Homework 6:

Due May 1 at midnight anywhere on Earth. Submit your solutions typed and in a pdf document. To receive full credit, explain your answers.

If you collaborate with another student or use outside sources, please list those students' names and the URL/title/etc. of the sources that you referred to. Collaboration is permitted, but you must write up your own solutions.

Problem 1: In class, we learned that if a network has edges with integer capacities, then the Ford-Fulkerson algorithm will find an integer flow. Suppose we have a network in which all capacities are multiples of 4. Will the Ford-Fulkerson algorithm find a flow such that the amount of flow sent along each edge is a multiple of 4? If yes, explain why; if no, give a counterexample.

Problem 2: There are n patients who are sick and need to see a doctor. These patients have different illnesses: e.g., some have heart disease and need to see a cardiologist; others have a cold and need to see a general practitioner. Each patient p_i must be seen for a total of h_i hours: for example, one might need a 10-hour surgery while others might only need a 1 hour appointment. It is ok to split this time between multiple doctors, as long as they are all able to treat the illness: for instance, in the 10 hour surgery, one doctor can do 1 hour of the surgery, another could do 7, and a third could do 2.

There are m doctors, and each doctor is qualified to treat certain illnesses, but not necessarily all (for example, some doctors might be able to treat allergies and asthma, but not heart disease). Additionally, each doctor D_j has a cap C_j on the total number of hours that she is able to work. Assume all h_i and C_j values are positive integers. Your goal is to determine how to assign doctors to patients so that each patient is seen for the required number of hours, and by doctors who are qualified to treat her.

(a) Design an algorithm to solve this problem by using network flows. Be precise about the structure of the graph. It is sufficient to explain how to create the graph, and then say to apply an existing network flow algorithm to the graph.

(b) Suppose now that a patient is not allowed to split their time among multiple doctors. Does your solution still work? If yes, explain why. If no, give a counterexample.

Problem 3: Recall the stack with MultiPop from class. In the problem from class, a stack had three operations: Push, Pop, and MultiPop. In the MultiPop operation, the user specifies a value k, and k elements are popped from the stack. Each Push costs \$1, each Pop costs \$1, and each MultiPop cost \$$k$.

We change the problem as follows: In the new version of the problem, each MultiPop costs \$$2k$, instead of \$$k$ (i.e., \$2 per element that gets popped). Push costs \$2, and Pop costs \$3. Perform an amortized running time analysis on this problem.

Problem 4: In class, we discussed a stack with three operations: Push, Pop, and MultiPop(k). The MultiPop(k) operation is implemented as a series of k pops. Push and Pop each take 1 unit of time, and MultiPop(k) takes k unit of time. Using amortized analysis, we determined that the average cost of an operation, over n operations, was 2 units of time.

(a) Suppose that in addition to MultiPop(k), we want to implement a new operation MultiPush(k), which is a series of k pushes. Can we modify our amortized analysis to this case? Does amortized analysis help in this case? Why or why not?

(b) Suppose now that we're only allowed to call MultiPush(k) *immediately* after calling a MultiPop(k), where the k for the MultiPush must be no greater than the k for the MultiPop. Now can we modify our amortized analysis for this case? Why or why not? If yes, do it.

Problem 5: Recall the *extensible array* problem from class. In this problem, we define the extensible array data structure, an array in which the user does not need to specify the size at initialization. The only operation supported by this data structure is 'Insert', which adds an element. The array initially contains only one element. If you try to insert into a full array of size k, the operating system allocates $2k$ units of space elsewhere and copies over the existing elements. Inserting an element costs 1 unit of time. It also takes 1 unit of time to copy each existing element into the new location. Allocating memory is free.

In the in-class version of this problem, these were the only costs associated with the extensible array. Here, we add a new cost: whenever we reallocate space, we must *clean up* the previous location. it costs 1 unit of time to delete each element from its old location.

Use amortized analysis to calculate the average running time for a sequence of N Insert operations on this modified extensible array.

Extra Credit: Consider the extensible array data structure that we learned in class. Consider the version in which there is no extra cost for allocating new memory (the first version we looked at). Suppose that we want to add another operation to this data structure: Remove, which deletes the last element added. In order to make sure that the array doesn't take up too much space, we say that if the array is at least half empty, we will reallocate memory that is only half the size of the current array, and copy all the elements over. This is basically the opposite of the insertion operation from before.

Part a: Does amortized analysis make sense here? In other words, does it allow us to get a tighter bound than the standard analysis?

Part b: Instead of reallocating memory if the array becomes half empty, we reallocate/copy if the array becomes at least 3/4 empty (only 1/4 of the array cells are being used). Analyze the running time of a sequence of n operations using amortized analysis. Hint: this is a straightforward modification of the original analysis. If we shrink an array, how many elements get added before we next double it? If we are deleting elements, how many do we delete before shrinking it?