges in the transportation network are directed. Briefly describe how you would modify your algorithm from part (i) to account for this while still maintaining the $O(m + n)$ runtime. You don't need to write a formal proof here – just give a one-paragraph description of your modified algorithm and a brief justification of why it works.

Problem Six: Course Feedback (1 Point)

We want this course to be as good as it can be, and we'd appreciate your feedback on how we're doing. For a free points, please answer the following questions. We'll give you full credit no matter what you write, as long as you write something.

i. How hard did you find this problem set? How long did it take you to finish? Does that seem unreasonably difficult or time-consuming for a five-unit course?

ii. Did you attend office hours? If so, did you find them useful?

iii. Did you read through the textbook? If so, did you find it useful?

iv. How is the pace of this course so far? Too slow? Too fast? Just right?

v. Is there anything in particular we could do better? Is there anything in particular that you think we're doing well?