Please enter your name and uID below.

Name: Sabath Rodriguez

uID: u1254500

Submission notes

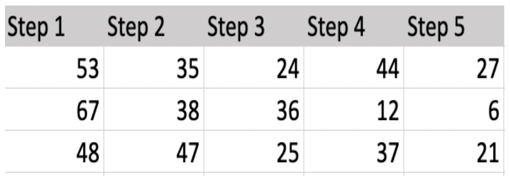
- **(PS 1 specific) This is a warm-up assignment. Full points will be given for answers that have some correct elements and that clearly been given a legitimate effort.**

- *Solutions must be typeset* using one of the template files. For each problem, your answer must fit in the space provided (e.g. not spill onto the next page) *without* space-saving tricks like font/margin/line spacing changes.
- Upload a PDF version of your completed problem set to Gradescope.
- Teaching staff reserve the right to request original source/tex files during the semester, so please retain the original files.
- Please remember that for problem sets, collaboration with other students must be limited to a high-level discussion of solution strategies. If you do collaborate with other students in this way, you must identify the students and describe the nature of the collaboration. You are not allowed to create a group solution, and all work that you hand in must be written in your own words. Do not base your solution on any other written solution, regardless of the source.

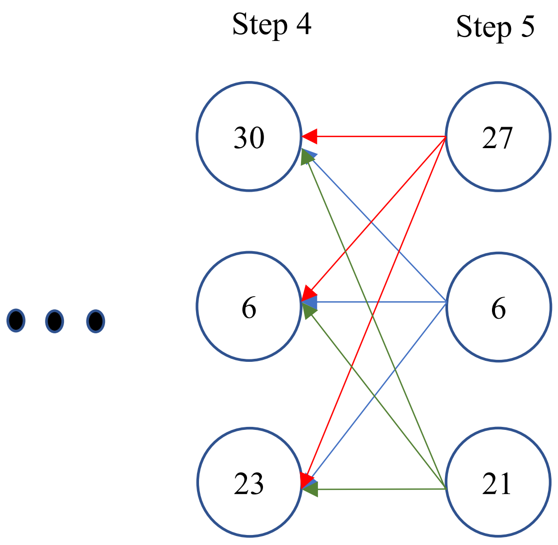
1. (Analysis of your coding solution)

My algorithm simply starts with 2 2D arrays, one for containing the cost at each step at each lane, which I'll explain more later, and the other to keep track of the lanes we used to get our optimal solution. We start by looking for dependencies, meaning which parts of our partial solutions(s) depend on another piece to finish their part, and the answer here is that we need to know what the least "expensive" option was to get to our current late at our current step, in other words, what's the cheapest way to get to our current position? Once we know what the least expensive way was, all we know need to know now is look at our options going forwards and pick the best option from there. Okay, so if we know we need to look at the last step to find out what the least expensive option was to get to our current position, then that should intuitively tell us what our dependencies are, in this case it is our last step at all lanes, because we can ask "what is the least expensive way to get here?" and the answer is simple, it's that step on that lane in and of itself. So we start from the last step on all lanes, then for the next move, we move onto the next step (to be clear if our last step was n, we move onto step n-1) and check what is the least expensive option to move from our current lane at our current step to either the same lane and the next step or another lane and the next step, and we do this for all lanes, we do this for all steps until we reach the 1st step, which we will then know on the 1st step what the best possible solution is, because we are aggregating the costs from before at every step onto the current step we are on. As we're iterating through each lane at each step and finding the best option for that step and lane, we should store that optimal option in a separate array, and once we're done with finding the optimal cost, the last step is to trace back how we got there, and for that we use our array where we were storing the optimal options. I will leave an example below to better illustrate my method for solving this problem using dynamic programming.

Step 1



Step 1	Step 2	Step 3	Step 4	Step 5
53	35	24	44	27
67	38	36	12	6
48	47	25	37	21



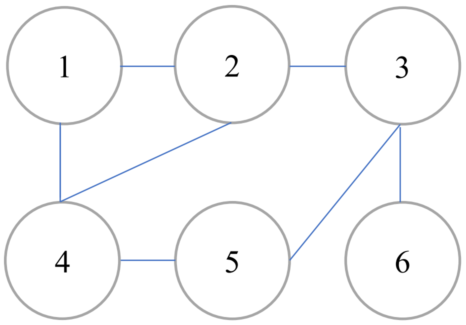
Now for the last step, the upper bound of my algorithm. We're going move from right to left, meaning for s steps, and from top to bottom, meaning for n steps, and we're going to check all lanes against all lanes, meaning n^2. So we're going to do $s * n^2$ steps, so my algorithm is upper bounded by $O(s * n^2)$.

2. (Greedy Islands)

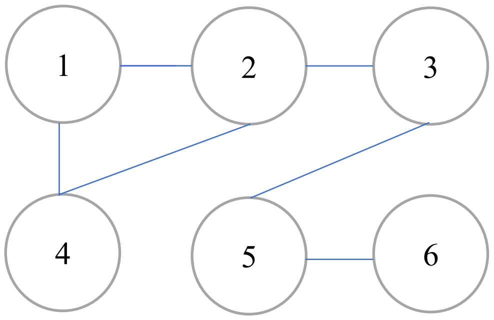
Approach #1 (Islands 1)

Check Island n and n+1, pick whichever has the greater number of connections and recurse.

Islands 1



Islands 2



If we follow the metrics of our simple greedy algorithm, we quickly come to find out that it will not give us the correct answer. For example, if we follow our approach of simply selecting the island with the greatest number of connection starting at island 1 and comparing it to island 2, or more generally, n.connections < n+1.connections, and we choose 2 in this case, which covers islands 1,3, and 4, we now pick from the next islands that need coverage, which in our case is 5 and 6, so we choose 5, since 5 has a larger number of connections, and finally we choose 6, which has a connection of 1. So the total islands we chose are 2, 5, and 6. Okay, now let's find a better approach, let's not choose the most immediate island with the largest number of connections, instead let's try all possible combinations and choose the best, meaning the least number of shops to place on all islands, in this case it would be 4 and 6, which is 1 less shop that we need opened and is not the island with the immediate highest number of connections.

Approach #2 (Islands 2)

Visit all islands iteratively and then afterwards choose the one with the highest number of connections, or the first one with that number of connections that we visited. If we follow the metrics of this simple greedy algorithms, then we start at island 1 and use that coverage, then the next island that needs coverage and use that island plus whatever coverage it gives us, and so on until all islands are covered. In our case with the given example islands, we start at island 1 which covers islands 2 and 4, next we choose 3 which covers island 5, and finally we choose 6, we now have all of our islands covered. However, if we step back and try to find a better solution, we will quickly come to realize that choosing island 2 and 6, will cover all islands as well, so that's a difference of placing 3 shops vs. 2 shops.

3. (Positive Intervals)

My algorithm:

```
1.  A = [+3, -5, +7, -4, +1, -8, +3, -7, +5, -9, +5, -2, +4]
2.  positiveIntervals = [[]]
3.  for i = 1 -> |A|
4.      sum = A[i]
5.      interval = []
6.      for j = i + 1 -> |A|-1
7.          sum += A[j]
8.          if (sum + A[j+1]) >= 0:
9.              if j equals (i + 1) // is the first number
10.                 interval.add(A[i])
11.             interval.add(A[j])
12.             interval.add(A[j+1])
13.             sum += (A[j+1])
14.             j += 1
15.         else
16.             i = j //start from j, not all the way back from the original i.+ 1
17.             break out of first looop
18.     positiveIntervals.add(interval)
19.     interval.empty()
```

My algorithm is essentially looking if a number at i plus a number at i + 1in front of it can be offset by the next number if it's negative at i + 2, if it can't be offset then we don't include it in our positive intervals. I essentially have 2 loops, the first to tell us where to start looking for our positive interval, and our second to tell us where to stop, the second loop is also where we check if the two numbers can be offset by the 3rd. I'll explain a little more clearly, on line 6 we will know where our positive interval starts from, then on line 4-7, we add elements from i and i + 1 (j), we then check if they are positive or if they can offset by the next element at i+2 (j+1) on line 8, if they can, we add i+1(j) and i+2(j+1) to our interval array, we also want to move our j element up by one to take into account that we checked i+2(j+1), once we cannot offset the next element at i+2(j+1), we stop our interval, set our next starting point for i where we left off in j, on line 16, then add our interval array to positive intervals, and we start over. The reasoning for my algorithm is such that if we have 2 consecutive numbers that add up to be negative, can they be made into a positive by adding the number in front of both of them, and this will create a positive interval if so, if not, we move on to the next element and start the process over again, guaranteeing us that if there's a positive interval, we will find it going from left to right. This will give us an answer that is at least as good as our globally optimal algorithm. The running time of my algorithm is O(n), and to show this, I'll presume an array that has 0 positive intervals, which could simply mean an array of all negative numbers. If we have an array with no positive numbers we can see that our first for loop on line 3, will iterate n times, however, with a small caveate, if we look at line 16, we'll notice that we make i = j if we don't find any positive intervals, so since j = i + 1, our next i will instead be i = i + 2 rather than just i = i + 1, regardless, our loop will then iterate n – 2 times, rather than just n times, so n – 2 is still dominated by n, therefore my algorithm runs in O(n) time.