COMPSCI 130 Algorithms
Practice Problems

1. Prove or disprove the following: if the frequencies of an alphabet monotonically decrease, then the optimal Huffman code lengths for the alphabet monotonically increase.

   This is false; consider a small alphabet with frequencies {5,4,3,2,1}. Even ignoring the fact that the last two symbols will necessarily have the same code length, it is easy to see that the optimal coding assigns the same length to the characters with frequencies 4 and 5. However, it is provably true that the optimal Huffman code lengths are non-decreasing.

2. Suppose that we want to implement a stack using a dynamic array, but instead of doubling the size when the array is full, we decide to multiply the size of the array by 3/2. Assume that increasing the size of the array from n to $3n/2$ costs n. Thus, the actual cost of a push is 1 if the array was not full, and $(1 + n)$ if it was full.

   (a) What is the amortized cost of a push?

   Despite choosing to increase the size to 3n/2 instead of doubling, the amortized time of a push is still $O(1)$. You can see this using the accounting method, giving each item \$4 when it is pushed on to the stack. \$1 is paid immediately for the push; when the stack is filled, 1/3 of the items on the stack have never been copied and retain \$3, which covers the cost of copying all items on the stack.

   (b) Suppose that we want to make sure that the array never becomes less than 2/3 full. One way to do this is to shrink the array from size n to $2n/3$ whenever we pop from an array that is 2/3 full. Does this yield an amortized cost of $O(1)$ per operation?

   No; consider what happens if we add to a full array of size n, increasing its size by $3n/2$, with a cost of copying n items. If we then pop the last item, we immediately decrease the size of array once again to n. (Depending on the mechanism used to decrease the array size, this may incur another cost of copying n items.) If we continue to push and pop, alternately, we incur a cost of n copies every other operation, for an amortized cost of $n/2$, or $O(n)$.

   (c) Another approach is to reduce the size from n to $5n/6$ whenever we pop from an array that is 2/3-full. Does this yield an amortized cost of $O(1)$?

   Yes; if we consider the worst case of popping right after a push increases the size of the array, then we still have a growth of $1n/6$ over n. Using the accounting method, we have to pay each item \$7 to cover the costs of copying, which is still $O(1)$. (This assumes that there is no cost to decreasing the size of the array.)

3. You are traveling by a canoe down a river and there are n trading posts along the way. Before starting your journey, you are given for each $1 \leq i < j \leq n$, the fee $f_{i,j}$ for renting a canoe from post i to post j. These fees are arbitrary. For example it is possible that $f_{1,3} = 10$ and $f_{1,4} = 5$. You begin at trading post 1 and must end at trading post n (using rented canoes). Your goal is to minimize the rental cost. Give the most efficient algorithm you can for this problem. Be sure to prove that your algorithm yields an optimal solution and analyze the time complexity.

   Let $m[i]$ be the rental cost for the best solution to go from post i to post n for $1 \leq i \leq n$. The final answer is in $m[1]$, and $m[n] = 0$. The optimal cost for traveling starting at post i is given by $m[i] = \min_{j:i<j\leq n}(m[j] + f_{i,j})$.

The canoe must be rented starting at post i (the starting location) and then returned next at a station among $i+1, \ldots, n$. We try all possibilities for the next (with j being the station where the canoe is next returned). For the time complexity there are n subproblems to be solved each of which takes $O(n)$ time. These subproblems can be computed in the order $m[n], m[n1], \ldots, m[1]$. Hence the overall time complexity is $O(n^2)$.

4. Let $G = (V, E)$ be a flow network with source s, sink t and integer capacities. Suppose that we are given a maximum flow in G.

   (a) Suppose that the capacity of a single edge $(u, v) \in E$ is increased by 1. Give an $O(V + E)$ time algorithm to update the maximum flow.

   Compose the residual graph on the original flow. Add a positive 1 capacity on the edge that has been increased. Using BFS, search for an augmenting path; if the path exists, we can update the flow, otherwise, the flow is unchanged. We only need to do this once, as the augmenting path, if it exists, increases the flow by 1, which is the maximum increase possible.

   (b) Suppose that the capacity of a single edge $(u, v) \in E$ is decreased by 1. Give an $O(V + E)$ time algorithm to update the maximum flow.

   Again, compose the residual graph on the original flow. If the decreased edge was not at capacity (that is, it still has positive residual capacity), then we can decrease the edge capacity by one without affecting the maximum flow. If not, then we add one to the negative capacity on the edge, and look for an augmenting path in reverse (going from t to s instead of from s to t) which includes the decreased edge.

5. Suppose someone presents you with a solution to a max-flow problem on some network. Give a linear time algorithm to determine whether the solution does indeed give a maximum flow.

   First, verify that the solution is a valid flow by comparing the flow on each edge to the capacity of each edge, for cost $O(|E|)$. If the solution is a valid flow, then compose the residual graph ($O(|E|)$) and look for an augmenting path, which using BFS is $O(|V| + |E|)$. The existence of an augmenting path would mean the solution is not a maximum flow.

6. Prove that the following problem, the Non-bored Joggers Problem (NBJP), is NP-complete, by reduction from Subset Sum. Recall that in Subset Sum you are given a set of integers and a value k, and asked if there is a subset of the integers which sum exactly to k.

   In NBJP, you are given as input a weighted undirected graph G (loops and multiple edges are allowed and the weights are positive integers), a specified node v, that is called home, and an integer ≥ 0. You must determine if there exists a route for the jogger (i.e. a path in G) that starts at vertex v, never repeats an edge, and returns to v after travelling a distance of exactly l.

   First, NBJP is clearly in NP since we can verify the solution by simply tracing the path given as a solution, adding up the edge weights, for cost at most $O(|E|)$. Next we reduce Subset Sum to NBJP in the following manner: we construct a graph with the single home node, v, with loop edges (self edges), one per integer in the Subset Sum input, with weights equal to the integers. (Note that this only works if we use Subset Sum as defined in the book, where the integers are all positive.) Finally, we set the jogger's target distance equal to the target value k for the Subset Sum problem. Since edges cannot be repeated in the solution to NBJP, any valid path for the jogger which matches the target value requires the jogger to transverse edges which can be seen as selecting integers from the Subset Sum input which sum to k. Construction of the NBJP is straightforwardly polynomial, since the number of edges is the same as the number of integers in the input. Recovery of the solution is similarly straightforward.