COSC 311: ALGORITHMS
HW4: ALGORITHMIC PARADIGMS AND NETWORK FLOW

Due **Monday**, November 12 in class

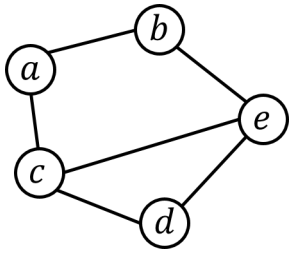
1 Submission and Collaboration

This assignment is due on Monday, November 12 at the start of class (note the extra weekend due to the midterm on Mon. 12/5); bring a hard copy of your typed responses to submit in class. **Each problem must be on a separate sheet of paper, with your name and section number at the top of the page.** If you use more than one sheet of paper for an individual problem, please staple together those pages. There is no need for you to rewrite the problems on your submission.

In accordance with this course's intellectual responsibility policy, you are welcome to discuss the problems with other students who are currently taking the course, but you must write up your solutions individually. Please note at the top of your submission with whom you discussed each problem.

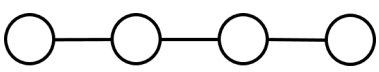
2 Problems

Problem 1: Independent sets.
Let $G = (V, E)$ be an undirected graph with n nodes. A subset of nodes is called an *independent set* if no two nodes in the subset are connected by an edge. For example, here's a graph:



The set of nodes $\{a, e\}$ is an independent set because there is not an edge between nodes a and e, but the set of nodes $\{b, c, d\}$ is not an independent set because there is an edge connecting nodes c and d. In general, it is hard to find large independent sets in graphs efficiently (and we will discuss this problem in much more detail in a few weeks!) But in some cases—namely if the graph is "simple" enough—it can be done.

We say that a graph is a *path* if its nodes can be written as $v_1, v_2, \ldots, v_n$ and there is an edge between every pair of vertices v_i and v_{i+1}. For example, here's a path graph on 4 nodes:



We will also associate with each node a positive integer weight (note that these weights are attached to *nodes*, not to edges). For example, here's a path graph with node weights:

In the *path weighted independent set* problem, we have:

Input: an undirected, unweighted path graph $G = (V, E)$ with $|V| = n$ nodes, and integer weight $w_i > 0$ on each node i

Output: a subset of nodes S such that no two nodes in S have an edge between them, and the total weight of the nodes in S, $W = \sum_{i \in S} w_i$, is maximized

(a) Give an example of a path graph on which the following "heaviest first" greedy algorithm does not produce the optimal solution.

```
1 HeaviestFirst(G = (V,E))
2     S = {}
3     while V isn't empty
4         pick the node v in V with maximum weight
5         add v to S
6         delete v and its neighbors from V
7     return S
```

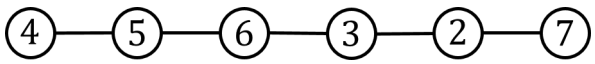
(b) Give an example of a path graph on which the following "even-odd" algorithm does not produce the optimal solution.

```
EvenOdd(G = (V,E))
    let S1 be the set of all nodes in an even position in the path
    let S2 be the set of all nodes in an odd position in the path
    return whichever of S1 and S2 has greater total weight
```

(c) Write a recurrence that expresses the optimal solution for a path graph of n nodes in terms of smaller subproblems.

(d) Translate your recurrence from part (c) into a dynamic programming algorithm that solves the path weighted independent set problem optimally; that is, your algorithm should take as input a weighted path graph, and output an independent set of nodes with maximum total weight. Your algorithm's runtime should be polynomial in n and should not depend on the node weights.

(e) Consider the following path graph:



Explain how your algorithm goes about finding the optimal solution for this graph. What solution does your algorithm produce?

(f) What is the runtime of your algorithm, in terms of n?

Problem 2: Maximizing profit.
Suppose we are given a list of prices for a given stock for n consecutive days, where the days are numbered $i = 1, 2, \ldots, n$. For each day, we are given the price per share p_i for the stock on that day (and we'll assume that the price stayed the same all day). We want to determine on what day i to buy the stock and on what day $j > i$ to sell the stock in order to maximize our profit per share, p_j/p_i. For the *stock profit* problem, we have:

Input: a list of prices $p_i > 0$ for each day $i = 1, 2, \ldots, n$

Output: a pair of days (i, j) on which to buy and sell the stock, such that $j > i$ and the profit p_j/p_i is maximized (if there is no possible way to make a profit, i.e., if $p_j/p_i \leq 1$ for all days i, j, then we should output "no solution" instead)

(a) There are several possible ways to solve this problem. We've discussed three different algorithmic paradigms: divide-and-conquer, greedy, and dynamic programming. If you were to design an algorithm to solve this problem, which paradigm would you choose? Explain why you believe your chosen paradigm is the best fit for this problem. You may want to discuss whether this problem satisfies the properties needed to apply a paradigm and attain an optimal solution, what runtimes you might be able to achieve, or anything else that might factor into your decision about what paradigms are well-suited to this problem.

(b) Give an algorithm that fits within the paradigm you chose in part (a) to solve the stock profit problem optimally for any input.

(c) Here's a list of prices:

i	1	2	3	4	5	6	7	8
p_i	4	24	2	16	20	1	3	9

Explain how your algorithm goes about finding the optimal solution for this particular input. What solution does your algorithm find?

(d) What is the runtime of your algorithm, in terms of n?

Problem 3: Blood transfusions.
Suppose you work for a hospital, and your job is to determine whether the current supply of blood will be enough to meet the projected demand over the next week for blood transfusions.

Unfortunately, it's not as simple as checking whether the number of liters currently available is greater than the number of liters required: people have *blood types* that constrain the types of blood they can receive. A person's blood supply has certain antigens present, and a person cannot receive blood that contains a particular antigen if their own blood does not have this antigen. Concretely, blood is divided into four types: A, B, AB, and O. Blood of type A has antigen A, blood of type B has antigen B, blood of type AB has both antigens, and blood of type O has neither. So a patient with type-A blood can receive only blood types A or O in a transfusion, a patient with type-B blood can receive only blood types B or O, a patient with type-AB blood can receive any of the four types, and a patient with type-O blood can only receive blood type O. In the *blood transfusion* problem, we have:

Input: s_A, s_B, s_{AB}, and s_O: the supply of each of the different blood types (assume these are all positive integers), and d_A, d_B, d_{AB}, and d_O denote the demand of each of the different blood types in the next week (again, assume positive integers)

Output: "yes" or "no" to the question: is it possible to meet the demand for each blood type? And if so, how?

(a) Give an algorithm to determine if the current supply of blood is enough to meet the demand. Your algorithm should model the blood transfusion problem as a network flow problem (i.e., represent the problem using a graph, and then run a max-flow algorithm on the resulting graph). In your algorithm description, it should be clear (i) what the vertices and edges represent in the graph you create, (ii) how, after running a max-flow algorithm on the graph you created, you can determine if the current blood supply is sufficient, and (iii) if the blood supply is sufficient, how the available blood should be allocated to patients with each blood type.

(b) Consider the following example:

blood type	**supply**	**demand**
O	50	45
A	36	42
B	11	8
AB	8	3

Show how your algorithm works on this example. You should draw a picture of the graph you create and describe one possible run of Ford-Fulkerson on the graph.

(c) Your algorithm from part (a) should output two things: a yes/no answer and (if yes) an allocation of blood supply to patients that meets the demand for all blood types. Suppose you run your algorithm and learn that the available blood supply is not enough to meet the demand in the next week. Even though your algorithm outputs the answer "no," you could imagine outputting the latter part of the solution anyway. In this case, this piece of the solution doesn't give an allocation that meets the full demand. But it isn't entirely meaningless. What do you learn from the output if it was not possible to meet the full demand?