This problem set is worth 5% of your mark, and it is important to do your own work. You may consult with others concerning the general approach for solving problems on assignments, but you must write up all solutions entirely on your own. Anything else is *cheating* and is subject to prosectuion under the University's Code of Behavior. This homework is due at the beginning of tutorial on Wednesday, October 4. Late assignments will receive a grade of 0. This homework will help to prepare you for the first test, which will take place during tutorial on October 11.

1. Suppose the runtime of an algorithm is $T(n)$. Circle the expressions below for $T(n)$ which would make the algorithm polytime.

   (a) 2^{n^2}—Not polytime: assume there exists $c, N, k > 0$ such that $cn^k \leq 2^{n^2}$ whenever $n \geq N$. But $cn^k = 2^{\log c + k \log n}$, so as long as $n > 2\max\{\sqrt{\log c}, k\}$, then $n^2 > \log c + k \log n$ and $2^{n^2} > cn^k$.

   (b) $n \log_2 n$—Polytime: if $n \geq 1$, then $n \geq \log_2 n$, so $n^2 \geq n \log_2 n$.

   (c) $3^{\log_2 n}$—Polytime: $\log_2 n = \log_3 n / log_3 2$, so $3^{\log_2 n} = n^{1/log_3 2} < n$ for all $n \geq 1$.

   (d) $n^{\log_2 n}$—Not polytime: assume there exists $c, N, k > 0$ such that $cn^k \geq n^{\log_2 n}$ whenever $n \geq N$. But, if $n > \max\{c, 2^{k+1}\}$, then $cn^k < n^{k+1} < n^{\log_2 n}$.

2. True or False:

   (a) $2^n \in O(3^n)$—True: $2^n \leq 3^n$ for all $n \geq 1$.

   (b) $3^n \in O(2^n)$—False: $3^n = (1.5 * 2)^n = 1.5^n 2^n$. But there are no constants c, N such that $c \geq 1.5^n$ for all $n \geq N$.

   (c) $n + 100n \log n \in O(n)$—False: if $cn \geq n + 100n \log n$ for all $n \geq N$ for some $c, N > 0$, then we would have $c \geq 100 \log n$ for all $n \geq N$.

   (d) $n + 100n \log n \in O(n^2)$—True: for $n \geq 2^{20}$, $100 \log n < n$, so $100n \log n < n^2$. Since $n < n + 100n \log n$, $n + 100n \log n < 2n^2$.

3. Prove Lemma 1 of the lecture notes on Greedy Algorithms.

   Solution:
   The lemma states that the greedy solution $\bar{M}$ is at least half the optimal solution M. As suggested, we will show that $\bar{M} > 1/2C$, assuming that $\sum_{i=1}^{d} w_i > 1/2C$ (that is, all the objects together must fill more than half the knapsack).

   Assume for the sake of contradiction that we have $\bar{M} \leq 1/2C$. But then, consider an object i that we didn't add to the knapsack. We know $w_i > 1/2C$, otherwise we would have added it since we know it fits. On the other hand, if $w_i > 1/2C$, it must precede any of the objects that we did add in order of non-decreasing weight. So, it should have been the first object added. Therefore, we have a contradiction.

4. (a) Suppose that the edge costs are all distinct in an input G to the MST problem. Prove that G has a unique minimum cost spanning tree. Do this by considering the *proof* that Kruskal's algorithm always finds a minimum cost spanning tree.

Solution:
We know that Kruskal will produce a minimum spanning tree, we just need to show that no other spanning tree has minimum cost. Consider any two spanning trees T_1 and T_2. Let e be the lowest cost edge that appears in exactly one of these spanning trees (say it is in T_1, without loss of generality). Then, by the Exchange Lemma, there is e' in $T_2 - T_1$ such that $T_2' = T_2 \cup \{e\} - \{e'\}$ is a spanning tree. Since $c(e) < c(e')$, T_2' has lower cost than T_2. So, for any two spanning trees, both of them cannot be optimal.

(b) Now suppose that all edge costs are distinct, except $c_1 = c_2$, where $c_1 = c(e_1)$ and $c_2 = c(e_2)$ are the two minimum edge costs. Prove or give a counter-example: G has a unique minimum cost spanning tree.

Solution:
In this case, Kruskal's algorithm will certainly choose e_1 and e_2 to be in the spanning tree. In fact, let T_1 be any spanning tree including e_1 and e_2 and T_2 a spanning tree not including both. Then, by the above argument, we see that T_2 is not optimal. Therefore we can consider only spanning trees T_1 and T_2 that both contain e_1 and e_2. Then, since all the rest of the edges have distinct costs, the same argument again shows that there is a unique minimum spanning tree.

(c) Finally suppose that all edge costs are distinct, except $c_1 = c_2 = c_3$, where $c_1 = c(e_1), c_2 = c(e_2), c_3 = c(e_3)$ are the three minimum edge costs. Prove or give a counter-example: G has a unique minimum cost spanning tree.

Solution:
Consider a cycle on three nodes (a triangle) where each edge has the same cost. Then, any set of two edges forms a minimum spanning tree.

5. The Longest Monotone Subsequence problem is defined as follows. The input is a sequence $a_1, a_2, \ldots, a_n \in N$. The output is a number L where L is the length of the longest monotone (nondecreasing) subsequence of $a_1, \ldots a_n$. Note that the subsequence need not be consecutive. For example, if the input is the sequence

$$7, 5, 2, 10, 4, 10, 4, 12, 9$$

then a monotone subsequence is

$$2, 4, 4, 9$$

We can give a dynamic programming solution to this problem as follows. First, define an array $R(1 \ldots n)$ where for $1 \leq i \leq n$

$$R(i) = \text{ length of longest monotone subsequence of } a_1, \ldots a_i \text{ ending with } a_i$$

(a) Give a recurrence (including initialization) which can be used to compute the array $R(1 \ldots n)$.

Solution:
Clearly $R(1) = 1$. If $i > 1$, there are two cases. If $a_j > a_i$ for each $j < i$, then $R(i) = 1$. If on the other hand there is at least one $a_j \leq a_i$ for $j < i$, then that a_j could be the second last element in a monotone subsequence ending at a_i. To find the length of the longest monotone subsequence ending at a_i, we

take the maximum over all such candidates a_j of 1 + the length of the longest monotone subsequence ending at a_j.

By making the convention that max{empty set} = 0 we can combine all these cases into one. Here is the recurrence:

for $1 \le i \le n, R(i) = 1 + \max\{R(j) \mid j < i \text{ and } a_j \le a_i\}$

(b) Give (at a high level) an algorithm which evaluates the array using your recurrence.

Solution:

for i : $1..n$
    $max \leftarrow 0$
    for j : $1..i-1$
        if ($a_j \le a_i$ and $max < R(j)$) then
            $max \leftarrow R(j)$
        end if
    end for
    $R(i) \leftarrow max + 1$
end for

(c) Give an efficient high-level algorithm to find the solution L (and the subsequence) to the problem from the array $R(1 \dots n)$. You may output the subsequence in reverse order, if you prefer.

Solution:

It is possible to give a recursive procedure that outputs an optimal subsequence, using the array, in time $O(n)$. Here we present a sequential algorithm which outputs an optimal subsequence in reverse order, also in time $O(n)$

First it is necessary to find the length L of the optimal subsequence and an index K such that $R(K) = L$. Here is a program that finds L and K.

$L \leftarrow R(1)$
$K \leftarrow 1$
for i : $2..n$
    if $L < R(i)$ then
        $L \leftarrow R(i)$
        $K \leftarrow i$
    end if
end for

Now we sweep through the array in reverse order, starting at position K, and print out the subsequence.

print a_K
$last \leftarrow a_K$ % $last$ is the most recent element printed.
$goal \leftarrow L - 1$ % $goal$ is the number of remaining elements to be printed
for decreasing i : $K - 1$ downto 1
    if ($R(i) = goal$ and $a_i \le last$) then
        print a_i
        $last \leftarrow a_i$
        $goal \leftarrow goal - 1$

end if
end for

(d) Use big-O notation to estimate the runtime of your algorithm.

Solution:

The algorithm for filling in the array $R(1..n)$ has two nested for loops, each iterated at most n times. Since the assignment statements and comparisons take constant time, it follows that the runtime $T(n) = O(n^2)$.

Outputting the solution (finding L and going backwards through the array) simply steps through R one space at a time. Therefore they each take time $O(n)$ and together they still take time $O(n)$.

6. Let f^* be any maximum flow in a network G and let f be any non-maximum flow in G.

(a) Prove that there is an augmenting path p for f which satisfies the condition

$$f(u, v) < f^*(u, v) \text{ for all } (u, v) \text{ in } p$$

(Hint: study the proof of Theorem 27.7 in CLR, showing that every maximal flow f has an augmenting path. Modify the proof to show that f must have an augmenting path satisfying the above condition.)

Solution:

The parts of the proof of Theorem 27.7, page 593, (2) $\Longrightarrow$ (3) and (3) $\Longrightarrow$ (1) together imply that if the residual network G_f contains no augmenting path, then f is a maximum flow in G. We now modify these two proofs to show that if the residual network G_f contains no augmenting path *satisfying the condition (1)*, then f is a maximum flow in G. (This will establish (a); or equivalently, the contrapositive of (a).)

Define

$$S = \{v \in V : \text{there exists a path } p \text{ from } s \text{ to } v \text{ in } G_f \text{ that satisfies condition (1)}\}$$

and let $T = V - S$. The partition (S, T) is a cut, since $t \notin S$ by our assumption that there is no augmenting path in G_f satifying condition (1). For each pair of vertices u and v such that $u \in S$ and $v \in T$ we have

$$f(u, v) \geq f^*(u, v) \tag{1}$$

since otherwise we could conclude $v \in S$ from the assumption $u \in S$. ¿From (2) we conclude

$$f(S, T) \geq f^*(S, T)$$

That is, the flow of f across the cut (S, T) is at least as great as the flow of f^* across (S, T). But by Lemma 27.5, this implies that $|f| \geq |f^*|$, so f is a maximum flow.

(b) Find an example of a network G with a maximum flow f^* and a non-maximum flow f such that f has an augmenting path p which does *not* satisfy the condition above. (Note that this does not contradict what you proved in part (a) because your example will have another augmenting path that does satisfy the condition.) (Hint: There is a four-node diamond shaped example like those on page 596 in the text. You just need to find the right capacities and the right flows.)

Solution:

Consider the network in figure 27.2, page 595. Let f^* be the maximum flow shown on the right side of (d). Let f be the flow shown on the right side of (a), and let the augmenting path for f be the one shown in the left side of (b). Then $f(v_1, v_2) = 0 > -1 = f^*(v_1, v_2)$, so condition (1) is violated.

For a simpler example, consider the diamond shaped network on the top of page 596, where we alter the capacities so that

$$c(s, u) = c(u, v) = c(v, t) = 3 \text{ and } c(s, v) = c(u, t) = 2$$

Then the maximum flow f^* has the flow through every edge equal to its capacity, except that $f^*(u, v) = 1$. Let f be the non-maximum flow defined by

$$f(s, u) = f(u, v) = f(v, t) = 2 \text{ and } f(s, v) = f(u, t) = 0$$

Let p be the augmenting path s, u, v, t. Then

$$f(u, v) = 2 > 1 = f^*(u, v)$$

so condition (1) is violated.