4041 Homework 5

Fletcher Gornick

November 4, 2021

16.1

16.1-2

Suppose that instead of always selecting the first activity to finish, we instead select the last activity to start that is compatible with all previously selected activities. Describe how this approach is a greedy algorithm, and prove that it yields an optimal solution.

This approach is a greedy algorithm because if we're given a set A containing activities $a_1, a_2, \ldots, a_n$, and we take the element with the latest start time, we leave resources available for as many other activities as possible. So assuming our greedy choice $g = a_i$, when looking for our next optimal activity, we can look at the substructure $A_i = \{a_k \in A : f_k \leq s_i\}$, where s_i is the starting time of a_i and f_i is the finishing time. We know that this substructure contains more elements than any other substructure because our s_i upper bound of the set A_i is chosen to be the largest, thus, this is a greedy algorithm.

To show this approach yields an optimal solution, we can take the same approach as Theorem 16.1 in the book, and let A_k be a max subset of non-overlapping activities in S_k. Also let a_j be the activity in A_k with the latest start time, and a_m be the activity in S_k with the latest start time (the greedy choice). If $a_j = a_m$ then our work is done because it's shown that our greedy solution is part of the optimal solution. Now, if $a_j \neq a_m$, we can take $A'_k = A_k - \{a_j\} \cup \{a_m\}$. We know the activities in this new set are disjoint because A_k is disjoint, and $a_j \in A_k$ is the last activity to start in the subset, and since $s_m \geq s_j$, it must be the case that no elements in A'_k overlap with a_m. Therefore, $|A_k| = |A'_k|$, meaning our greedy choice also yields an optimal solution.

16.1-3

Not just any greedy approach to the activity-selection problem produces a maximum-size set of mutually compatible activities. Give an example to show that the approach of selecting the activity of least duration from among those that are compatible with previously selected activities does not work. Do the same for the approaches of always selecting the compatible activity that overlaps the fewest other remaining activities and always selecting the compatible remaining activity with the earliest start time.

For these examples, I'll represent each set of activities as an array of tuples, where the first element in each tuple represents the start time, and the second element represents the finish time. Array a contains all activities, and array m contains the maximum non-overlapping subset of a. Also, for the sake of simplicity, these times will just be represented as integers.

Greedy Choice: Least Duration - Take $a = \{(1, 6), (5, 7), (6, 12)\}$. a_2 is the activity with the least duration, which goes from 5 to 7, but it overlaps the other two activities. The optimal solution is actually $m = \{(1, 6), (6, 12)\}$.

Greedy Choice: Least Overlapping - For this problem, we must take a more complex approach. Take $a = \{(1, 5), (1, 10), (1, 11), (1, 12), (5, 9), (9, 13), (12, 14), (13, 17)\}$. This gives us one optimal solution $m = \{(1, 5), (5, 9), (9, 13), (13, 17)\}$. In this example, the activity that overlaps the least is a_8, or m_4 which is in our optimal solution, but the value that overlaps the second least is a_7 which can only yield a solution with 3 elements. But our optimal solution contains 3 elements that all overlap more than a_7, thus this approach does not always yield an optimal solution.

Greedy Choice: Earliest Start - Finally take $a = \{(1, 5), (2, 3), (4, 5)\}$. The optimal solution $m = \{(2, 3), (4, 5)\}$ does not contain a_1 which is the activity with the earliest start-time, so this approach also does not work.

16.2

16.2-4

Professor Gekko can travel m miles before running out of water. Give an efficient method by which he can determine which water stops he should make. Prove that your strategy yields an optimal solution, and give its running time.

Let us first assume that every water stop is less than or equal to m miles away from the next stop. The most obvious method for this algorithm would be to always pick the water stop furthest away from the current one that's still less than m miles. Now I will explain why this method yields an optimal solution.

First note that when we make our greedy choice, we choose a water station to stop at, so the only subproblem is the rest of the path from that water stop. Now all we need to show is that making the greedy choice doesn't make our solution any less optimal.

Let's assume we have an optimal solution X, which contains the set of points $\{x_1, x_2, \ldots, x_n\}$, each representing water stations that the professor stops at on his trip. Now say we want to swap out x_1 with our greedy choice g_1. If $x_1 = g_1$ then our work is done, and the greedy choice is already in our optimal solution. Otherwise, if we replace x_1 with g_1, then the distance between g_1 and x_2 is smaller than in our optimal solution, so it must also be the case that the distance between the two stops is less than m miles, making the new solution containing g_1 also valid, and since it contains the same number of elements as the optimal solution, it too is optimal.

Now we can't just swap any element in our optimal solution with a greedy choice. If you were to arbitrarily swap x_i with g_i where $1 \leq i \leq n$, then there may be a case where g_i is more than m miles from x_{i-1}, but if we start from element 1 and swap each element in the optimal solution with it's corresponding greedy choice, then we can be sure that the new solution containing only greedy choices up to element i works, and also contains the same number of elements as X.

16.2-5

Describe an efficient algorithm that, given a set $\{x_1, x_2, \ldots, x_n\}$ of points on the real line, determines the smallest set of unit-length closed intervals that contains all of the given points. Argue that your algorithm is correct.

Assuming the set is ordered, we want to maximize the possibility that our interval contains two points. We can do this by taking our first interval to start at x_1, because we know anything before x_1 isn't in our set anyway, so it'd be useless to make our left endpoint anything other than x_1.

We can call our set of unit-length intervals U, and our set of points X. If we have a subproblem X_i, which looks like $\{x_i, x_{i+1}, \ldots x_n\}$, we would take u_j to be the interval $[x_i, x_i + 1]$. Then we would recurse to the next subproblem X_k, where x_k is the next element in X_i such that $x_k - x_i > 1$.

So our algorithm starts at x_1 and makes an interval $u_1 = [x_1, x_1 + 1]$. Then we go jump to x_k to make our next interval in this same fashion. In this case x_k represents the first element in X that's more than 1 greater than x_1. We then follow this procedure until every element in X is within a unit-length interval in U.

We can show this greedy approach is correct by imagining an optimal solution that doesn't contain our greedy choice. Suppose $u_i = [a, b]$ is in our optimal solution, and take $x_j \in u_i$ to be the first element in the interval (or the only one if you can only fit one value in the interval). Since the leftmost value of the interval isn't x_j (because we assumed the optimal solution doesn't contain our greedy solution), we can swap the interval with one that DOES start at x_j, and since x_j is the first element in the interval that's in X, we won't lose any values by making this swap. And now our greedy interval extends further than the one in our optimal solution, so we know it must contain all the same values. And since we can swap any interval in an optimal solution with a greedy choice, our algorithm must work.

16.2-7

Suppose you are given two sets A and B, each containing n positive integers. You can choose to reorder each set however you like. After reordering, let a_i be the ith element of set A, and let b_i be the ith element of set B. You then receive a payoff of $\prod_{i=1}^{n} a_i{}^{b_i}$. Give an algorithm that will maximize your payoff. Prove that your algorithm maximizes the payoff, and state its running time.

If we take A and B to both be in sorted order, it seems that this would yield the maximum payoff. Let's say our algoritm sorts both sets from least to greatest, and since multiplication is commutative, sorting in descending order also yields the same payoff, but let's just assume we're sorting from least to greatest for our algorithm. We will show that this will maximize the payoff.

To show this algorithm maximizes payoff, let's take elements a_x, a_y, b_x, b_y from our sorted sets A and B, where $x < y$. Our claim is that $a_x^{b_x} a_y^{b_y} \geq a_x^{b_y} a_y^{b_x}$ for all $0 \leq x, y \leq n$. So we must show $a_x^{b_x} a_y^{b_y} - a_x^{b_y} a_y^{b_x} \geq 0$.

$$a_x^{b_x} a_y^{b_y} - a_x^{b_y} a_y^{b_x} \;=\; a_x^{b_x}(a_y^{b_y} - a_x^{b_y - b_x} a_y^{b_x}) \;=\; a_x^{b_x} a_y^{b_x}(a_y^{b_y - b_x} - a_x^{b_y - b_x}) \;\geq\; a_x^{b_x} a_y^{b_x}(a_y - a_x) \;\geq\; 0$$

This shows that sorting our sets into ascending (or descending) order yields a maximum payoff. Also, we were able to make these $\geq$ jumps in the above process because we know $a_x \leq a_y$ and $b_x \leq b_y$, since A and B are sorted. And since this property holds for any two multiplied elements, we know it must work for the whole chain, because we can just treat this new product as a single element for the next multiplication (inductive reasoning).

All this algorithm does is sort sets A and B. Since both sets contain n elements, our algorithm will run in $\Theta(n \lg n)$ time, assuming we're using HeapSort or MergeSort. If our algorithm also includes the computation of the payoff, that's an additional linear time operation, because we must simultaneously iterate through every element of A and B, and apply constant time math operations. But since computing the actual payoff is linear, it doesn't affect the running time of our whole algorithm, and we're left with a $\Theta(n \lg n)$ running time.