Assignment 4

Name: Mousumi Akter
SID: 904095347
I Discussed with: N/A

April 18, 2021

Greedy Algorithm

1. (16.1-2) To prove that the stated approach yields an optimal solution, we have to prove two things: (1) that the choice being made is the greedy choice (this proof will be along the lines of the proof of Theorem 16.1 in the text; but do not copy that proof; your proof will be different, yet similar in structure), and (2) that the resulting solution has optimal substructure.

   The question in 16.1-2 is:

   "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."

   We have to prove in two things firstly, we need to prove that the choice being made is the greedy choice and secondly, we need to prove that the resulting solution has optimal substructure.

   In this algorithm the activities are sorted according to their starting time, from the latest to the earliest, where a tie can be broken arbitrarily. Then the activities are greedily selected by going down the list and by picking the activity that is compatible with the current selection. It is greedy because we make the best looking choice at each step.

   Activities i and j are compatible if their time periods are disjoint, that is, $s_j \geq f_i$ or $s_i \geq f_j$.

   Now, consider any nonempty subproblem S_k, and let a_m be an activity in S_k with the latest start time. Then for the first part we need to prove that am is included in some maximum-size subset of mutually compatible activities of S_k.

   Proof of part 1:

   Let A_k be a maximum-size subset of mutually compatible activities in S_k, and let a_j be the activity in A_k with the latest start time. If a_j

$= a_m$, we are done, As we have shown that a_m is in some maximum-size subset of mutually compatible activities of S_k. If $a_j \neq a_m$, let the set $A_k^{'} = A_k - \{a_j\} \cup \{a_m\}$ be be A_k but substituting a_m for a_j. The activities in $A_k^{'}$ are disjoint, which follows because the activities in A_k are disjoint, a_j is the last activity in A_k to start, and $s_m \geq s_j$. Since $|A_k^{'}| = |A_k|$, we conclude that $A_k^{'}$ is a maximum-size subset of mutually compatible activities of S_k, and it includes a_m.

Proof of part 2: A problem exhibits optimal substructure if an optimal solution to the problem contains within it optimal solutions to subproblems. In essence, all we really need to do is to argue that an optimal solution to the subproblem, combined with the greedy choice already made, yields an optimal solution to the original problem.

Once we make the first greedy choice, now the problem is $S^{'} = \{i \in S : f_i \leq s_1\}$, with optimal solution $A^{'} = A - \{a_1\}$. This solution must be optimal. If $B^{'}$ solves $S^{'}$ with more activities than those in $A^{'}$, adding activity a_1 to $B^{'}$ makes it bigger than A, which is a contradiction. Therefore, greedy choice yields an optimal solution.

2. (16.1-4) Suppose that we have a set of activities to schedule among a large number of lecture halls, where any activity can take place in any lecture hall. We wish to schedule all the activities using as few lecture halls as possible. Give an efficient greedy algorithm to determine which activity should use which lecture hall.

   Sort the activities in order of start time. Assign each activity to a Lecture Hall that has already some scheduled activity but is available for the activity at the starting time the activity under consideration. If no such Lecture Hall is available, assign the activity to a new Lecture Hall. This algorithm ensures that fewer lecture halls are engaged at any point of time and the activities are compatible. Let a_i be i^{th} activity and there be total n activities.

Algorithm 1 Lecture Hall Assignment

```
1:  procedure LectureHallAssignment(S)        ▷ S be a set of n activities.
2:      Sort the activities in order of start time
3:      HallCount = 1
4:      i = 1
5:      Assign a_i to L_HallCount                    ▷ a_i denotes i^th activity
6:      for i= 2 to n do
7:          if a_i is compatible with any L_HallCount that has already some sched-
    uled activity then
8:              Assign a_i to L_HallCount
9:          else
10:             HallCount = HallCount + 1
11:             Assign a_i to L_HallCount
```

Running time of this algorithm is O($nlogn$).

3. (16.2-1) Prove that the fractional knapsack problem has the greedy-choice property.

   A optimal solution to the fractional knapsack is one that has the highest total value density. Since we are always adding as much of the highest value density we can, we are going to end up with the highest total value density. Suppose that we had some other solution that used some amount of the lower value density object, we could substitute in some of the higher value density object meaning our original solution could not have been optimal.

   Let item i be of value v_i and weight w_i. Let j be the item with maximum $\frac{v_j}{w_j}$. Then according to greedy choice property, there exists an optimal solution in which one can take as much of item j as possible.

   Proof: Let us say that there exists an optimal solution in which one didn't take as much of item j as possible. In this situation there are two possibilities – either the knapsack is not full or knapsack is full.

   If the knapsack is not full, one can add some more of item j, and one can have a higher value solution. Therefore, this proves that if knapsack is not full then by not filling knapsack with item j, optimal solution was not obtained. We arrive at a contradiction.

   Now, let us assume that the knapsack is full. In such a situation, there must exist some item $k \neq j$ with the relationship $\frac{v_k}{w_k} < \frac{v_j}{w_j}$ where item k is in knapsack. At the same time, entire item $j$must not be in the knapsack. One can therefore take a piece of item k, weighing σ (where $\sigma > 0$) out of the knapsack, and put an equal weight of j, that is, σ (where $\sigma > 0$) into the knapsack, This increases the value of knapsack's value by $\sigma\frac{v_j}{w_j} - \sigma\frac{v_k}{w_k} = \sigma(\frac{v_j}{w_j} - \frac{v_k}{w_k}) > 0$ because both σ and $\frac{v_j}{w_j} - \frac{v_k}{w_k}$ are greater than zero. This leads us to contradiction that original solution was optimal.

   Thus, an optimal solution to the fractional knapsack is one that has the highest total value density.

   In terms of algorithm, it turns out as below:

Algorithm 2 Fractional Knapsack Problem

```
1: procedure FRACTIONAL KNAPSACK(S)          ▷ S be a set of n items.
2:     Sort items by v_i/w_i                 ▷ v_i denotes value and w_i weight of i^th item
3:     for i= 1 to n do
4:         Put as much of item i as possible into the knapsack
```

4. (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 i-th element of set A, and let b_i be the i-th

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.

Algorithm 3 Maximize Payoff Problem

1: **procedure** MAXIMIZEPAYOFF(A, B) ▷ A and B be a set of n integers.
2: `Sort A in nondecreasing order` $a_1 \geq a_2 \geq \ldots \geq a_n$
3: `Sort B in nondecreasing order` $b_1 \geq b_2 \geq \ldots \geq b_n$
4: **return** $\prod_{i=1}^{n} a_i^{b_i}$

Proof: Since an identical permutation of both sets doesn't affect this product, suppose that A is sorted in ascending order. Then, we will prove that the product is maximized when B is also sorted in ascending order. To see this, suppose not, that is, there is some $i < j$ so that $a_i < a_j$ and $b_i > b_j$. Then, consider only the contribution to the product from the indices i and j. That is, $a_i^{b_i} a_j^{b_j}$, then, if we were to swap the order of b_i and b_j , we would have that contribution be $a_i^{b_j} a_j^{b_i}$, we can see that this is larger than the previous expression because it differs by a factor of $(\frac{a_j}{a_i})^{b_i - b_j}$, which is bigger than one. So, we couldn't of maximized the product with this ordering on B.

Running Time: If the two sets A and B are already sorted, the running time complexity is O(n). If the sets are not sorted, then due to sorting them first, the running time complexity is O(n log n).

5. . Let's consider a long, straight country road with n houses scattered very sparsely along it. We can picture this road as a long line segment with an eastern endpoint and a western endpoint. Let d_i be the distance of the i-th house from the eastern endpoint. We are given as input d_i, for all $1 \leq i \leq n$. Further, let's suppose that despite the bucolic setting, the residents of all these houses are avid cell phone users. You want to place cell phone base stations at certain points along the road, so that every house is within four miles of one of the base stations. Give an efficient algorithm that achieves this goal, using as few base stations as possible. Identify the complexity of your algorithm.

   We need to provide cell phone signal coverage to all the houses with minimum number of base stations. It is given that base stations provide coverage upto a radius of 4 miles. It is obvious that if two houses are 8 miles apart on a straight line then one base station at the middle point will provide needed coverage.

   Keeping this in mind, let us start at the western end of the road and move east until the first house h is exactly four miles on the western side. We place a base station at this point. Now remove all the houses from the set of houses to be covered that are covered by this base station.

We repeat the same process for the remaining houses. Let B be the set of BaseStation, r be radius of coverage of Base Station and c be the number of covered houses.

Algorithm 4 Base Station Problem

```
1: procedure PlaceBaseStations(H, r)          ▷ H be a set of n houses
                                              ▷ r be radius of coverage of Base Station
2:     Sort d_i in monotonically decreasing order for 1 ≤ i ≤ n
3:     B = ∅                                  ▷ B be the set of BaseStation
4:     i = 0
5:     Start at the western most point of the road
6:     while i < n and c < n do               ▷ c be the number of covered houses
7:         Travel east until we are exactly r miles east of an
   uncovered house
8:         Place a base station at current location
9:         i = i + 1
10:        B = B ∪ Location[i]
11:        Ignore all houses covered by base stations at locations
   j where 1 ≤ j ≤ i
12:        if    h[i] number of houses are covered by BaseStation at
   Location[i] then
13:            c = c + h[i]
14:        return i, B
```

Running Time: Line 2 has complexity of O(nlogn). Lines 3, 4 and 5 have O(1). Lines 6 to 13 have complexity of O(n). Hence the overall complexity is O(n logn). If the houses are already sorted then the complexity will be O(n).