# Colorado CSCI 5454: Algorithms
# Homework 3

Instructor: Bo Waggoner
Due: September 19, 2019 at 11:59pm
Turn in electronically via Gradescope.

## Problem 1 (12 points)

In the *Longest Common Subsequence* problem, the input consists of two strings, $x$ and $y$. The output should be the length of the longest sequence of characters that is a subsequence of both $x$ and $y$.

Recall that a subsequence does not have to be *consecutive*. For example, if $x = ALGORITHM$ and $y = ANARCHISM$, then $ARHM$ is a subsequence of both, and so is $ARIM$.

(This is used by the `diff` utility and version control systems like `git` to compare text files line-by-line. In these cases, each line of a document is treated as a "character" and we want to find the longest common subsequence of lines.)

**Part a (10 points)** Give a dynamic programming algorithm to solve the longest common subsequence problem. Clearly identify the definition of a subproblem *(2 points)* and the recurrence (i.e. how to solve a subproblem given previous subproblem solutions, *2 points*), and justify correctness of your recurrence *(2 points)*. Then give the full algorithm *(2 points)*, and briefly argue running time and space usage *(2 points)*.

**Solution.** The subproblem $E[i, j]$ is the length of the common subsequence in string $x$ up to index $i$, and string $y$ up to index $j$.

If $x[i] \neq y[j]$, then any common subsequence will still be present either if we delete the $i$th character from $x$; or else if we delete the $j$th character from $y$. So in the case $E[i, j] = \max\{E[i-1, j], E[i, j-1]\}$. On the other hand if $x[i] = y[j]$, then the longest common subsequence is achieved by including this character at the end of the optimal longest common subsequence up to $i-1$ and $j-1$, so $E[i, j] = E[i-1, j-1] + 1$. (Note, to be careful, that in this case we could still try deleting each of the final characters, but we'd still get at most $E[i-1, j-1] + 1$ so this is optimal without loss of generality.) This proves that the recurrence is

$$E[i, j] = \begin{cases} \max\{E[i-1, j]\ ,\ E[i, j-1]\} & x[i] \neq y[j] \\ E[i-1, j-1] + 1 & x[i] = y[j] \end{cases}. \tag{1}$$

The full algorithm: Initialize the base cases, $E[0, j] = 0$ and $E[i, 0] = 0$ for all $i, j$ because there is no common subsequence with an empty string. Then for each $i = 1, \ldots, |x|$, for each $j = 1, \ldots, |y|$, set $E[i, j]$ according to the above recurrence. Finally, return $E[|x|, |y|]$.

The running time is $O(|x| \cdot |y|)$ because of the two for loops, and we just do a constant number of lookups and arithmetic operations inside the loops. Initialization and returning the answer take $O(|x|)$, $O(|y|)$, and $O(1)$ time, which does not increaes the big-O worst case runtime. Space usage here is $O(|x| \cdot |y|)$ because we fill the two-dimensional array.

**Part b (2 points)** Explain how to modify your algorithm to return the subsequence itself.

**Solution.** Create an array $F[i, j]$. In the inner loop, if we set $E[i, j] = E[i-1, j]$, then set $F[i, j]$ :=left. Otherwise, if we set $E[i, j] = E[i, j-1]$, then set $F[i, j]$ :=up. Otherwise, we must set $E[i, j] = E[i-1, j-1] + 1$, so set $F[i, j]$ =both.

At the end of the algorithm, we build the answer from right to left. Start with an empty list and start at $i = |x|, j = |y|$. Then read $F[i, j]$. If it says "left", subtract 1 from $i$. If it says "up", subtract 1 from $j$. If it says "both", subtract 1 from both and add the character $x[i]$ to the beginning of our list. (Note this equals the character $y[j]$. Repeat until $i = 0$ or $j = 0$, then stop and return the list.

# Problem 2 (10 points)

*(Max flow with vertex capacities.)*

Recall the *maximum s to t flow* problem:

We are given a directed graph $G = (V, E)$ and two vertices $s, t \in V$, and we must output a function $f : E \to \mathbb{R}_{\geq 0}$. We say $f$ is a *valid flow* if it satisfies *flow conservation* constraints and *edge capacity* constraints.

The flow constraints are:

$$\sum_{e \in \delta^{in}(v)} f(e) = \sum_{e \in \delta^{out}(v)} f(e) \qquad \forall v \in V \setminus \{s, t\} \tag{2}$$

where $\delta^{in}(v)$ is the set of incoming edges of $v$ and $\delta^{out}(v)$ is the set of outgoing edges of $v$. This says the total flow into vertex $v$ must equal total flow out. (Notice the source $s$ and sink $t$ are exempt from this requirement.)

We are also given an *edge capacity function* $c : E \to \mathbb{R}_{\geq 0}$ and the edge capacity constraints are:

$$f(e) \leq c(e) \qquad \forall e \in E. \tag{3}$$

This says the flow through an edge is at most its capacity.

The *value* of a flow $f$ is then total flow going into $t$, which is $\sum_{e \in \delta^{in}(t)} f(e)$. We cover in class how to find the maximum-value $s$ to $t$ flow in polynomial time. Now for this problem, suppose we are also given a *vertex capacity* function $w : V \to \mathbb{R}_{>0}$. Now we say a flow is valid if it satisfies 2, 3, and

$$\sum_{e \in \delta^{in}(v)} f(e) \leq w(v) \qquad \forall v \in V \setminus \{s, t\}. \tag{4}$$

These are called the *vertex capacity* constraints.

*Example: We wish to route traffic over the road network from $s$ to $t$ where road segments are edges and intersections are vertices. Both road segments and intersections can only handle a certain rate of traffic flow, so we have both edge and vertex capacity constraints.*

**Part a (2 points)** Suppose that VertCapAlg is an algorithm which solves the maximum $s$ to $t$ flow problem when edge and vertex capacities are specified. Describe how VertCapAlg can be used to solve the original maximum $s$ to $t$ flow problem when only edge capacities are specified.

**Solution.** We are given an instance of the original maximum $s$ to $t$ flow problem. We take this instance and add a vertex capacity function $w(v) = \infty$ for all vertices. Now VertCapAlg will return the correct answer, because the vertex capacity constraints will never be binding or influence the algorithm (in other words the optimal flow solution is the same).

**Part b (4 points)** Give an efficient algorithm to solve the maximum $s$ to $t$ flow problem when edge and vertex capacities are specified. Argue correctness and running time. *(Hint: You should reduce to a problem we already know how to solve efficiently.)*

**Solution.** Given an instance with graph $G = (V, E)$ and capacities $c, w$, we create a new graph $G' = (V', E')$ with capacities $c'$. For each vertex $v \in V$, we create two vertices $v_1, v_2 \in V'$. (Think of this as the entry-point of the old vertex and the exit-point.) For each edge $e = (u, v) \in E$, we create an edge $e' = (u_2, v_1) \in E'$ from the exit of $u$ to the entrance of $v$. We set $c'(e') = c(e)$, i.e. the same capacity as the old edge.

For each vertex $v \in V$, we create an edge $e_v = (v_1, v_2) \in E'$ and set $c(e_v) = w(v)$. In other words, the flow "inside" the vertex (from its entrance to its exit) must be at most $w(v)$.

Now we run a max-flow algorithm on this graph. Then we translate it back to a flow on the original graph via the edges of the form $(u_2, v_1)$.

We have the same big-O running time as max flow on the original graph would have. First, the time to create the new graph is linear in the size of the original graph. Second, the new graph is only at most twice as large in vertices and edges, because the number of vertices has only doubled and the number of edges has only gone up by $|V|$ (we have been assuming $|E| \geq |V|$, so this is at most doubling). So we run max flow on a graph at most twice as big as the original.

Correctness sketch: each of our edges $(u_2, v_1)$ corresponds exactly to the original graph, and all flow that passes through a vertex $v$ of the original graph must go along the edge $(v_1, v_2)$ in the new graph. So satisfying the original constraints is equivalent to satisfying our constructed edge constraints.

**Part c (2 points)** Now suppose that we have a variant of the max $s$ to $t$ flow problem where *only* vertex capacities $w : V \to \mathbb{R}_{>0}$ are given. Now, each edge can support any amount of flow. Describe how VertCapAlg can be used to solve this problem.

**Solution.** Given an instance of the problem with only vertex capacities, we add edge capacities and set them all to infinity. Give this instance to VertCapAlg, and the solution will be a solution to the original problem, because only the vertex constraints will bind.

**Part d (2 points)** Given a directed graph $G$ and vertices $s, t$, an *$s$ to $t$ vertex cut* is a set of vertices such that, if they are removed from the graph, there is no path from $s$ to $t$. If we are also given a vertex weight function $w : V \to \mathbb{R}_{>0}$, a *min $s$ to $t$ vertex cut* is a vertex cut $S$ of minimum total weight, i.e. minimizing $\sum_{v \in S} w(v)$.

Describe an algorithm for finding a min $s$ to $t$ vertex cut. You do not need to argue correctness and efficiency (but it should be correct and efficient).

*Hint: recall the max-flow min-cut theorem and your solution to the previous parts.*

**Solution.** Given the vertex weights, treat them as capacities and use the previous subpart to solve the min $s$ to $t$ flow problem with only vertex capacities. The reduction will give a max flow through edges only, and the binding sets of edges will be the "internal" edges of the form $(v_1, v_2)$ described above. The min cut of these edges, which we get from the max-flow min-cut theorem, gives the minimum vertex cut.

# Problem 3 (14 points)

*(Programming and theory assignment.)*

You are a cashier whose goal is to give customers their correct change while using as few total coins as possible. The input is the set of available coin denominations as integers $x_1, \ldots, x_n$; and the amount of change that is due, an integer $W$. The output is the fewest number of total coins that add up to $W$.

For example, the American coin denominations are $x_1 = 1$ (the penny), $x_2 = 5$ (the nickel), $x_3 = 10$ (the dime), $x_4 = 25$ (the quarter), $x_5 = 50$ (the 50-cent piece), and $x_6 = 1$ (the dollar coin). If $W = 17$, the optimal solution is a dime, a nickel, and two pennies for a total of 4 coins.

You may assume that $x_1 = 1$ and that $x_i < x_{i+1}$ for each $i = 1, \ldots, n-1$.

**Part a (4 points)** The *greedy* algorithm for this problem is: give as many as possible of the largest coin denomination without going over $W$; then as many of the the next-largest as possible, and so on.

Implement the greedy algorithm in a programming language of your choice and attach the source code. Run it on the following inputs and report the number of coins required as well as a list of how many of each coin is used.

- The U.S.A. coin denominations with $W = 42$.
- The U.S.A. coin denominations with $W = 1728$.
- Coin denominations $1, 8, 20, 30, 80, 200$ and $W = 42$.

- The previous denominations and $W = 1728$.

**Solution.**

- 5 coins (optionally, the number of each denomination: $[2, 1, 1, 1, 0, 0]$)
- 21 coins $[3, 0, 0, 1, 0, 17]$
- 6 coins $[4, 1, 0, 1, 0, 0]$
- 14 coins $[2, 2, 0, 1, 1, 8]$

**Part b (2 points)** Give an example showing that the greedy algorithm is not always optimal.

**Solution.** The third example above (denominations $1, 8, 20, 30$ and $W = 42$). Greedy gives one 30 coin, one 8 coin, and four 1 coins (six total). The optimal solution is two 20 coins and two 1 coins (four total).

**Part c (4 points)** Give a dynamic programming algorithm to solve this problem optimally. Argue correctness, running time, and space use.

**Solution.** Create an array $K[w]$ representing how many coins are required to optimally make change for input $w$.
Create an array $N[w, j]$ representing, when we make optimal change for $w$, how many coins of denomination $j$ we use.
Set $K[0] = 0$, the base case.
Set $N[0, j] = 0$ for all $j$.
For $w = 1, \ldots, W$, set

$$K[w] = \min_{j: x_j \leq w} 1 + K[w - x_j].$$

If $j$ achieves the min, then we set $N[w, j] = N[w - x_j, j] + 1$ and we set $N[w, j'] = N[w - x_j, j']$ for all $j'$. (In other words, use the same number of each coin, plus one more of denomination $j$.) Return $K[W]$ and $N[W, j]$ for all $j$.

Correctness: the base case $K[0] = 0$ is immediately correct. Now suppose $K[0], K[1], \ldots, K[w-1]$ are all correct. The optimal solution $K[w]$ must involve adding some coin to a previous solution (because we need to add up to total change $i$). For each coin $j$ we add, consider the optimal solution given that we must include that coin. It uses one of coin $j$ plus $K[w - x_j]$ other coins. We take the minimum over all possible coins we might add, so we have the optimal solution for $w$. We also correctly record the number of total coins used. This proves correctness of the recurrence, thus the whole algorithm (since we just return the solution to the subproblem of size $W$).

The running time is $O(W \cdot n)$. Recall $n$ is the number of different coin sizes. There is an outer loop of $W$ iterations. For each $w$, we first do up to $n$ iterations of a loop to set $K$, then up to $n$ more iterations of a loop to set $N$. All operations inside the loops are constant time, and outside operations just include initialization. The space use is $O(W)$ for $K$, and $O(W \cdot n)$ for the array $N$.

**Part d (4 points)** Implement your dynamic programming algorithm in a language of your choice (attach source code) and run it on the examples from part (a); report the results.

**Solution.**

- 5 coins (optionally, number of each denomination: $[2, 1, 1, 1, 0, 0]$)
- 21 coins $[3, 0, 0, 1, 0, 17]$
- 4 coins $[2, 0, 2, 0, 0, 0]$
- 12 coins $[0, 1, 2, 0, 1, 8]$