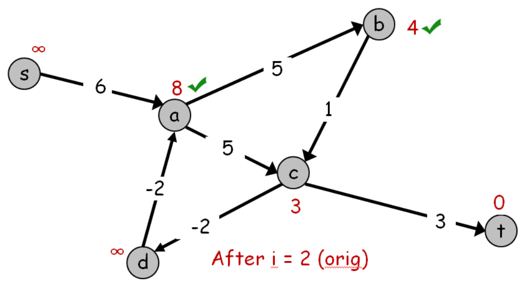
CSC373— Algorithm Design, Analysis, and Complexity — Spring 2018

Solutions for Tutorial Exercise 4: Dynamic Programming

1. **Bellman-Ford Examples.** This exercise illustrates the execution of two different versions of the Bellman-Ford algorithm, in cases with and without negative t-connected cycles.

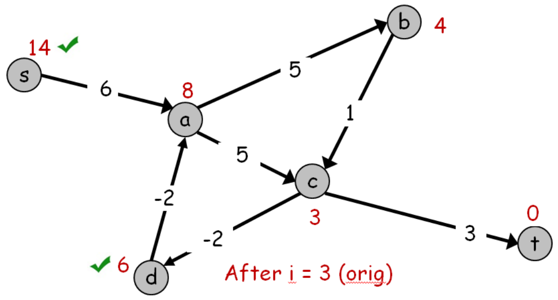
   1a) First, consider the algorithm described on slide 6 of the Dynamic Programming in Graphs lecture notes. Trace the execution of this algorithm on the directed graph $G = (V, E, w)$ shown below. Suppose the iterations through the vertices V occur in "alphabetic" order (i.e., in the sequence $(a, b, \dots, s, t)$). Show the value $M(2, v)$ after the **first two** iterations through all the vertices.

   Solution 1a) The results are below, the checkmarks indicate vertices that changed on iteration $i = 2$.



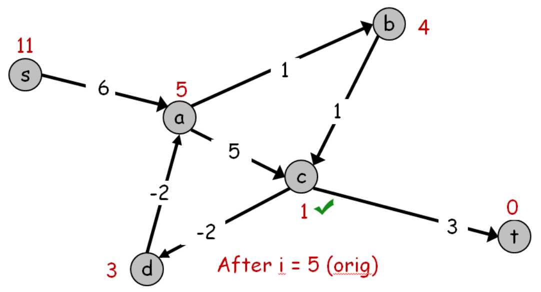
   1b) In the lectures we noted that this algorithm could terminate when it first detects that either: a) $M(i, v) = M(i - 1, v)$ for all vertices v, or b) it completes the iteration for $i = |V|$ (i.e., one more iteration than on slide 6, since this allows us to detect cycles). After what iteration number, i, does this algorithm terminate when given the graph in part (1a) as input?

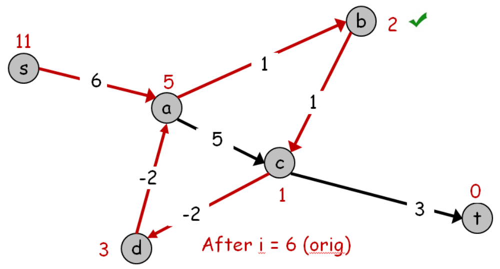
   Solution 1b) The results for $i = 3$ are below. The algorithm terminates after $i = 4$ completes with no change in $M(i, v)$.



   1c) Next suppose we change the weight of the edge (a, b) from 5 to 1 (which is shown below). Show $M(i, v)$ on the figure below after the fifth and sixth iteration. Note that there is at least one vertex v at which $M(i, v) < M(i - 1, v)$ for all $i \le 6$. Draw all the successor edges at the end of the sixth iteration. Is this "successor" graph acyclic?

   Solution 1c) Red edges indicate successors when $i = 6$, and the graph is cyclic.



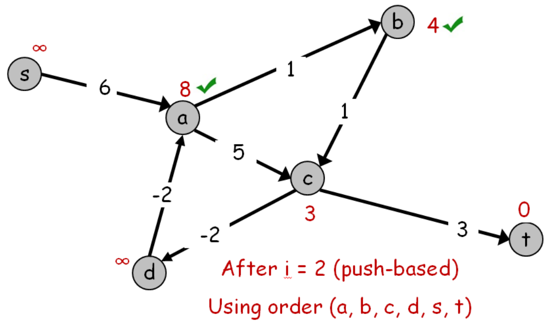


1d) Consider the push-based implementation that was briefly described in the lecture notes (slide 9 of Dynamic Programming in Graphs). A detailed function definition is given below. We assume that the loops over the vertices are executed in the same order as before, namely in the order (a, b, c, d, s, t). (This order now matters, as we shall see.)

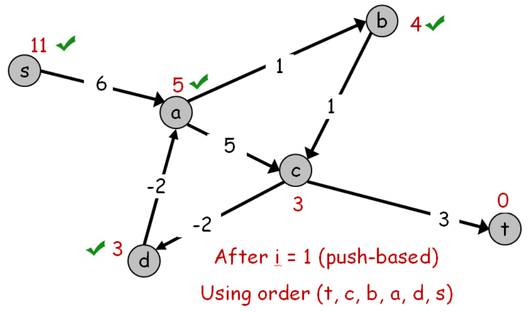
```
Push-Based-Shortest-Path(G, t)
   foreach node v ∈ V {
      M[v] ← ∞;   C_prev[v] ← false
      successor[v] ← ϕ
   }
   M[t] = 0; C_prev[t] = true
   for i = 1 to |V| {
      C_cur(v) = false for all v ∈ V
      foreach node w ∈ V with (C_prev[w] ∨ C_cur[w]) true {
         foreach node v such that (v, w) ∈ E {
            if (M[v] > M[w] + c_vw) {
               M[v] ← M[w] + c_vw
               C_cur(v) ← true
               successor[v] ← w
            }
         }
      }
      If C_cur[w] is false for all w ∈ V, break.
      C_prev[w] ← C_cur[w] for all w ∈ V
   }
   return M, successor, any(C_cur)
```

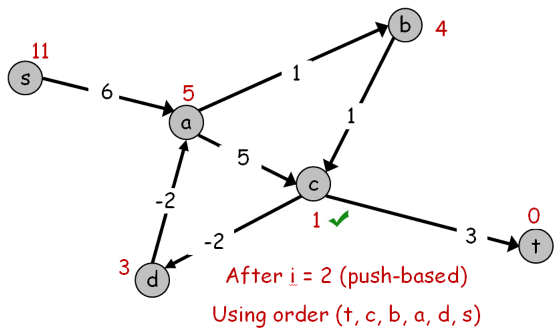
After just two iterations through the loop over i, show the $M(v)$ computed by this push-based implementation applied to the graph in part (1c). (For your convenience we have redrawn this graph below.) Compare this with the result $M(2, v)$ of two itertions of the original implementation (now applied to this new graph). You can indicate $M(2, v)$ on the second copy of the graph below. After each iteration, is it true that $M(v) = OPT(i, v)$? How about $M(v) \leq OPT(i, v)$?

Solution 1d) The values are the same for both the push based implementation and the original, at least for the vertex ordering $(a, b, \ldots, s, t)$.



However, if the ordering of the vertices is changed to (t, c, b, a, d, s) (i.e., a breadth-first search, starting at t in the reversed graph, might find the vertices in this order), then





Clearly the results of the push-based iterations depend on the order the vertices are traversed in. The claim is that, after iteration i, $M(v) \leq OPT(i, v)$ and $M(v)$ equals the length of some path from v to t.

2. **Longest Increasing Subsequence.** Given an array of n integers $[x(1), \ldots, x(n)]$ define an **increasing subsequence** of x, to be a subsequence $x(j(1)) < x(j(2)) < \ldots < x(j(p))$ where the indicies $j(i)$ are also strictly increasing with i, that is, $1 \leq j(1) < j(2) < \ldots < j(p) \leq n$. We are interested in a **longest increasing subsequence** of x, i.e., one of the ones with the maximum length p. Define $LLIS(x)$ to be the **length** of a longest increasing subsequence of x (there may be more than one increasing subsequence with this length).

   For example, given the array $x = [22, 5, 8, -3, 10, 1]$ the longest increasing subsequence is $[5, 8, 10]$ which has length is 3. Therefore, for this example, $LLIS(x) = 3$.

   In addition, define the array $[q(1), \ldots, q(n)]$ to have elements $q(k)$ which equal the length of the longest increasing subsequence of x **ending with the value** $x(k)$. In particular, such a subsequence only uses values from the first k elements of x, namely $[x(1), \ldots, x(k)]$ and must end with $x(k)$ itself.

   For the example array x above, $[q(1), \ldots, q(6)] = [1, 1, 2, 1, 3, 2]$.

   2a) Give an equation (a recurrence relation) which expresses $q(k)$ for $1 < k \leq n$ in terms of the values of $q(j)$ for various $j < k$ and possibly other simple expressions involving elements of the array x. Explain.

   Solution 2a)

   $$q(k) \;=\; 1 + \max\left[\{0\} \cup \{q(j) \mid 1 \leq j < k \text{ and } x(j) < x(k)\}\right]$$

   Explanation: Two cases for the longest increasing subsequence ending at x(k):
   Case 1: The subsequence is length 1, consisting of x(k) alone. This case is dealt with
   by including the term {0} in the max above (the second set could be empty).
   Case 2: The longest subsequence ending at $x(k)$ has length larger than 1.

Let the second last item in the subsequence be x(j) for some $j < k$.
For the subsequence to be increasing, we require that $x(j) < x(k)$.
Since we are looking for the longest subsequence, we must use the longest subsequence ending at $x(j)$, which has length $q(j)$.
With $x(k)$ appended, the length of the resulting subsequence is $1 + q(j)$.
Therefore $q(k) \geq 1 + q(j)$ for each such carefully chosen j (if any).

Finally, since we are looking for the longest increasing subsequence we want to maximize over all possible choices from these two cases.

2b) Express $LLIS(x)$ in terms of the array q. Explain.

Solution 2b)
$LLIS(x) = \max\{q(k) \mid 1 \leq k \leq n\}$.
Explanation: The longest increasing subsequence has to end at some $x(k)$, in which case it has length $q(k)$. So we should choose the max $q(k)$.

2c) Given the arrays x and q defined above, provide pseudo-code for an algorithm which extracts a longest increasing subsequence. (In this case precisely worded English sentences describing simple steps are allowed in your pseudo-code.) In the previous notation, the algorithm should return $[x(j(1)), x(j(2)), \ldots, x(j(p))]$ for the maximum p. The $j(k)$'s themselves do not need to be returned, just the corresponding x values (in order).

Solution 2c)

```
[s] = LIS(x, q, n)
    Find an index k such that q(k) is the maximum of [$q(1), \ldots, q(n)$].
    s = [ ]                // s is initially an empty list.
    while q(k) >= 1
        s = [x(k) s]       // Prepend s with x(k).
        if q(k) == 1
            return s
        Find (any) j in the range $1 \leq j < k$ such that q(j) = q(k) - 1 and $x(j) < x(k)$.
            Raise an exception if such a j does not exist.
        k = j
    end
```

2d) Briefly explain why your algorithm is correct.

Solution 2d) Explanation:

As in part (b) above, the longest inceasing subsequence ends at x(k), for a k where q(k) is maximal.
For $q(k) > 1$, the second-last item x(j) must have q(j) = q(k) - 1, since the subsequence ending at $x(j)$ is just one shorter.
This second-last item x(j) must also satisfy $x(j) < x(k)$ since the subsequence is increasing.
The algorithm finds exactly such a j and iterates.
Each iteration decreases q(k) by one, until the q(k) = 1.
For q(k) = 1 we know the first element of the sub-sequence is x(k).

3. **Pseudo-Polynomial Time.** In class we briefly discussed the fact that the dynamic programming solution to the Knapsack problem is not a polynomial time solution. We revisit that here.

The issue is that a polynomial time algorithm must formally have a runtime that is bounded by a polynomial in the size of the input. What's a simple measure for the size of the input for an instance of the Knapsack problem? The input, say I, includes a list of n pairs of values and weights, say $\{(v_i, w_i)\}_{i=1}^{n}$, along with another integer, W, which is the capacity of the backpack. If all these integers were specified in B bits, then the input size for n, W and all pairs, would be $(2n+2)B$ bits. (This will suffice for defining the input size. Fortunately, due to approximation properties of big-Oh, we will not need a more accurate estimate of the number of bits needed to specify the input. But, for the insatiably curious, see Information Theory.)

We say that an algorithm runs in polynomial time iff there is some constant $q > 0$ such that, for all input I, the runtime $T(I)$ satisfies

$$T(I) \in O(|I|^q). \tag{1}$$

That is, as the size of the input $|I|$ goes to infinity, the runtime must remain bounded by a constant times this monomial $|I|^q$.

In class we showed the dynamic progamming solution to Knapsack runs in time $\Theta(nW)$. Show that there is no q for which (1) is true.
Hint 1: You can treat n as fixed and consider only the effect of increasing B.
Hint 2: Look up l'Hospital's rule.

Solution Q3. Reduce the problem to showing $2^B \notin O(|B|^q)$ for any constant $q > 0$. Use contradiction. It follows from $2^B \in O(|B|^q)$ and the definition of big-Oh that there must be constants c and B_0 such that

$$2^B \leq c|B|^q \text{ for all } B > B_0 \ . \tag{2}$$

WLOG we can take $q = ceil(q)$ and $B_0 > 1$. Then (2) is true iff

$$\frac{2^B}{|B|^q} \leq c \text{ for all } B > B_0 \ . \tag{3}$$

Applying l'Hospital's rule q times, shows the limit of the LHS in (3) is

$$\lim_{B \to \infty} \frac{(log(2))^q 2^B}{q!} = \infty. \tag{4}$$

Therefore the LHS in (3) is unbounded as $B \to \infty$, and so there can be no such constant c. ■

4. **Checkerboard Pebbling**. Suppose you have something similar to a checkerboard, with every square on the board coloured black or red, and with these two colours alternating along every row and column. (As a result, the colours are the same on diagonals.) Moreover, this board has an integer valued "score" listed on each square, say $S(i,j)$, which can be both positive, negative or zero. For this problem, we assume the board has only four rows, and $n > 0$ columns, and you have $2n$ pebbles.

You can select a square, and gain the score for that square, by placing a pebble on it. The constaints on placing your pebbles is that you cannot place two pebbles on two squares in the same row and neighbouring columns (say, row i and both columns j and $j+1$), nor on two squares in the same column and neighbouring rows (say, column j and both rows i and $i+1$). (Note that placing pebbles along any diagonal in the board is allowed, as is placing two pebbles on just the black squares in a column, as are many other configurations.) Pebble placements that satisfy these constraints are said to be **feasible**.

The problem is to place some or all of your pebbles on the board in a feasible way such that you maximize the sum of the scores $S(i,j)$ for the squares which have a pebble on them. Moreover, we wish to use a dynamic programming approach to find a solution of this problem.

4a) Consider the following choice of subproblems. For each $1 \leq i \leq 4$, $1 \leq j \leq n$ and $u = 0, 1$, define

$OPT(i,j,u) \equiv$ Maximum score for feasibly placing pebbles in the first $j-1$ columns; and in the first $i-1$ rows (if any) of column j; and, if $u = 1$, then placing a pebble on the square (i,j), otherwise leaving that square empty. (5)

Assume that $OPT(i,0,0) = 0$ and $OPT(i,0,1) = -\infty$ for all i. You can assume the score is $-\infty$ for any illegal placement.

Either: A) derive a recurrence in terms of just these sub-problems and the scores $S(i,j)$; or, B) if you find that additional subproblems are needed then clearly explain why and explain what they might be. In case (B) you do not need to derive the recurrence relation for these new subproblems.

Solution 4a) For the problem $OPT(i,j,1)$ for $j > 1$, we need to know if the square to the left, i.e., at $(i, j-1)$ has a pebble or not and, if not, what the resulting optimum value is for **both** squares $(i, j-1)$ and $(i-1, j)$ being empty. This would need to be a new sub-problem. Moreover, we would need still more state history in order to to construct a solution based on $OPT(i, j-1, u)$ and the optimum way to fill in the rest of column $j-1$ plus the top part of column j, ending up with or without a pebble on square $(i-1, j)$. This approach is unravelling, it is time to reconsider the definition of the sub-problems.

4b) An alternative approach to this problem is to first define all feasible configurations for one column, and then do dynamic programming over these column configurations.

Specifically, suppose we denote the location of pebbles on one column using a four bit vector, $\vec{b} = (b_1, b_2, b_3, b_4)$, where $b_i = 1$ iff there is a pebble on row i in this column. Then there are eight feasible binary patterns (i.e., which don't have two 1's as neighbours). We can denote these bit patters as $\vec{b}(k)$ for $k = 1, 2, \ldots 8$, and we will use the notation $b_i(k)$ to denote the i^{th} bit of $\vec{b}(k)$. We assume that the first such pattern is $\vec{b}(1) = (0,0,0,0)$.

What is the recurrence relation for the following choice of subproblems? Define

$$OPT(k,j) \equiv \text{ Maximum score for placing pebbles in the first } j-1 \text{ columns, and placing pebbles in column } j \text{ according to the pattern } \vec{b}(k). \tag{6}$$

Here you can assume $OPT(1,0) = 0$, $OPT(k,0) = -\infty$ for $1 < k \leq 8$, and the score for any infeasible arrangement is $-\infty$.

If it is not possible to define such a recurrence relation, explain why. Otherwise, explain why your recurrence relation is correct and state the runtime order for solving this problem.

Solution 4b) Define the bipartite graph (W, B) with eight vertices on each side, say u_i, v_i, for $i = 1, 2 \ldots 8$. (So $W = \{u_i \mid 1 \leq i \leq 8\} \cup \{v_i \mid 1 \leq i \leq 8\}$. Here u_i and v_i represent pattern $\vec{b}(i)$ at adjacent columns of the checkerboard. We define an edge (u_i, v_j) to be in B iff it is feasible to place pebbles in patterns $\vec{b}(i)$ and $\vec{b}(j)$ in adjacent columns. Also define $\vec{S}(j) = (S(1,j), \ldots, S(4,j))^T$ to be the column vector of scores for column j. (In which case placing pebbles with pattern $\vec{b}(k)$ on column j would produce the score given by the inner-product $\vec{b}^T(k)\vec{S}(j)$. Then we have

$$OPT(k,j) = \begin{cases} 0 & \text{if } k = 1 \text{ and } j = 0 \ , \\ -\infty & \text{if } k > 1 \text{ and } j = 0 \ , \\ \max\{\vec{b}^T(k)\vec{S}(j) + OPT(i, j-1) \mid \text{ for } 1 \leq i \leq 8 \text{ and } (u_i, v_k) \in B\} & \text{for } 1 \leq j \leq n. \end{cases} \tag{7}$$

This recurrence is correct since it exhaustively enumerates all possibilities at each stage and computes the corresponding optimum values. The runtime order for solving this recurrence relation and then finding an optimal pebble placement is $O(n)$.