COSC 320: Assignment 2

This assignment is due **Friday, February 7 at 7 PM**. Late submissions will not be accepted. All the submission and formatting rules for Assignment 1 apply to this assignment as well.

1 List of names of group members (as listed on Canvas)

Provide the list here. This is worth 1 mark. Include student numbers as a secondary failsafe if you wish.

2 Statement on collaboration and use of resources

To develop good practices in doing homeworks, citing resources and acknowledging input from others, please complete the following. This question is worth 2 marks.

1. All group members have read and followed the guidelines for groupwork on assignments given in the Syllabus).

   ○ Yes ○ No

2. We used the following resources (list books, online sources, etc. that you consulted):

3. One or more of us consulted with course staff during office hours.

   ○ Yes ○ No

4. One or more of us collaborated with other COSC 320 students; none of us took written notes during our consultations and we took at least a half-hour break afterwards.

   ○ Yes ○ No

   If yes, please list their name(s) here:

5. One or more of us collaborated with or consulted others outside of COSC 320; none of us took written notes during our consultations and we took at least a half-hour break afterwards.

   ○ Yes ○ No

   If yes, please list their name(s) here:

3 Logarithmic functions grow more slowly than "polynomial" functions

The textbook (2.8, page 41) provides the following useful fact, stating roughly that logarithmic functions are big-O-upper-bounded by simple "polynomial" functions, specifically functions that are n to some constant power:

> **Fact:** *For every $b > 1$ and every $x > 0$, we have $\log_b n = O(n^x)$.*

1. (4 points) Use the fact above to prove the stronger assertion that logarithmic functions of n grow strictly more slowly, in the little-o sense, than functions that are n to some constant power:

   For every $b > 1$ and every $x > 0$, we have $\log_b n = o(n^x)$.

   If you want, you can start your proof as follows:

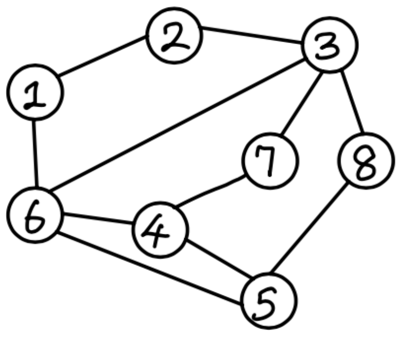
   Fix $b > 0$ and $x > 0$. Using the fact, and since $x/2 > 0$, it must be the case that $\log_b n = O(n^{x/2})$.

2. (4 points) Now use the fact of part 1 to show that $\sqrt{n} = o(n/\log^3 n)$. Here the log is to the base 10, and $\log^3 n = (\log n)^3$.

4 Counting Shortest Paths

Let $G = (V, E)$ be an undirected, unweighted, connected graph with node set $\{1, 2, \ldots, n\}$, where $n \geq 1$. For any node v, let $c(1, v)$ be the total number of shortest paths (i.e., paths with the minimum number of edges) from 1 to v.

Example: The following graph has one shortest path from 1 to 6. Also there are three shortest paths from 1 to 8. Two of these, namely path 1,2,3,8, and path 1,6,3,8, go through node 3, while one, namely path 1,6,5,8 goes through node 5.



1. (2 points) Draw a breadth first search tree rooted at 1 for the graph above. Include all dashed edges as well as tree edges. (A scanned hand-drawn figure is fine as long as it is clear.)

2. (2 points) How many shortest paths are there from node 1 to node 7? That is, what is the value of $c(1, 7)$? Give a list of the paths.

3. (4 points) Here is an inductive definition for $c(1, v)$: The base case is when $v = 1$, in which case we define $c(1, 1)$ to be 1, since there is exactly one shortest path from 1 to itself (with no edges). When $v > 1$, let $d[v]$ be the depth of any node v in the bfs tree of G rooted at 1. Then

$$c(1, v) = \sum_{\substack{u \mid (u,v) \in E, \text{ and} \\ d[u] = d[v] - 1}} c(1, u).$$

Intuitively, on the right hand side we are summing up the number of shortest paths from node 1 to all nodes u at level $d[v] - 1$, such that there is an edge of G (either a tree edge or a dashed edge) from u to v. In our example above, $c(1, 8) = c(1, 3) + c(1, 5)$.

Provide code in the spaces indicated below, that leverages this inductive definition to obtain an algorithm that computes $c(1, v)$ for all nodes v. This code first initializes $c(1, v)$ for all v, and then calls a modified version of breadth first search that you will flesh out.

```
procedure COUNT-SHORTEST-PATHS($G$.1)
    ▷ $G$ is an undirected, connected graph with nodes $\{1, 2, \ldots, n\}$, where $n \geq 1$
    ▷ compute $c(1, v)$, the number of shortest paths, from 1 to $v$, for all nodes $v$
    ▷ add code here to initialize $c(1, v)$ for all $v \in V$:



    call MODIFIED-BFS($G$)
end procedure
```

```
procedure MODIFIED-BFS($G$)
    ▷ Assume that this procedure can access and update the variables $c(1, v)$
    add node 1 as the root of the bfs tree
    $d[1] \leftarrow 0$                                                ▷ node 1 is at level 0
    for all nodes $v > 1$ do
        $d[v] \leftarrow \infty$                                       ▷ $v$ is not yet in the tree
    end for
    $d \leftarrow 1$
    while not all nodes are added to the tree do
        for each node $u$ in the tree with $d[u] = d - 1$ do
            for each $v$ adjacent to $u$ do
                if $d[v] == \infty$ then                    ▷ $v$ has not yet been added to the tree
                    $d[v] \leftarrow d$            ▷ put node $v$ at level $d$ of the tree (as a child of $u$)
                end if
                ▷ add your code here to update $c(1, v)$:



            end for
        end for
        $d \leftarrow d + 1$
    end while
end procedure
```

5 Provision planning

You run a business to provide provisions to individuals with plans for long-distance hikes. An individual requesting your help tells you:

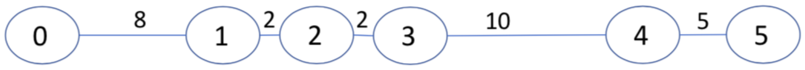
- d: The distance (in km) that the individual can hike per day, where d is a positive integer;
- p: How many days of provisions (food, water, etc.) they can carry, where p is a positive integer.

In addition you have access to:

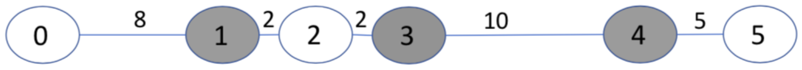
- $R[1..k]$: inter-town distances along the planned route, with $k \geq 1$. That is, there are $k+1$ towns along the route with town 0 being the start and town k being the destination; and $R[i]$ is the distance (in km) from town $i-1$ to town i for $1 \leq i \leq k$.

You need to store provisions in towns along the way, that the hiker will pick up en route. For this problem, you need only concern yourself with instances $(d, p, R[1..k])$ that have valid solutions. A *valid solution* is a list of towns where provisions can be placed, so as to ensure that the hiker will not run out of provisions. In a valid solution, the distance traveled between the starting town and the first town in the solution, or between a consecutive pair of towns in the solution, or between the last town in the solution and the destination, is $\leq dp$ (where d and p are defined above). You never need to provide provisions at town 0 or town k (the hiker has their own provisions at the start of the trip and does not need them once the destination is reached). You want to find an *optimal solution*, that is, a valid solution of minimum length.

Example: A hiker plans to hike for at most 7km per day, and can carry provisions that last for two days, so $d = 7$ and $p = 2$. Also, $k = 5$ and $R[1..5] = [8, 2, 2, 10, 5]$, so distances between towns are:



One valid solution is the list of towns 1, 3, 4:



1. (3 points) In the example above, the solution is not optimal. Give *three different optimal solutions*, which have only two towns in each.

2. (2 points) Consider the following greedy algorithm.

```
 1: procedure GREEDY-PROVISIONS($d$, $p$, $R[1..k]$)
 2:     $i \leftarrow 0$                                      ▷ 0 is the starting town
 3:     $L \leftarrow$ empty list
 4:     while $i < k$ do
 5:             ▷ find the town $j \leq k$ that's furthest away from town $i$, among those of distance $\leq dp$
 6:         $j \leftarrow i + 1$
 7:         $d' \leftarrow R[j]$
 8:         while ($j < k$) and ($d' + R[j + 1] \leq dp$) do
 9:             $j \leftarrow j + 1$
10:             $d' \leftarrow d' + R[j]$
11:         end while
12:         if $j < k$ then
13:             $L \leftarrow L, j$                            ▷ append $j$ to the solution $L$
14:         end if
15:         $i \leftarrow j$                                   ▷ update $i$
16:     end while
17:     output the list $L$
18: end procedure
```

Explain why the output L is a valid solution, i.e., one in which the hiker will not run out of provisions, given that instance $(d, p, R[1..k])$ is guaranteed to have a valid solution.

3. (3 points) Give a big-O bound on the running time of the greedy algorithm as a function of k, the number of towns, and justify your answer. (The lines of pseudocode above are numbered so that you can refer to specific lines in your reasoning.)

4. (3 points) Let L be the output of the greedy algorithm, and let i_1 be the first town in list L. Let L^* be an optimal solution for instance $(d, p, R[1..k])$, and let i_1^* be the first town in list L^*. Let L' be the list obtained from L^* by replacing i_1^* by i_1. Explain why L' is also an optimal solution for instance $(d, p, R[1..k])$.

5. (3 points) Complete the following argument that uses induction on k to show that the greedy algorithm outputs an optimal solution on any instance $(r, p, R[1..k])$ with a valid solution. You need to fill in the details for the base case, and parts (i) and (ii) of the inductive step. (The inductive hypothesis is done for you.)

   Base case: If $k = 1$ then...

   Inductive hypothesis: Let $k \geq 1$. The greedy algorithm outputs an optimal solution for any instance with $k + 1$ towns (assuming that the instance has a valid solution).

   Inductive step: Show that the greedy algorithm outputs an optimal solution when there are $k + 2$ towns along the route. (Refer back to earlier parts of this problem.)

   (i) There is an optimal solution that starts with i_1 because ...

   (ii) Once i_1 is added to the list, the remaining solution chosen by the Greedy algorithm is optimal because...

   Combining (i) and (ii), we can conclude that our algorithm finds an optimal solution for instance $(d, p, R[1..k])$.