15-451/651 Algorithms, Spring 2019
Recitation #5 Worksheet

Recap of this week's lectures:

- Dynamic programming: top-down and bottom-up
- Examples: Knapsack, Independent set in trees, LCS, etc.
- Single-source shortest path algorithms: Dijkstra and Bellman-Ford
- All-pairs shortest path algorithms: Floyd-Warshall and Matrix-Mult
- Subset DP for TSP

---

1. **Off-Line Stock Market Problem** You're given a sequence of stock prices $[p_1, p_2, \ldots, p_n]$. You want to find the maximum profit that you could have made on the stock in hindsight. In other words, you want to find i and j with $1 \leq i \leq j \leq n$ such that $p_j - p_i$ is maximal. Your algorithm should run in $O(n)$ time.

   Solution: Scan the sequence from first to last. After processing p_i keep two things: (1) m_i the minimum of $p_1, \ldots, p_i$, and (2) g_i the maximum profit achievable so far. It's easy to update these when processing the next stock price p_{i+1}.

   $$m_{i+1} = \min(m_i, p_{i+1})$$

   $$g_{i+1} = \max(g_i, p_{i+1} - m_{i+1})$$

   The final answer is m_n. (For completeness, note that $m_0 = \infty$ and $g_0 = -\infty$.)

2. **Longest Increasing Subseqeuence:** Given an array A of n integers like `[7 2 5 3 4 6 9]`, find the longest subsequence that's in increasing order (in this case, it would be `2 3 4 6 9`). Give a dynamic-programming algorithm that runs in time $O(n^2)$ to solve this problem.

   (a) To keep things simple, first let's say you just need to output the *length* of the longest-increasing subsequence. E.g., in the above case, the length is 5.

   Solution: Suppose that for each $i' < i$ you have computed the length of the LIS of $A_{0\ldots i'}$ that ends with $A[i']$. How would you use this to solve the corresponding problem for i? $L[i] = \max\{L[i'] + 1 : i' < i, A[i'] < A[i]\}$, or $L[i] = 1$ if there are no such i'.

   (b) Now extend your solution to actually find the LIS.

   Solution: One approach is when computing the max above, to also have a separate array that stores the argmax, that is, the index i' such that $L[i] = L[i'] + 1$. One can then read off the sequence by going backwards from the end.

3. **Making Change:** You are given denominations $v_1, v_2, \ldots, v_n$ (all integers) of the various kinds of currency you have. (Say $v_1 = 1$, so you can make change for any integer amount $C \geq 1$.) Given C, give a dynamic programming solution which makes change for C with the fewest bills possible.

   (Again, as a first stab, compute the number of bills required, and then extend the solution to output the number of bills of each denomination needed.)

   Solution: Create an array B where $B[C']$ represents the fewest bills needed to make change for C'. We can fill this in using the formula $B[C'] = \min\{B[C' - v_i] + 1 : v_i \leq C'\}$, where we begin with $B[0] = 0$ and then work upward from $C' = 1$ to C. The total time taken is $O(Cn)$.

4. **Making Change (Part II):** Now suppose you have only one bill of each denomination i. Given C, give a dynamic programming solution which makes change for C using the fewest bills, using no more than one bill of each denomination i (or says this is not possible).

   Solution: One approach is to create a 2-dimensional array B where $B[C', i]$ represents the fewest bills needed to make change for C' using denominations $1, 2, \ldots, i$ only (or infinity if it is not possible). Base case $B[0, 0] = 0$ and $B[C', 0] = \infty$ for $C' > 0$. For general values of i we have $B[C', i] = \min(B[C', i-1], B[C' - v_i, i-1] + 1)$ if $C' - v_i \geq 0$ or else $B[C', i] = B[C', i-1]$ if $C' - v_i < 0$.

5. **Making Change (Part III):** Can you solve the problem if you have ℓ_i bills of denomination i?

   Solution: We can just modify the formula for B above to:

$$B[C', i] = \min\{B[C' - jv_i, i-1] + j : 0 \leq j \leq \ell_i, C' - jv_i \geq 0\}.$$

6. **Balanced Partition.** You have a set of n integers each in the range $0, \ldots, K$. In time $O(n^2K)$, partition these integers into two subsets such that you minimize $|S_1 - S_2|$, where S_1 and S_2 denote the sums of the elements in each of the two subsets.

   Solution: Let S be the sum of all the integers. Then $S \leq nK$. To minimize $|S_1 - S_2|$ it suffices to find a set A_1 whose numbers sum to $S_1 \leq \lfloor S/2 \rfloor$, that is as close to $S/2$ as possible. And this can be done by a dynamic program like for knapsack, in time $O(nS) = O(n^2K)$.

7. **Bottleneck Paths.** Given a directed graph G, suppose each edge has a non-negative capacity c_e. Given a directed path from s to t, the bottleneck edge of this path is the min-capacity edge on it. The s-t bottleneck path asks for such a path whose bottleneck edge capacity is as large as possible.

   Show how to modify Dijkstra's algorithm to solve this problem in $O(m \log n)$ time (or $O(m + n \log n)$ time using Fibonacci heaps).

   Solution: The basic idea is as follows: Start from the set $X = s$, and find the vertex adjacent to X that has the highest bottelneck. Add that vertex to X.

   Claim (Dijktra's Principle for bottlenecks): For any partition of V into X with $s \in X$ and $Y = V \backslash X$, let

$$p(v) = \max_{x \in X}(\min(bottle(s, x), cap(x, v)))$$

   Then $\max_{y \in Y} p(y) = \max_{y \in Y} bottle(s, y)$

   Proof. Let a vertex $v \in Y$ be such that $bottle(s, v) = max_{y \in Y} bottle(s, y)$, and consider the path from s to v. Since the path starts at $s \in X$, there must be some $u \in Y$ such that the path goes from a vertex in X to u. However, the bottleneck is the minimum of all capacities along the path, so $bottle(s, u) \geq bottle(s, v)$, which means that $\max_{y \in Y} p(y) \geq bottle(s, u) \geq bottle(s, v) \geq max_{y \in Y} bottle(s, y)$, which completes the proof.

   So now we do the same as Dijkstra's Algorithm, keeping a max heap of values adjacent to our current set, and adding the max to our set. To update the heap we insert the minimum of the bottleneck and the capacity to the new vertex.

8. **Johnson's Algorithm.** We did not get a chance to discuss Johnson's algorithm for APSP, we do that now. (Details in notes, Section 4.3 of Lecture 8).

   (a) If the edge-weights are non-negative, running Dijkstra from each node takes $n \cdot O(m + n \log n)$ time. Much faster than F-W if the graph is sparse, i.e., $m \ll n^2$.

   Solution: Seems good

   (b) Suppose we assign an "offset" Φ_v to each vertex v, and define the offset edge-length of edge (u, v) to be $\ell'_{uv} := \Phi_u + \ell_{uv} - \Phi_v$. Show that a s-t path is a shortest path with respect to lengths ℓ if and only if it is a shortest path with respect to lengths ℓ'.

   Solution: Consider any path from s to t, and lets say that it goes through a vertex v. Then there is an edge into v (that contributes a Φ_v), and an edge out of v, which contributes a $-\Phi_v$. So when we sum all of the edges on the path, any internal vertices cancel out, leaving us with $\Sigma \ell_{uv} + \Phi_s - \Phi_t$. This holds for any path, and since Φ_s and Φ_t are constant, the shortest path with the new edges weights is the same.

(c) Call an offset "feasible" if $\ell'_{u,v} \geq 0$ for all edges, even if ℓ could have been negative. Suppose there is a vertex x that has finite distance to every other vertex. Show that $\Phi_v := dist(x, v)$ is a feasible offset.

Solution: We want to show that $\ell'_{uv} \geq 0$, or equivalently, that $\ell_{uv} \geq \Phi_v - \Phi_u$. We can replace the potentials with the distances to x, giving us the following: $\ell_{uv} \geq dist(x, v) - dist(x, u)$ however the distance from x to v is upperbounded by the distance from x to u plus the edge length from x to v, because this is a valid path from x to v, which tells us that $dist(x, v) \leq dist(x, u) + \ell_{uv}$. Rearranging finishes the problem.

Note: if there is no x that has finite distance to all other points, you can just add a new vertex x and add edges from it to all other nodes, give these new edges arbitrary lengths (say even zero). This does not create negative-cycles (why?) and hence now you can do Bellman-Ford as above.

(d) Infer that you can compute APSP by running Bellman-Ford once (to compute the feasible offset) and then n Dijktras.

Solution:

i. Create a dummy node and connect it to every node in the graph with length 0.
ii. Run Bellman-Ford from the dummy node and get distances to every vertex in the graph. If you see a negative cycle stop.
iii. Using the distances as a potential modify all of the edges in the graph and run Dijkstras from every vertex.

9. **Coloring Dynamically.** Given a graph $G = (V, E)$, a (proper) k-coloring is a coloring of the vertices of the graph using k colors, so that the endpoints of each edge get distinct colors. An equivalent definition is that the vertices having any single color form an *independent set*.

This problem is NP-hard for $k \geq 3$, so we explore fast exponential-time algorithms.

(a) Show that the problem of 2-coloring a graph can be solved in linear time.

Solution: A graph is 2-colorable if and only if it is bipartite. That can be checked using DFS/BFS.

(b) The naive algorithm to k-color a graph takes k^n time. Give an algorithm that runs in time $O(k3^n)$. [A simpler bound is $O(k4^n)$.]

Solution: Let $G[X]$ be the "induced" graph obtained by just keeping the vertices in X, and the edges that go between them. Let $T(X, \ell) = 1$ when the graph $G[X]$ can be colored using at most ℓ colors, and $T(X, \ell) = 0$ otherwise.

Clearly, $T(\emptyset, 0) = 1$, and $T(S, 0) = 0$ for all non-empty S. Moreover,

$$T(X, \ell) = 1$$
$$\iff$$
there is some independent set $Y \subseteq X$, such that $T(X \setminus Y, \ell - 1) = 1$.

So to calculate $T(X, \ell)$, we can enumerate over all subsets Y, and use the values for $T(\cdot, \ell - 1)$. This takes time $2^{|X|}$.

Hence the total time to fill all entries $T(\cdot, \ell)$ is — there are 2^n choices of X and each takes at most 2^n time, giving 4^n. One can do better: the total time is, in fact,

$$\sum_{X \subseteq V} 2^{|X|} = \sum_{k=1}^{n} \binom{n}{k} 2^k = (1+2)^n.$$

Summing this for $\ell = 1, 2, \ldots, k$, gives us runtime $O(k3^n)$.

10. **Bin-Packing.** You are given a collection of n items, each item has size $s_i \in [0, 1]$. You have many bins, each of unit size, and you want to pack the n items into as few bins as possible. (Each bin can take a subset of items, whose total size is at most 1.)

    Show that you can solve this problem in time $O(n3^n)$. (Hint: subset DP.)

    Solution: Consider the following subproblems: $P(X, b)$ which represents the question "Can we pack the elements of X into b bins?".

    The bases case are $P(\emptyset, 0) = 1, P(X, 0) = 0$ for any non-empty X.

    We can solve this for some set X by doing the following: For every subset of X that can be fit into a single bin, Y, ask $P(X \backslash Y, b - 1)$. Any of these subproblems returning true means we return true.

    The runtime is basically the same as the previous problem, but instead of having k colors, we now can have no more than n bins, which gives us a runtime of $O(n3^n)$