COMS 311: Homework 6
Due: Dec 5, 11:59pm
Total Points: 30

Late submission policy. If you submit by Dec 6, 11:59PM (and after the specified deadline), there will be 10% penalty. That is, if your score is x points, then your final score for this homework after the penalty will be $0.9 \times x$.

Submission after Dec 6, 11:59PM will not be graded without (prior) explicit permission from the instructors.

Submission format. Your submission file should be in pdf format. Name your submission file: `<Your-net-id>-311-hw6.pdf`. For instance, if your netid is `asterix`, then your submission file will be named `asterix-311-hw6.pdf`.

You are required to type-set your solution. You can use word, latex and any other type-setting tool to type your solution; you need to make sure you export/save it in pdf before submission.

For questions where you are asked to write an algorithm, you are required to write pseudocode. Do not write code in any specific language such as Java, Python, C/C++, etc. You are also required to present the runtime of your algorithm. Part of the grade will depend on the efficiency of your algorithm.

Algorithm Design Problems

- **When asked to design/write an algorithm, write pseudo code, not code of any specific programming language and not library operations of any specific programming language library.**
- For any algorithm design problem, present some justification for correctness of the algorithm you are presenting. For instance, checking whether there exists a subarray whose sum is 0 depends on computing prefix sums. This is because if there are two prefix sums that are equal then there exists a subarray whose sum is 0.
- For any algorithm design problem, you will present the runtime of your algorithm. The derivation does not need to be detailed but sufficient for explaining the final runtime.
- For all algorithm design problems, part of the grade depends on its runtime efficiency.

Learning outcomes.

1. Greedy Algorithms
2. Dynamic Programming

---

1. You own a car repair business, and each morning you have a set of customers whose cars need to be fixed. You can fix only one car at a time. Customer i's car needs r_i time to be fixed. Given a schedule (i.e., an ordering of n jobs), let f_i denote the finishing time of fixing customer i's car. For example, if job j is the first to be done, we would have $f_j = r_j$; and if job i is done right after job j, we would have $f_i = f_j + r_i$. Each customer i also has a given value v_i that represents his or her importance to the business. The happiness of customer i is expected to be dependent on the finishing time of i's job. So you have decided to schedule the jobs with the objective to minimize the weighted sum of the completion times, $\Sigma_{i=1}^{n} v_i f_i$.

   You have been told that to realize the optimization objective, the jobs should be scheduled in decreasing order of v_i/r_i. Prove/disprove that this scheduling strategy gives an optimal solution, i.e., minimizes $\Sigma_{i=1}^{n} v_i f_i$.

   First, let's assume there exists an optimal order that does not satisfy the greedy strategy suggested in the problem. This means that there exists a v_i/r_i that appears before v_j/r_j but $v_i/r_i < v_j/r_j$ (inversion pair). This means there exists an inversion pair that appear consecutively between d_i and d_j. Otherwise, this would result in $v_i/r_i > v_{i+1}/r_{i+1} > v_{i+2}/r_{i+2} ... v_{j-1}/r_{j-1} > v_j/r_j$, which means $v_i/r_i > v_j/r_j$ and that is a contradiction to our assumption that v_i/r_i and v_j/r_j are inversion pairs.

   Now, since we know there is a consecutive inversion pair either between v_i/r_i and v_j/r_j or they are a consecutive inversion pair themselves, we can use the exchange argument below. We will call the consecutive pair v_i/r_i and v_j/r_j.

   In the proposed optimal solution where the i and j jobs are consecutively scheduled, the weighted sum is...
   (O = weighted sum up until the i job, f = finish time of the job before i)
   $O + (f + r_i)v_i + (f + r_i + r_j)v_j = O + (f + r_i)v_i + (f + r_j)v_j + r_i v_j$

   If you exchange the i and j jobs but keep the rest of the jobs the same, the weighted sum is...
   (O = weighted sum up until the j job, f = finish time of the job before j)
   $O + (f + r_j)v_j + (f + r_j + r_i)v_i = O + (f + r_j)v_j + (f + r_i)v_i + r_j v_i$

   Reducing the weighted sums and from $v_i r_j < v_j r_i$, we get...
   $r_j v_i \leq r_i v_j$

   This is saying: that when we exchange inversions, it results in a weighted sum that is no more than the one with the inversion. Meaning, removing an inversion results in another optimal solution. As we continue to remove inversions, we get closer to the descending order that the greedy strategy suggests, proving that the greedy strategy produces an optimal ordering.

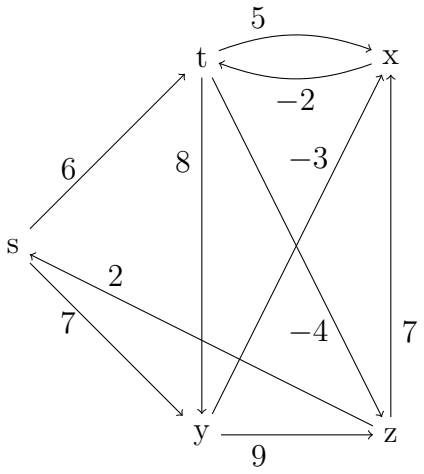
2. Write an algorithm to find the number of shortest paths from a given source vertex s to a given destination vertex t in a graph where the weights of the edges can be positive or negative integers. You can assume that the graph does not contain any negative weight cycles.

```
// edge = (u, v) MEANS edge from u to v!
numShortestPaths(graph G, vertex s, vertex t){
    for(all v in V)
        dict[0][v] = infinity
        numPaths[v] = 0
    for(i = 0 to (n-1))
        dict[i][s] = 0
    numPaths[s] = 1
    for(i = 1 to (n-1)){
        for(all v in V){
            dict[i][v] = dict[i-1][v]
            for(all edge = (u, v)){
                if(dict[i][v] > dict[i-1][u] + wt(u, v)){
                    dict[i][v] = dict[i-1][u] + wt(u, v)
                    numPaths[v] = numPaths[u]
                }
                else if(dict[i][v] == dict[i-1][u] + wt(u, v)){
                    numPaths[v] += numPaths[u]
                }
            }
        }
    }
    return numPaths[t]
}
```

Justification: The general idea is to run Bellman-Ford since we are evaluating a graph that could potentially have negative edge weights. Therefore, the majority of the algorithm is just the Bellman-Ford algorithm given during class. However, we need to keep track of the number of shortest paths to each vertex, so we do that with another array. All of the vertices start with 0 shortest paths (except for the source vertex, which starts with 1). We then run the shortest path algorithm to populate the dictionary in order for Bellman-Ford to function. If we have a shorter path than the one already found, we update the array containing the number of shortest paths to equal the number from the previous vertex. If the paths are equal, we add onto that array value with the number of paths from the previous vertex.

Runtime: O(VE) because the bulk of the algorithm depends on Bellman-Ford (which has a for loop (each edge) inside another for loop (each vertex)) and that multiplies those runtimes giving us O(VE).

3. Consider the following directed graph.



Apply Bellman-Ford algorithm to find the shortest distances to all vertices from the source s. Your solution must present the dictionary entries for each vertex for each iteration of the outer loop (as discussed in class Lecture).

iter	s	t	x	y	z
0	0	∞	∞	∞	∞
1	0	6	∞	7	∞
2	0	6	4	7	2
3	0	2	4	7	2
4	0	2	4	7	-2

4. **Extra Credit** Trinity is stationed at the gates of the last underground city of rebels. A swarm of robots are approaching the city. She needs to destroy some of these robots so that Neo has some chances of defending the city from the rest of the robots that enter the city. Trinity has the EMP generator, which can destroy the robots and the remote sensor, which informs her how many robots are coming toward the gate in the next n seconds.

   The EMP power depends on how long it has been charged. If j seconds pass since its last use (assume Trinity is charging since its last use), then the EMP is capable of destroying $f(j)$ robots. Every time the EMP is used, it gets fully discharged.

   Here is the problem: Trinity knows $x_1, x_2, \ldots x_n$ robots are coming toward the gate; x_i robots will arrive at the gate at the i-th second. The EMP starts off completely drained at the 0th second. That is, if it is used in the 1st second, then it is just charged for $(1-0)$ 1 second and can destroy at most $f(1)$ robots; if it is used for the first time in the 3rd second, then it is charged for $(3-0)$ 3 seconds and can destroy at most $f(3)$ robots. Also note that, if the EMP used in the k-th second and after that it is

used again at the l-th second ($l > k$), then the EMP is charged for $l - k$ seconds and is capable of destroying $f(l - k)$ rebots if used at the l-th second.

Trinity needs to figure out the maximum number of robots she can destroy.

Here is an example scenario: Let

$$x_1 = 1,\ x_2 = 10,\ x_3 = 10,\ x_4 = 1$$

and

$$f(1) = 1,\ f(2) = 2,\ f(3) = 4,\ f(4) = 8$$

In this scenario, the maximum number of robots Trinity can destroy is 5. This is because Trinity can use EMP in the 3rd second after charging for 3 seconds destroying $\texttt{min}(10, 4) = 4$ robots; then in the 4th second, EMP has only 1 second to get charged up and Trinity used it again to destroy 1 robot.

Write recursive characterization for the solution. Consider the function $OPT(i)$ that computes the maximum number of robots that can be destroyed for the problem instance where $x_1, x_2, \ldots x_i$ robots are arriving. We know the base case for the recursive function is $OPT(0) = 0$.

Write a DP (iterative) algorithm based on the the recursive characterization.

Recursive Definition:

$OPT(0) = 0$

$$OPT(i) = max \begin{cases} OPT(i-1) \\ max_{j=1}^{i}(OPT(i-j) + min(f(j), x_i) \end{cases}$$

Iterative Algorithm:

```
OPT(i){
    dict[0] = 0
    for(a = 1 to i){
        // Assign OPT(i-1) so we can compare using max in recursive
        dict[a] = dict[a - 1]
        for(b = 1 to a){
            if(dict[a] < dict[a - b] + min(f(b), x_a)){
                dict[a] = dict[a - b] + min(f(b), x_a)
            }
        }
    }
    return dict[i]
}
```

Justification: In my recursive definition, we have 2 options, either to use the EMP or not. If we do not use it, then we set the number of robots killed to be the number killed

in the last use. Otherwise, if we use it, find the max return which finds the highest killed at some point before and the total killed during the i-th use. Then, we simply converted this recursive definition into an iterative algorithm using a dictionary that stores each of the OPT values. Our function will return dict[i] since that will hold the value for OPT(i).

Runtime: $O(i^2)$ because we are doing the iteration over each of the dictionary entries for each value of a, which is quadratic.