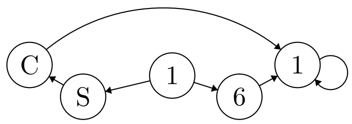
CS 161 **Homework 4**
Summer 2018 **Due:** Thursday 8/2 at 5 p.m. on Gradescope

Exercises

Exercises should be completed **on your own**.

Drawing graphs: You might try `http://madebyevan.com/fsm/` which allows you to draw graphs with your mouse and convert it into LaTeX code:



---

1. **(Fun with Dijkstra) (8 pt.)**

   Let $G = (V, E)$ be a weighted directed graph. For the rest of this problem, assume that $s, t \in V$ and that **there exists a directed path from** s **to** t. The weights on G could be anything: **negative, zero, or positive.**

   For the rest of this problem, refer to the implementation of Dijkstra's algorithm given by the pseudocode below.

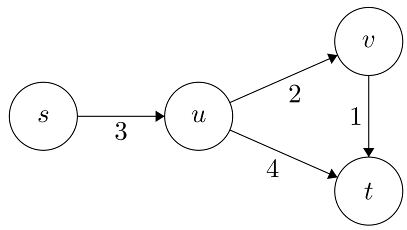
```
def dijkstra_st_path(G, s, t):
  for all v in V: set d[v] = Infinity
  for all v in V: set p[v] = None

  # we will use p to reconstruct the shortest s-t path at the end
  d[s] = 0
  F = V
  D = []
  while F is not empty:
    x = vertex v in F such that d[v] is minimized
    for y in x.outgoing_neighbors:
      d[y] = min(d[y], d[x] + weight(x,y))
      if d[y] was changed in the previous line: set p[y] = x
    F.remove(x)
    D.add(x)

  // use p to reconstruct the shortest s-t path
  path = [t]
  current = t
  while current != s:
    current = p[current]
    add current to the front of the path
  return path, d[t]
```

   Notice that the pseudocode above differs from the pseudocode in the notes. The variable `p` maintains the "**p**arents" of the vertices in the shortest s-t path, so it can be reconstructed at the end.

(a) **(1 pt.)** Step through `dijkstra_st_path`(G, s, t) on the graph G shown below. Complete the table below to show what the arrays `d` and `p` are at each step of the algorithm, and indicate what path is returned and what its cost is.



G

[We are expecting the table below filled out, as well as the final shortest path and its cost. No further justification is required.]

	`d`$[s]$	`d`$[u]$	`d`$[v]$	`d`$[t]$	`p`$[s]$	`p`$[u]$	`p`$[v]$	`p`$[t]$
When entering the first while loop for the first time, the state is:	0	∞	∞	∞	None	None	None	None
Immediately after the first element of D is added, the state is:	0	3	∞	∞	None	s	None	None
Immediately after the second element of D is added, the state is:								
Immediately after the third element of D is added, the state is:								
Immediately after the fourth element of D is added, the state is:								

(b) **(1 pt.) Prove or disprove**: In every such graph G, the shortest path from s to t exists. Here, a *path* from s to t is formally defined as a sequence of edges

$$(u_0, u_1), (u_1, u_2), (u_2, u_3), \ldots, (u_{M-1}, u_M)$$

such that $u_0 = s$, $u_M = t$, and $(u_i, u_{i+1}) \in E$ for all $i = 0, \ldots, M-1$. A *shortest path* is a path $((u_0, u_1), \ldots, (u_{M-1}, u_M))$ such that

$$\sum_{i=0}^{M-1} \text{weight}(u_i, u_{i+1}) \leq \sum_{i=0}^{M'-1} \text{weight}(u'_i, u'_{i+1})$$

for all paths $((u'_0, u'_1), \ldots, (u'_{M'-1}, u'_{M'}))$.

(c) **(2 pt.) Prove or disprove**: In every such graph G in which the shortest path from s to t exists, `dijkstra_st_path`(G, s, t) returns a shortest path between s and t in G.

(d) **(2 pt.) Prove or disprove**: In every such graph G in which there is a negative-weight edge, and for all s and t, `dijkstra_st_path`(G, s, t) does not return a shortest path between s and t in G.

(e) **(2 pt.)** Your friend offers the following way to patch up Dijkstra's algorithm to deal with negative edge weights. Let G be a weighted graph, and let w^* be the smallest weight that appears in G. (Notice that w^* may be negative). Consider a graph $G' = (V, E')$ with the same vertices, and such that E' is as follows: for every edge $e \in E$ with weight w, there is an edge $e' \in E'$ with weight $w - w^*$. Now all of the weights in G' are non-negative, so we can apply Dijkstra's algorithm to that:

```
modified_dijkstra(G,s,t):
  Construct G' from G as above.
  return dijkstra_st_path(G',s,t)
```

Prove or disprove: Your friend's approach will always correctly return a shortest path between s and t if it exists.

Problems

You can collaborate with your classmates about the problems. However:

- Try the problems on your own *before* collaborating.
- Write up your solutions yourself, in your own words. You should never share your typed-up solutions with your collaborators.
- If you collaborated, list the names of the students you collaborated with at the beginning of each problem.

---

1. **(Currency Exchange) (9 pt.)**

   (a) **(3 pt.)** Suppose the economies of the world use a set of currencies $C_1, \ldots, C_n$; think of these as dollars, pounds, Bitcoin, etc. Your bank allows you to trade each currency C_i for any other currency C_j, and finds some way to charge you for this service. Suppose that for each ordered pair of currencies (C_i, C_j), the bank charges a flat fee of $f_{ij} > 0$ dollars to exchange C_i for C_j (regardless of the quantity of currency being exchanged).

   Devise an efficient algorithm which, given a starting currency C_s, a target currency C_t, and a list of fees f_{ij} for all $i, j \in \{1, \ldots, n\}$, computes the cheapest way (that is, incurring the least in fees) to exchange all of our currency in C_s into currency C_t. Also, justify the correctness of your algorithm and its runtime.

   [**We are expecting a description or pseudocode of your algorithm as well as a brief justification of its correctness and runtime.**]

   (b) **(3 pt.)** Consider the more realistic setting where the bank does not charge flat fees, but instead uses exchange *rates*. In particular, for each ordered pair (C_i, C_j), the bank lets you trade one unit of C_i for $r_{ij} > 0$ units of C_j. Devise an efficient algorithm which, given starting currency C_s, target currency C_t, and a list of rates r_{ij}, computes a sequence of exchanges that results in the greatest amount of C_t. Justify the correctness of your algorithm and its runtime. [Hint: How can you turn a product of terms into a sum? Take logarithms.]

   (c) **(3 pt.)** Due to fluctuations in the markets, it is occasionally possible to find a sequence of exchanges that lets you start with currency A, change into currencies, B, C, D, etc., and then end up changing back to currency A in such a way that you end up with more money than you started with—that is, there are currencies $C_{i_1}, \ldots, C_{i_k}$ such that

   $$r_{i_1 i_2} \times r_{i_2 i_3} \times \cdots \times r_{i_{k-1} i_k} \times r_{i_k i_1} > 1.$$

   Devise an efficient algorithm that finds such an anomaly if one exists. Justify the correctness of your algorithm and its runtime.

2. **(Allocating Surfboards) (8 pt.)**

A group of n friends have respective heights $h_1 < h_2 < ... < h_n$ (where h_i is the height of friend i). They decide to go surfing and need to rent surfboards. The surf shop has a rack with $m > n$ surfboards ordered by lengths $s_1 < s_2 < ... < s_m$. In small/clean waves, the ideal surfboard has the same length as your height. Help us figure out a good allocation of the boards.

Formally, an allocation of surfboards is a function $f : \{1, ..., n\} \to \{1, ..., m\}$ that maps each surfer to a surfboard. More precisely, $f(2) = 3$ means that surfer 2 (with height h_2) receives surfboard 3 (with length s_3). An allocation f is optimal if it minimizes the quantity $\sum_{k=1}^{n} |h_k - s_{f(k)}|$. That is, an allocation is optimal if it minimizes the sum of the discrepancies of height between the surfers and their surfboards.

Let $A[n, m]$ denote this minimal difference.

(a) **(2 pt.)** Let $A[i, j]$ denote the sum of discrepancies of an optimal allocation of the first j surfboards to the first i surfers ($j \geq i$). Prove that, if surfboard j is used in an optimal allocation, then there is an optimal allocation in which it is allocated to surfer i.

Note: There might be multiple optimal allocations. This part asks you to show that if the longest board is used, then it might as well go to the tallest surfer.

[**We are expecting: A formal proof of your answer**]

(b) **(2 pt.)** Deduce a recurrence relation between $A[i, j], A[i, j-1]$ and $A[i-1, j-1]$.

Hint: Consider two cases, according to whether surfboard j is used or not.

[We are expecting: A statement of the recurrence as well as a short explanation of it.]

(c) **(3 pt.)** Design a dynamic programming algorithm that computes $A[n, m]$ and also outputs the optimal allocation.

[We are expecting: A description of a procedure or pseudocode of an algorithm.]

(d) **(1 pt.)** What is the runtime of your algorithm? Prove your answer.

[We are expecting: An informal analysis of the runtime.]