COMPSCI 311: Introduction to Algorithms **Spring 2022**

Homework 1

Released 1/30/2021 Due 2/11/2022 11:59pm in Gradescope

Brendan Cadogan
Collaborators: Nate Courchesne, Benjamin Tufano

Instructions. You may work in groups, but you must write solutions yourself. List collaborators on your submission. You are allowed to have at most 4 collaborators. Also list any sources of help (including online sources) other than the textbook and course staff. If you are asked to design an algorithm, please provide: (a) the pseudocode or precise description in words of the algorithm, (b) an explanation of the intuition for the algorithm, (c) a proof of correctness, (d) the running time of your algorithm and (e) justification for your running time analysis. **Submissions.** Please submit a PDF file. You may submit a scanned handwritten document, but a typed submission is preferred. Please assign pages to questions in Gradescope.

1. **(15 points) Stopping Schedule** A set of n drones perform observations using n landing pads to recharge. Each drone's flight schedule includes exactly one recharge at each of the pads. A pad can only be used by one drone at any given time, and their schedules are set up accordingly. Initially, all drones are in flight, but it is decided to bring them all in for an upgrade. Their programmed schedules cannot be modified, but only cut short: at any recharge pad, you can command the drone to stop. However, a pad can still hold only one drone, so you must decide where to stop each drone.
Knowing the drone schedules, give an algorithm that determines where to stop each drone, minimizing the time at which all are brought in. Prove that this is possible and your algorithm is correct, and determine its complexity (which should be polynomial-time).
Do not assume anything else about the length of flight or recharge times in the schedule.

   A. I'm going to use the Gale Shapley algorithm for solving this problem. In my version, only the drones will propose. The drone's will prefer the pads they land on first, while the pads will prefer the drones that land on them last. A pad accepts a drone's offer either if it has no drone paired to it yet, or if it does have a drone paired to it, if it prefers the new drone to the the old drone.

   B. The intuition behind this is by having the drones prefer pads that they land on first, using the stable matching algorithm, it would minimize the time at which all are brought in. By having the pads prefer drones that land on them last, it makes it so a stable match is only possible when there is not a crash at that pad.

   C. We've already proven Gale Shapley in class, so all I need to prove is that this problem is an instance of Gale Shapley. This is an instance of Gale Shapley because we have two groups each with a preference list and each member of each group can only be matched with one member of another group. Now, an instability in this problem is when a drone d_1 prefers a different pad P_1 than its current pad P_2 because it lands there first, and a pad P_1 prefers that drone d_1 over its current drone d_2 because that drone lands there after its current drone. So if there is an instability, it means that d_1 landed at P_1 and then got up and landed at a later time at P_2. It also means that at P_1, d_2 landed first and then d_1 landed, which would be a crash. Another thing, since drones propose, we minimize time, but if pads proposed, we would have a slower time that would still work.

   D. Gale Shapley is $O(n^2)$.

   E. This is $O(n^2)$ because the while loop iterates over at least every Drone, which is n, and can continue iterating over different pads for each drone which is also n. So $n * n = n^2$

2. **(30 points) Big-O and big-Ω.** For each function $f(n)$ below, find (1) the smallest *integer* constant H such that $f(n) = O(n^H)$, and (2) the largest *positive real* constant L such that $f(n) = \Omega(n^L)$. Otherwise, indicate that H or L do not exist. All logarithms are to base 2. Your answer should consist of: (1) the correct value of H, (2) a proof that $f(n)$ is $O(n^H)$, (3) the correct value of L, (4) a proof that $f(n)$ is $\Omega(n^L)$.

   a) $f(n) = n \log\log n / \log n$.
   b) $f(n) = \sum_{k=1}^{n} \sqrt[3]{k^2(n-k)}$
   c) $f(n) = \sum_{k=1}^{n} k/2^k$.
   d) $f(n) = \sum_{k=1}^{n} k \log k$.
   e) $f(n) = 2^{\sqrt{\log n}}$.

A.
The smallest integer H is 1
Because we have $\frac{nloglogn}{logn} <= \frac{nlogn}{logn} = n$
L doesn't exist
Because we have $\frac{nlogn}{logn} => \frac{nloglogn}{loglogn}$ and $\frac{nloglogn}{logn} => n$

B.
The smallest integer H is 2
The largest positive real constant is L 1

C.
$H = 1$
Because $k/2^k < 2^k/2^k = n^0$ So $f(n) = O(n^1)$
L does not exist

D.
$H = 3$
Because $\sum_{k=1}^{n} k = n^2$ which is n^2
and $logk < n$ which is n
so $f(n) = O(n^3)$
$L = 2$
Because $\sum_{k=1}^{n} k = n^2$ which is n^2
and $logk > n^0$ which is n^0
sof(n) $= \omega(n^2)$

E.
$H = 1$
$2^{\sqrt{logn}} <= 2^{logn} = n$
L does not exist
L does not exist because $2^{\sqrt{logn}}$ is asymptotic so any value of L would eventually overpower $2^{\sqrt{logn}}$.

3. **(15 points) Alternating Sums** Given an array A of n integers, fill an $n \times n$ array B such that for all $i < j$, $B[i,j]$ is the alternating sum $A[i] - A[i+1] + \ldots + (-1)^{j-i}A[j]$. For $i \geq j$, ignore $B[i,j]$.

```
for $i = 1, 2, \ldots, n$
    for $j = i+1, \ldots, n$
        compute $A[i] - A[i+1] + \ldots + (-1)^{j-i}A[j]$.
        store in $B[i,j]$.
```

(a) What is the running time of the given algorithm as a function of n? Specify a function f such that the running time of the algorithm is $\Theta(f(n))$.

The running time is $T(n) = n^3$ because the two for loops are n and then there is one constant time operation in the for loops. The compute $A[i] - A[i+1] + \ldots + (-1)^{j-i}A[j]$ is n so $n * n * n * 1 = n^3$
A function with $\Theta(f(n))$ is $f(n) = n^3$ because big o is $0(n^3)$ and big omega is $\omega(n^3)$ and $c_1 n^3 <= 1n^3 <= c_2 n^3$

(b) Design and analyze a faster algorithm for this problem. You should give an algorithm with running time $O(g(n))$, where $\lim_{n\to\infty} g(n)/f(n) = 0$.

A.

```
for $i = 1, 2, \ldots, n$
    $t[i] = A[i]$
    for $j = i+1, \ldots, n$
        $t[i] = t[i] - (-1)^{i+j+1} * A[j]$
        $B[i,j] = t[i]$
```

B. The previous algorithm was $O(n^3)$ because it had three for loops each with linear time. Since we still need at least two for loops because we are populating a 2D array, the quickest we can get is $O(n^2)$. To do this, all we need to do is convert the third for loop, which is the compute line in the given pseudo code, to constant time.

C. We can prove this by proving our algorithm is equivalent to the alternating sum equation. By doing some algebra, we determine that they are equivalent and that our algorithm works.

D. The running time is $O(n^2)$

E. The running time is $O(n^2)$ because there are two linear time for loops and then three constant time operations. $3 * n * n = 3n^2$

4. **(20 points) Butterfly Species.** A group of butterfly lovers examine n butterflies to determine which are part of the same species. Their examination produces a set of m assertions of the form $S(x, y)$, meaning "butterflies x and y are of the same species". Assume these assertions are all accurate, and that "same species" is an equivalence relation. Also assume that if the assertions together with the rules for equivalence relations *do not* imply that two butterflies are of the same species, then they belong to different species.

   a) (8p) Give an algorithm that has as input the list of butterflies and assertions, and two butterflies x and y, and returns a boolean stating whether x and y are of the same species. Argue that your algorithm is correct and analyze its running time.

   A. A butterfly x is in the same species as another butterfly y if there is a path from x to y through the assertions. If there are no set of assertions that connect x and y, then those butterflies are not of the same species. To solve this problem, we can use a simple while loop with BFS in it that will search all of the components of the graph until we find x and y in the same component which would mean they are the same species, or until we've discovered every node without finding them in the same component.

```
While there is some unexplored node $s$
    BFS(s)
        if(x and y are found in the component)
            return true
    Extract connected component containing s
return false if exited the while loop
```

   B. The intuition for this is that we can define the assertions as edges and the butterflies as nodes. Knowing this, all we need to do is to run BFS on every component until we find x and y are in the same component, or that they are not.

   C. Using proof by contradiction, we can prove that all butterflies are reached through our algorithm. Lets say a butterfly has not been discovered, if that is the case, then not all butterflies have been visited because the while loop can't be exited until all have been visited. Using proof by induction, we can prove that any butterfly in component $c_1$is connected to any other butterfly in component c_1. The base case is when nodes = 1 which is true. The hypothesis is that $c_n + 1 = c_{n+1}$ which is true so all butterflies in the same component are connected. Therefor, the algorithm will not terminate until all nodes are visited or until x and y are found in the same component

   D. The run time is $O(m + n)$

   E. The run time is $O(m + n)$ because the algorithm iterates over each node, which is n, and each edge, which is m.

   b) (5p) Give an algorithm that has as input the list of butterflies and assertions and returns the number of different butterfly species. Argue that your algorithm is correct and analyze its running time.

   A. To solve this problem, all we need to do is count how many components there are in the graph because each component is one species. This is similar to the previous question, except that we now keep a counter of how many components we have iterated through using BFS.

```
While there is some unexplored node $s$
    run BFS(s)
    Extract connected component containing s
    speciesCount++
.return speciesCount
```

B. The intuition for this problem is that all we need to do is count the amount of components, which can be done with a BFS over every component, removing the already seen nodes, and counting up every time the while loop loops.

C. Given the proof from part a, all we need to prove is that every component is reached. This can be done by contradiction, and the idea that as long as there is an unseen component, the while loop won't exit, so the algorithm must still be running if a species hasn't been found yet.

D. The run time is $O(m + n)$

E. The run time is $O(m + n)$ because it is just a modified BFS which is $O(m + n)$.

c) (7p) Give an algorithm that takes as input the list of butterflies and the list of assertions and returns a minimum-size subset of assertions that suffice to answer for any two butterflies whether they are of the same species, i.e., question (a). Argue that your algorithm is correct and analyze its running time.

A. We can do this by taking the algoritm from part A and modify it to detect back edges which is something we learned that BFS can do in class. When we detect a back edge, all we need to do is delete that edge. At the end of the while loop that we run BFS in, we return the new component if true, or null if false.

```
While there is some unexplored node $s$
        Check for x and y in the component
        for all nodes v in L[i] do
            for all neighbors w of v do
                if w is not marked "discovered" then
                mark w as "discovered"
                put w in L[i]
                parent[w] = v store parent of each node
            else if w $!= parent[v]$ then
                delete that edge
        if x and y were found, store the component
        Extract connected component containing s
return component
```

B. The intuition for this problem was that it was part A, with the added restriction that we needed to get rid of back edges.

C. With the proof from part a, all we need to prove is that all back edges are eliminated and that the resulting path is the shortest. To do this, we can use induction. The base case is with one node which is obviously true. Then allwe need to do is prove $nodes_n + 1 = nodes_{n+1}$. Doing out the math, this is true and therefor all back edges are eliminated and the resulting component is the shortest path.

D. The run time is $O(m + n)$

E. The run time is still $O(m + n)$ because we still iterate over at most every node and edge, just like the previous two algorithms

5. **(20 points) Path Lengths modulo k.** Let G be an undirected graph, s and t arbitrary nodes in G, and $k > 1$ an integer. Design an algorithm that finds the shortest path from s to t whose length is divisible by k, or determines that no such path exists. Argue that your algorithm is correct and analyze its running time.

   A. To solve this problem, we'll be using DFS. Starting from node s, we try and find node t. If we don't find t within the graph, no path exist. A path can also not exist depending on the oddness and evenness of k and the path from s to t. This happens when the only possible paths from s to k are odd while k is even. When we run our DFS we want to find the shortness path to s, and then jump back and forth between t and an adjacent node until we get a number divisible by k. This will be of course only possible if k and the path to t fit our requirement above.

   B. The intuition for this problem was that we had to use a DFS because depth was important

   C. To prove this, we need to prove that when the algorithm is supposed to return a path, it returns the minimum, and that when a path does not exist, it doesn't return a path. This can be done by induction.

   D. The run time is $O(m + n + k)$

   E. The run time is $O(m + n + k)$ because it is a DFS search which is $m + n$ with at least k jumps.

   Extra credit (5p): Solve the problem with the additional constraint that a path may not immediately go back on the last edge traversed, i.e., may not contain a sequence of the form $v - w - v$.