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. Rin Meng, 51940633
2. Mika Panagsagan 29679552
3. Kevin Zhang 10811057

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):

   (a) DeepSeek R1, ChatGPT to generate ideas, explain concepts, and check if we are going in the right direction.

   (b) Google Search, Google's AI Overview

   (c) COSC 320 Lecture Slides

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

   ○ Yes ● No

4. One or more of us collaborated with other CPSC 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:

   Proof. Now we want to show that:

   $$\lim_{n\to\infty} \frac{\log_b n}{n^x} = 0$$

   Fix $b > 0$ and $x > 0$. Using the fact, and since $x/2 > 0$, it must be the case that $\log_b n = O(n^{x/2})$. This means that there exists a constant $c > 0$ such that for all n sufficiently large, we have $\log_b n \leq cn^{x/2}$. Now we can write:

   $$\frac{\log_b n}{n^x} \leq \frac{cn^{x/2}}{n^x} = \frac{c}{n^{x/2}}$$

   Now we can see that:

   $$\lim_{n\to\infty} \frac{c}{n^{x/2}} = 0$$

   Therefore, by the limit comparison test, we have that:

   $$\lim_{n\to\infty} \frac{\log_b n}{n^x} \leq \frac{c}{n^{x/2}} = 0$$

   therefore, we have shown that $\log_b n = o(n^x)$.

   □

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 $\log^3 n = (\log n)^3$.

   Proof. We wish to show that

   $$\sqrt{n} = o\left(\frac{n}{\log^3 n}\right),$$

   which, by definition, means that

   $$\lim_{n\to\infty} \frac{\sqrt{n}}{n/\log^3 n} = 0.$$

   Notice that

   $$\frac{\sqrt{n}}{n/\log^3 n} = \frac{\sqrt{n}\,\log^3 n}{n} = \frac{\log^3 n}{\sqrt{n}}.$$

By the fact from part 1, for every base $b > 1$ and every exponent $x > 0$, we have

$$\log_b n = o\left(n^x\right).$$

In particular, for $b = 10$ and $x = \frac{1}{3}$ we obtain

$$\log_{10}^3 n = o\left(n^{1/3 \times 3/2}\right) = o\left(n^{1/2}\right).$$

If

$$\frac{log_n^3}{\sqrt{n}} = \frac{o(n^{1/2})}{o^{1/2}} = o(1)$$

Based on the limit,

$$\lim_{n \to \infty} \frac{\log^3 n}{\sqrt{n}} = 0,$$

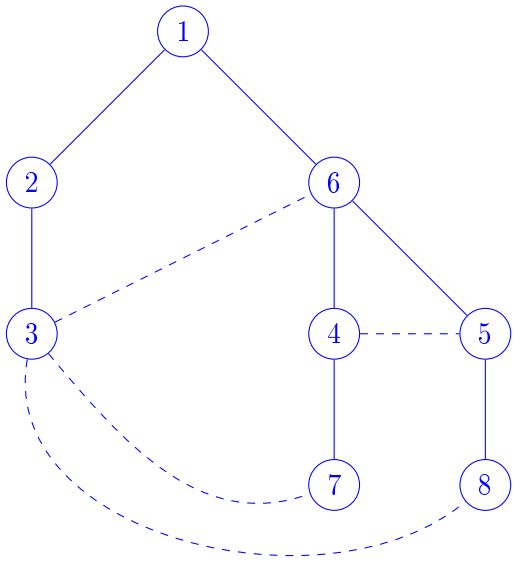
$\therefore$ We have shown that $\sqrt{n} = o(n/\log^3 n)$. $\square$

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.

$c(1, 7) = 3$ with paths $\{1, 2, 3, 7\}, \{1, 6, 3, 7\}, \{1, 6, 4, 7\}$

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, ..., n}, where n ≥ 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 ∈ V:
    for each node v ∈ V do
        c(1, v) ← 0
    end for
    c(1, 1) ← 1                          ▷ There is exactly one shortest path from 1 to itself.
    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] ← 0                                                          ▷ node 1 is at level 0
    for all nodes v > 1 do
        d[v] ← ∞                                                      ▷ v is not yet in the tree
    end for
    d ← 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] == ∞ then                                     ▷ v has not yet been added to the tree
                    d[v] ← 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):
                if d[v] == d then                                     ▷ update the count if v is at the current level d
                    c(1, v) ← c(1, v) + c(1, u)
                end if
                end for
            end for
            d ← 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:

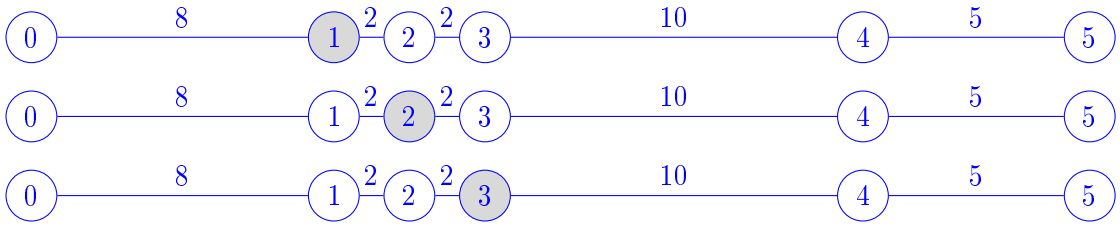
- $R[1..k]$: inter-town distances along the planned route, with $k \geq 1$. That is, there are $k+1$ towns along the route with town 0 being the start and town k being the destination; and $R[i]$ is the distance (in km) from town $i-1$ to town i for $1 \leq i \leq k$.

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 ← 0                                              ▷ 0 is the starting town
3:      L ← empty list
4:      while i < k do
5:              ▷ find the town j ≤ k that's furthest away from town i, among those of distance ≤ dp
6:          j ← i + 1
7:          d' ← R[j]
8:          while (j < k) and (d' + R[j + 1] ≤ dp) do
9:              j ← j + 1
10:             d' ← d' + R[j]
11:         end while
12:         if j < k then
13:             L ← L, j                                   ▷ append j to the solution L
14:         end if
15:         i ← 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.

Proof. We want to show that the list L given by GREEDY-PROVISIONS is a valid solution, which is the distance between consecutive stops (or between the start and the first stop, or the last stop and the destination) is at most dp.

At the beginning, the hiker is at town 0 with provisions enough to travel dp kilometers. In each iteration of the while loop, the algorithm sets the current town as i and then finds the furthest town j (with $j \leq k$) such that the total distance

$$R[i+1] + R[i+2] + \cdots + R[j] \leq dp.$$

Since we are given that the instance $(d, p, R[1..k])$ has a valid solution, there is at least one town reachable from town i within distance dp. Therefore, the inner loop always terminates with a valid choice of j.

If $j < k$, the algorithm adds j to the list L. This means that the distance from town i to town j is at most dp kilometers long, ensuring that the hiker can safely travel to j without running out of provisions. The algorithm then resets i to j and repeats.

Finally, when the algorithm reaches a point where $i = k$ (the hiker has reached the destination or can reach it from the last chosen stop within dp), we have ensured that every distance of the journey (from the start to the first stop, between consecutive stops, and from the last stop to the destination) is at most dp kilometers.

Thus, by always choosing the furthest reachable town in dp kilometers, the algorithm make sure that the hiker never face situations where their travel distance is longer than what their provisions allow. Therefore, the list L produced by GREEDY-PROVISIONS is indeed 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.)

Proof. Let $n = k$ denote the number of towns. We will see the running times below:

- The outer while loop (line 4) runs as long as $i < k$. In each iteration, the algorithm sets $j = i + 1$ (line 6) and then uses the inner while loop (lines 8–11) to extend j as far as possible such that the total distance from town i to town j is at most dp.
- In the inner loop, each iteration increments j by 1 (line 9), and the sum d' is updated in constant time (line 10). Since j can never exceed k, the total number of iterations of the inner loop across the entire execution is at most k.
- The outer loop itself moves the current position i to j (line 13), so even though the outer loop might iterate several times, each town is considered at most once in the inner loop.

Since all operations within both loops (arithmetic operations, comparisons, and list appends at line 12) are performed in constant time, the overall running time is $O(k)$.

Thus, the greedy algorithm runs in $O(k)$ time. □

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 $(d, p, R[1..k])$.

Proof. Let $L^* = (i_1^*, i_2^*, \ldots, i_r^*)$ be an optimal solution, so that each segment of the trip (from the start to i_1^*, from i_1^* to i_2^*, etc.) has total distance at most dp. In particular, the distance from town 0 to town i_1^* is at most dp.

The greedy algorithm selects i_1 as the farthest town reachable from the start while still keeping the distance within dp. Thus, by definition we have that

$$\text{distance}(0, i_1) \geq \text{distance}(0, i_1^*),$$

with the guarantee that $\text{distance}(0, i_1) \leq dp$.

Now, consider the list L' obtained from L^* by replacing i_1^* with i_1. Since i_1 is at least as far along the route as i_1^*, the following hold:

1. The segment from the start (town 0) to i_1 is at most dp (by the greedy selection).
2. For the remaining part of the route, observe that any segment in L^* following i_1^* was chosen so that the distance from i_1^* to the next provision town (or destination) is at most dp. Since i_1 is further along than i_1^*, the remaining segments, when shifted to start at i_1 instead of i_1^*, still have distances that do not exceed dp. (This is because if there were any issue with the segment from i_1 to the next stop, it would contradict the validity of L^* or the fact that the greedy algorithm extended as far as possible in the first move.)

Thus, the modified list L' remains a valid solution. Moreover, since L' has the same number of stops as L^*, it is also an optimal solution.

In summary, replacing i_1^* with i_1 does not increase the number of stops or the length of any segment; hence L' is an optimal solution for the 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 $(d, p, R[1..k])$.

Base case: If $k = 1$, there are two towns: town 0 (the start) and town 1 (the destination). The inter-town distance is $R[1]$. The greedy algorithm checks whether this distance is less than or equal to

dp. Since the instance has a valid solution, we are guaranteed that $R[1] \leq dp$. Then, no provision stop is needed, and the greedy algorithm outputs the empty list as the solution. Thus, the base case holds.

Inductive step: Assume that the greedy algorithm outputs an optimal solution for any instance with $k+1$ towns (i.e., k inter-town distances), given a valid solution. We must show that the greedy algorithm also outputs an optimal solution for an instance with $k+2$ towns.

(i) **There is an optimal solution that starts with i_1 because...** Recall from part 4 that i_1 is the farthest town from town 0 reachable within a distance of at most dp. By the argument in part 4, we can replace the first stop of any optimal solution with i_1 without losing our optimal solution, forming a new solution L'. Therefore, there exists an optimal solution that starts with i_1.

(ii) **Once i_1 is added to the list, the remaining solution chosen by the Greedy algorithm is optimal because...** By the greedy algorithm's design, after selecting i_1, the remaining instance has only $k+1$ towns (from i_1 to the destination). By the inductive hypothesis, the greedy algorithm finds an optimal solution for this reduced instance. Thus, the remaining solution is also optimal.

Conclusion: Combining (i) and (ii), we can conclude by induction that the greedy algorithm finds an optimal solution for any instance $(d, p, R[1..k])$ with a valid solution.