Homework 8

Tushar Jain, N12753339
CSCI-GA 1170-003 - Fundamental Algorithms

November 2, 2017

Problem 1. You are opening a new chain of hipster artisanal locally-sourced vegan coffe shops on First Avenue. You are given the expected revenue $r_i \geq 0$ of opening a shop on i-th street, for $1 \leq i \leq n$. Your only constraint is that you cannot open two shops on adjacent streets, i.e. on streets j and $j+1$ for some j. The task is to find the maximum expected revenue (forget the actual position of the shops for now) of opening a chain of shops subject to your constraints.

Problem 1.a. First give an example to show that the best solution does not necessarily have to pick $\lfloor n/2 \rfloor$ streets. That is, it is not enough to consider only the solution of picking all the odd-numbered streets, and all the even-numbered streets.

Solution.

Counter-example for picking only even streets:
Example: [10,2,1,11]
Given Algorithm's Answer: 2+11=13
Correct Answer: 10+11 = 21

Counter-example for picking only odd streets:
Example: [10,2,1,11] *Given Algorithm's Answer:* 10+1 = 11
Correct Answer: 10+11=21

□

Problem 1.b. Give an example to show that this algorithm *does not* always find an optimal solution: pick the street with the maximum revenue r_i, set the value of r_{i-1}, r_i, r_{i+1} to zero and repeat.

Solution.

Counter Example: [2,4,3]
Given Algorithm's Answer: 4
Correct Answer: 2+3 = 5 □

Problem 1.c. Give a dynamic programming algorithm to solve this problem in $O(n)$ time, by giving the recurrence equation along with a proof that it finds the optimal solution.

Solution.

Assumption: To use the recurrence, $n \geq 3$.
Recurrence Equation:

$$DP[i] = max(DP[i-1], DP[i-2] + r_i)$$

Pseudo-Code:

```
1  def maxRevenue(r):
2      n = len(r)
3      if n < 3:
4          return max(r)
5
6      DP = [0]*(n+1)
7      DP[1] = r[0]
8
9      for i in range(2,n+1):
10         DP[i] = max(DP[i - 1], DP[i - 2] + r[i-1])
11
12     return DP[n]
```

Running Time:
Looking at for loop starting at line 9, we can clearly see the total running time would be $O(n)$ as all other lines have $O(1)$ running time individually.

Correctness:
The algorithm can be proved using the following loop invariant:

The DP[i-1] contains the maximum revenue possible if only $i-1$ streets were given before the start of the i-th iteration.
Initialization: At $i = 2$, $i - 1 = 1$ street. Therefore, max revenue is revenue of the shop at street 1 which is the same as $DP[1]$
*Maintenance:*In every iteration, we chose max of the 2 cases of opening a shop here (and 2 streets prior) or opening a shop at the previous street. This leads to an optimal substructure and thus the loop invariant will hold for the next iteration as well.
Termination: After n+1 iterations, DP[n] contains the max possible revenue for n streets and which is exactly what the answer is.

□

Problem 1.d. Now suppose the city doesnt let you buy a shop on every street, and so you are instead given a list of street numbers $x_1, ..., x_n$ and the expected revenues $p_1, ..., p_n$ where $pi \geq 0$ is the revenue of opening a shop on x_i-th street. You also have a number K which is the minimum number of streets you need between two of your shops (so the previous parts were using $K = 2$). Modify your recurrence equation from the previous subproblem to solve this problem.

Solution.

Modified Recurrence Equation:

$$DP[i] = max(max_{j<i}(DP[j] + check(i,j) * DP[i]), r_i)$$

where the function check is defined as follows:

$$check(i,j) = \begin{cases} 0 \text{ if } x_i - x_j < k \\ 1 \text{ if } x_i - x_j \geq k \end{cases}$$

Pseudo-Code:

```
1  def check(i,j,x):
2      return int(x[i-1] - x[j-1] >= k)
3
4  def maxRevenue(r,x):
5      DP = [0]*(n+1)
6      DP[1] = r[0]
```

```
7
8       for i in range(2,n+1):
9           for j in range(1,i-1):
10              temp = DP[j] + check(i,j,x)*DP[i]
11              if temp > DP[i]:
12                  DP[i] = temp
13          DP[i] = max(DP[i], r[i-1])
14
15      return DP[n]
```

Running Time:
Looking at for loops starting at line 8 and 9, we can clearly see the total running time would be $O(n^2)$ as all other lines have $O(1)$ running time individually.

□

Problem 2. You graduated from NYU and set up a fancy new start-up that is going to change the world. Your start-up grows rapidly, and soon you have two offices, one in the Upper East Side, and one in the Upper West Side. Every day, you need to decide which office to work from. You have with you the list of upcoming clients to each office, along with the extra money youd make if you personally worked with them. That is, assume you are given $E_1, ..., E_n$ and $W_1, ..., W_n$ where E_i is the money youd make by working in the UES office on day i and W_i is the money youd make by working in the UWS office.

Since its a start-up you will also sleep in the office that you worked from, and so if you want to work from the other office the next day, you need to walk through Central Park in the morning. This takes ages, and so you expect to lose a fixed sum of C every time you switch offices. Your task is to calculate the maximum money you can make by scheduling your work optimally for the next n days.

Problem 2a. Give an example (i.e. values for E_i , W_i and C) where the following algorithm does not return the maximum value: each day, pick the office with the higher expected money made.

Solution:

Counter-example for picking simply picking higher expected money:
Example:
E: [1,2,3]
W: [2,1,4]
C = 1
Given Algorithm's Answer: 2+(2-1)+(4-1)=6
Correct Answer: 2+1+4 = 7

□

Problem 2b. Give a polynomial time algorithm to find the maximum money you can make. Briefly justify why it is optimal.

Solution:

Recurrence Equation:

$$DP[i] = \begin{cases} max(DP[i-1] + E[i], DP[i-1] + W[i] - C) & \text{, if at day } i-1 \text{, we were at E} \\ max(DP[i-1] + W[i], DP[i-1] + E[i] - C) & \text{, if at day } i-1 \text{, we were at W} \end{cases}$$

Assumption: The given graph is DAG.

Pseudo-Code:

```
1  def maxProfit(E,W,C):
2      n = len(E)
3      DP_E = [0]*n
4      DP_W = [0]*n
5      DP_E[0] = E[0]
6      DP_W[0] = W[0]
7
8      for i in range(1,n):
9          DP_E[i] = max(DP_E[i-1] + E[i], DP_W[i-1] + E[i] - C )
10         DP_W[i] = max(DP_W[i-1] + W[i], DP_E[i-1] + W[i] - C )
11
12     return max(DP_E[n-1], DP_W[n-1])
```

Running Time:
Looking at for loop starting at line 8, we can clearly see the total running time would be $O(n)$ as all other lines have $O(1)$ running time individually.

Optimal:
As we solve each subproblem which is optimal substructure, we get optimal solution in the end. □

Problem 3a. Give an example to show that this strategy is not optimal for the activity selection problem: always picking the shortest (least duration) activity from the remaining compatible activities until none remain.

Solution:

Counter-example for picking simply picking activity with shortest duration:
Activities with starting and ending time: [(1,5),(4,7),(6,10)]
Given Algorithm's Answer: 1 activity i.e. (4,7) as 7-4 =3 is the shortest
Correct Answer: 2 activities i.e. (1,5) and (6,10)

□

Problem 3b. Give an example to show that this strategy is not optimal for the activity selection problem: always picking the compatible remaining activity with the earliest start time.

Solution:

Counter-example for picking simply picking activity with earliest start time:
Activities with starting and ending time: [(1,5),(2,3),(4,10)]
Given Algorithm's Answer: 1 activity i.e. (1,5) as 1 is the smallest start time.
Correct Answer: 2 activities i.e. (2,3) and (4,10)

□

Problem 4. You decide to do the Apalachian trail, which is long hike, with places to stop and rest at $x_1, ..., x_n$ miles from the beginning of the trail (x_n is also the end of the trail). You are super fit, but still can only hike a maximum of M miles a day (dont worry, the stops are never more than M miles apart). Youre in a rush, so you want to find the minimum number of stops you need to make.

Problem 4a. Give a dynamic-programming-style recurrence equation for the minimum number of stops you need to make. Briefly justify the correctness of this equation.

Solution:

Recurrence Equation:

$$DP[i] = \begin{cases} 0 & \text{, if i=0} \\ 1 + min_{j<i}(DP[i-j]) & \text{, for j such that } x[i] - x[i-j]) \leq M \end{cases}$$

Correctness:
The first stop can be reached without any stops (as stops are never more than M miles apart is given). To reach other stops in the minimum jumps we check all possibilities of previous stops(optimal subproblem), and then continue doing the same for next stops(overlapping subproblems). Therefore, the Minimum to reach any other stop is 1 + minimum to reach all previous stops such that distance between previous stop and current stop $\leq M$. □

Problem 4b. Give a greedy algorithm to find the minimum number of stops, and prove that it is correct.

Solution:

In every iteration, we go to the furthest stop within the given m miles.
Assumption: x is given in sorted order.
Pseudo-Code:

```
1  def minStops(stops, m):
2      count = 0
3      maxreach, idx = 0, 0
4
5      while maxreach + m < stops[-1]:
6          j = 0
7          while stops[j] - maxreach <= m:
8              j += 1
9          idx = j-1
10         maxreach = stops[idx]
11         count += 1
12     return count
```

Correctness: The greedy solution solves this problem optimally, where we maximize distance we can cover from a particular point such that there still exists a stop to get rest before running again the next day. The first stop is at the furthest point from the starting position which is less than or equal to M miles away. The problem exhibits optimal substructure, since once we have chosen a first stopping point p, we solve the subproblem assuming we are starting at p. Combining these two plans yields an optimal solution for the usual cut-and-paste reasons. Now we must show that this greedy approach in fact yields a first stopping point which is contained in some optimal solution. Let O be any optimal solution which has in which we stop at positions $o_1, o_2, ..., o_k$. Let g_1 denote the furthest stopping point we can reach from the starting point. Then we may replace o_1 by g_2 to create a modified solution G, since $o2 - o1 < o2 - g1$. In other words, we can actually make it to the positions in G without running out of stamina. Since G has the same number of stops, we conclude that g_1 is contained in some optimal solution. Therefore the greedy strategy works. □

Problem 4c. What are the running times of both your above algorithms?

Solution:

Running Time:

- Dynamic Programming: $O(n^2)$
- Greedy Algorithms: $O(n)$

□

Problem 5. Little Johnny is extremely fond of watching television. His parents are off for work for the period $[S, F)$, and he wants to make full use of this time by watching as much television as possible: in fact, he wants to watch TV non-stop the entire period $[S; F)$. He has a list of his favorite n TV shows (on different channels), where the i-th show runs for the time period $[s_i, f_i)$, so that the union of $[s_i, fi)$ fully covers the entire time period $[S, F)$ when his parents are away.

Little Johnny does not mind switching in the middle of a show he is watching, but is very lazy to switch TV channels, so he wants to find the smallest set of TV shows that he can watch, and still stay occupied for the entire period $[S, F)$. Design an efficient $O(nlogn)$ greedy algorithm to help Little Johny.

Solution: We first sort the inputs according to starting times and then at every point select the channel which has start time less than current shows finishing time and the furthest possible finish time.

Assumption: At any time, at least one show must be running.

Pseudo-Code:

```
1  def minTVShows(shows, s, f):
2      sorted_shows = sorted(shows, key= lambda x: x.s)
3
4      curr = TVShow(s,s)
5      curr_s = s
6      ans = set()
7
8      i = 0
9      while i < len(sorted_shows):
10         if sorted_shows[i].s <= curr_s:
11             flag = True
12             if sorted_shows[i].f > curr.f:
13                 curr = sorted_shows[i]
14             i += 1
15         else:
16             flag = False
17             ans.add(curr)
18             curr = (curr.f, curr.f)
19             curr_s = curr.s
20
21     if flag:
22         ans.add(curr)
23     return ans
```

Running Time: Since we sort the list of shows according to their starting time which takes $O(n \log n)$ in running time. Then we traverse every show in the list of TV show doing $O(1)$ iteration, the traversal would take $O(n)$. Thus, the total running time would be $o(n \log n)$.

Correctness: After sorting the shows in sorted order, we find the show with the show with max finishing time and least starting time. We will prove the correctness of the algorithm using the following loop invariant:

At the start of each while loop on line 9, ans contains the the set of TV shows requiring least change of channels up to the i-th show's starting time.

Initialization: Before the first loop, i = 0, the ith show is the first show in the period when the parents are not at home, so before this time his parents were home and thus the set is empty. This is correct and thus loop invariant holds true.

Maintenance: In every loop, the algorithm selects the show which has the maximum finish time and start time less than finish time of current show. For any other selection of TV show, the number of switches would be more as it will finish before the above selection. Thus, the correct solution is to pick TV show with max finish time. Since the shows are sorted according to their start times, we check every show only once in the algorithm.

Termination: When $i = n-1$, we have the optimal TV shows in ans set due to the maintenance property of having max finishing time. □