Problem Set 6

All parts are due Thursday, December 3, 2015 at 11:59PM.

Name: David Walter

Collaborators: None

Part A

Problem 6-1.

(a) Newton's method is defined as

$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$.

and in our case,

$f(x) = x^2 - x - 1$

so the formula for the answer can be written as

$x_{n+1} = x_n - \frac{x_n^2 - x_n - 1}{2x_n - 1}$.

(b) $\phi = \frac{1+\sqrt{5}}{2}$

$$x_{n+1} = \phi - \frac{\phi^2 - \phi - 1}{2\phi - 1}$$

$$= \frac{1+\sqrt{5}}{2} - \frac{(\frac{1+\sqrt{5}}{2})^2 - \frac{1+\sqrt{5}}{2} - 1}{2\frac{1+\sqrt{5}}{2} - 1}.$$

$$= \frac{1+\sqrt{5}}{2} - \frac{(\frac{1+2\sqrt{5}-2}{4}) - \frac{2+2\sqrt{5}}{4} - \frac{4}{4}}{2\frac{1+\sqrt{5}}{2} - 1}.$$

$$= \frac{1+\sqrt{5}}{2} - \frac{\frac{0}{4}}{2\frac{1+\sqrt{5}}{2} - 1}.$$

$$= \frac{1+\sqrt{5}}{2} - 0 = x_{n+1} = \phi.$$

Problem 6-2.

(a) We will construct a bottom-up solution for this problem, start by initializing a $(b$ x $a)$ '2-D' array, call it $memo$, which we will use to store the values of $S(a_i, b_i)$ for $a_i = a_1, a_2, ..., a_a$ and $b_i = b_1, b_2, ..., b_b$, where a and b are defined in the problem. We will index $memo$ starting at 1, so $memo[a_i][b_i] = S(a_i, b_i)$. We will begin by finding $S(a_i = 1, b_i)$ for all b_i. To do this we interate through A starting with $S(1, 1)$ and $A[1]$ setting our $maxS = A[1]$, and setting $memo[1, 1] = maxS$. For all other $A[b_i]$ we will see if $A[b_i]$ is greater than $maxS$ and if so, we will set $maxS = A[b_i]$, and $memo[1, b_i] = maxS$. $S(1, b)$ will be the maximum of A. Now, we will compute $S(a_i = 2, b_i)$ for all b_i. We know that $S(a_i, = 0)$ to $S(a_i, a_i * D)$ does not have an answer, since the first $a_i * D$ days cannot possibly fit a_i psets. So starting at $S(a_i, a_i * D + 1)$ we will see if $A[b_i] + S(a_i - 1, b_i - D) > S(a_i - 1, b)$ (where $S(a_i - 1, b_i - D)$ and $S(a_i - 1, b)$ are stored in $memo$) and if so, we will set $maxS = A[b_i] + S(a_i - 1, b_i - D)$ and $memo[a_i, b_i] = maxS$, if not, we will not change $maxS$ and still set $memo[a_i, b_i] = maxS$. We will do similarly for $S(a_i > 2, b_i)$ up until we have reached our goal of $S(a, b)$. So in the end, $S(a, b) = max(S(a-1, b-D)+A[b], S(a-1, b))$

(b) To solve Prof. Smith's problem, $T(k, n)$, we will run $S(k, n)$ in the same way that we run $S(a, b)$ in part(a). The only difference is that every time we set $maxS$ and $S(a_i, b_i)$ we will also store b_i, namely the index of the element in A that creates the $maxS$ for $S(a_i, b_i)$. Each $S(a_i, b_i)$ will have one index stored along with the current score. In this case, $S(a_i, b_i) = (maxS, b_{index})$. To get all of the indices in A that form the answer, we will start with $S(k, n)[1]$ as the first index, and recursively finding $S(a_i - 1, S(a_i, b_i)[1])[1]$ until we get to $a_i = 1$. Once we have all k indices in A, $T(k, n)$ will return list of tuples formatted as following: $(index, A[index])$. This algorithm runs in $O(nk)$ time because we will go through each n index of A, k times to find $S(k, n)$, and recursively finding all indices for $T(k, n)$ takes $O(k)$ time.

Part B

Problem 6-3.

(a) We will start by hashing every element in L, so we can access, and check for the existence of each element in constant time. We will call this hash table D and creating D takes $O(d)$ time. Also, as we are going through L we will keep track of the length of the maximum word in L, namely we will find t as defined in the problem. Now we will start a bottom-up dynamic programming solution for this problem. We will start by creating a n length list of zeroes, called $memo$, which we will use to store the answer to each problem from $s[0]$ to $s[n_i]$. We will iterate from $n_i = 0$ to $n_i = n + 1$, and for each n_i we will iterate from $t_i = 0$ to $t_i = t$. At each nt iteration of the loop, we will be checking many things. First we will set $left = n_i$ as this will be the left index of the current word in S, and we will set $right = n_i + t_i$, as this will represent the right index of the word we are currently at in S. Now for each of these $S[left, right + 1]$ words we will first check if it is a valid word by checking if the current word in in D. If so, then we will go to the next level of checking. We will next check if $left = 0$ and if so we know that there is not a previous word that came before the current word. Otherwise we know there is a previous sentence at $memo[left - 1]$ We will also be checking if $memo[right] = 0$ and if not, we know that there is a previously found solution to the problem from $s[0 : right]$. So if we are checking $s[0 : right + 1]$ then we will simply check if there is a previous found solution and if there is compare the scores, and set the answer to the $maxScore$ and the left index of the current word as $memo[right] = (left, maxScore)$. Where $maxScore$ is calculated as the cube of the length of the current word. Now, if not $left = 0$ then we just do the same as specified earlier, execept $maxScore = memo[left - 1][1] + len(currentWord)^3$, and we will compare scored if there is a previous answer at $memo[right][1]$. After going through nk times, we would have found whether a sentence exists at memo[-1], and if so, memo[-1] will have the $maxScore$ and the index of the left of the last word in the solution. To construct the answer, we will initially set $right = n$ and $ans = []$, and $left = memo[right - 1][0]$. Now we will iterate, and append $s[left : right]$ to ans and make $right = left$ until $left! = 0$. Constructing ans takes $O(n)$ time. Now ans will be a backwards list of all of the words that we want to return, so we will reverse this list in $O(n)$ time. And finally we will use python's $.join()$ function to construct a string composed of every word in our answer with a space between each word. In the end we will either return the string representation of our sentence answer, or return None. This algorithm takes $O(nk + d)$ time. Constructing D takes $O(d)$ time and running the dynamic programming portion of our solution and finding the score and index of the last word in s at $s[-1]$, takes $O(nk)$ time, and construction our string representation of our answer takes $O(n)$ time. So our solution takes $O(d) + O(nk) + O(n) = O(nk + d)$ time.

(b) *Python script submitted.*