Solutions for 611 Homework 2

Mahim Agarwal, Hanwen Xiong, and Jackson Warley

October 6, 2017

1 Solution to Question 1

Let $\mathcal{M} = (E, \mathcal{I})$. For all $i \in [k]$, we have $|\emptyset \cap E_i| = 0 \leq 1$, so $\emptyset \in \mathcal{I}$. To prove heredity, let $S \in \mathcal{I}$ and suppose $R \subseteq S$. Then, if $i \in [k]$, we have $R \cap E_i \subseteq S \cap E_i$, implying $|R \cap E_i| \leq |S \cap E_i| \leq 1$. To prove exchange, first note the following property: because the (nonempty) sets $E_1, \ldots, E_k$ partition E and each independent set intersects any E_i in at most one element, any $S \in \mathcal{I}$ has nontrivial intersection with exactly $|S|$ of the sets $E_1, \ldots, E_k$. Suppose $A, B \in \mathcal{I}$ with $|A| < |B|$. Then B intersects $|B| > |A|$ of the sets $E_1, \ldots, E_k$, so there must be some i such that $E_i \cap A = \emptyset$ and $E_i \cap B = \{b\}$. Then $A \cup \{b\} \in \mathcal{I}$, since $|(A \cup \{b\}) \cap E_i| = 1$, proving that $\mathcal{M}$ is a matroid.

2 Solution to Question 2

We will present the algorithm first, followed by the analysis.

---

function ALG($P = \{p_1, p_2, \ldots, p_n\}$)
  $I \leftarrow \{[p_1, p_1 + 1]\}$
  for $p \in P$ left to right **do**
    $[a, b] \leftarrow \text{last}(I)$
    if $p > b$ **then**
      $I \leftarrow I \cup \{[p, p + 1]\}$
    end if
  end for
end function

Claim: ALG runs in $\mathcal{O}(n)$ time and returns the smallest possible set of intervals covering all the points in P.

Proof. Since ALG terminates exactly when we have considered every point in P, and P is finite, ALG must terminate. When it does, I will contain some number of intervals, greater than 1. If ALG ever encounters a point p that is not covered by an interval already in I, it adds an interval containing p to I. Thus, since every point is considered, all points are covered when ALG terminates.

Now, we must show that P cannot be covered using fewer than $|I|$ (after ALG has run to completion) unit-length intervals. Let $P' = \{p \in P \mid [p, p+1] \in I\}$. Since every interval in I begins with a point from P, $|P'| = |I|$. Furthermore, since ALG only constructs a new interval when the nearest endpoint is further than 1 away from the point under consideration, no two points in P'

are within unit distance of each other. Thus, any cover of P' by unit intervals must include at least $|P'| = |I|$ intervals, and in particular the solution yielded by ALG is optimal.

Finally, ALG considers each point in P exactly once. The work of comparing the points and retrieving the most recently added interval requires only constant time, so considering a single point is $\mathcal{O}(1)$. Thus, ALG has a total runtime of $\mathcal{O}(n)$. □

3 Solution to Question 3

(1) ($\Rightarrow$) We prove the contrapositive. Let $S \subseteq J$ and suppose that there exists some $d \in [n]$ for which $S_d > d$. Then, by the pigeonhole principle, in any ordering of S there must be some job s_i with $i > d$ and $d_{s_i} < d$. Since we can idenfity an overdue job s_i for any ordering of S, S must not be good.

($\Leftarrow$) Let $S \subseteq J$ with $S_d \leq d$ for all $d \in [n]$ and suppose toward a contradiction that S is not good. Then in any ordering of S, there is some overdue job. Let $d+1$ be the index of the first overdue job in S and choose the ordering $S = \{s_1, \ldots, s_k\}$ that maximizes d. Then S is of the form $S = \{s_1, \ldots, s_d, s_{d+1}, \ldots, s_k\}$ where jobs s_1 through s_d yield bonuses and s_{d+1} does not. Suppose that for some $i \in [d]$, we have $d_{s_i} > d$. Then we could exchange s_i with s_{d+1}, after which s_{d+1} would yield a bonus, and, since $d_{s_i} > d$, so would s_i. However, this would contradict the maximality of d, since it would result in an ordering in which the first $d+1$ jobs yield bonuses. This means that for all $i \in [d]$, $d_{s_i} \leq d$, implying that $S_d \geq d$. But we also have $d_{s_{d+1}} \leq d$, since s_{d+1} is overdue. Thus $S_d > d$, a contradiction, whereby S must be good.

(2) To prove heredity, suppose that $A \in \mathcal{I}$. Let $A = \{a_1, \ldots, a_k\}$ be a good ordering of A, and let $B \subseteq A$ be ordered identically up to removal of elements. Since B is obtained by removing jobs from A, the index in B of any job in $A \cap B$ is at most its index in A, so any job that yielded a bonus in A still yields one in B. Thus, since A was good, so is B, whence $B \in \mathcal{I}$ as desired.

Now, to prove exchange, let $A, B \in \mathcal{I}$ with $|A| < |B|$. Suppose that $A = \{a_1, \ldots, a_k\}$ and $B = \{b_1, \ldots, b_j\}$. Let $A' = A \cup \{b_j\}$. Since $B \in \mathcal{I}$, we must have $d_{b_j} \geq j \geq k+1$, therefore b_j yields a bonus in A'. The other jobs in A' have not moved from their positions in A, so they still yield bonuses. Thus $A' \in \mathcal{I}$, implying $(J, \mathcal{I})$ has independence.

(3) Let $w(j) = b_j$ for all $j \in J$. Since $(J, \mathcal{I})$ is a matroid, we know that we can apply the matroid greedy algorithm to find a maximum-weight independent set. If J is the set of jobs (somehow tied to their deadlines and bonuses so that both can be accessed in constant time), and MGA is the matroid greedy algorithm with the weight function $w(j) = b_j$, our algorithm is as follows:

---

function ALG(J)
  call MGA(J) to find a maximum-weight good set S
  sort S in non-decreasing order by deadline
  append $J \setminus S$ to S in any order
  return the resulting list
end function

Claim 1: ALG terminates in $\mathcal{O}(n \log n)$ time.

Proof. The total runtime of ALG is the runtime of MGA plus the $\mathcal{O}(n \log n)$ required to sort S and append $J \setminus S$ to it (the latter operation is at most linear in n). MGA first requires $\mathcal{O}(n \log n)$ to sort the jobs by bonus. Next, it considers each job once, choosing whether to include it or not based on whether it maintains the independence of the accumulator set. By Part (1), we can check whether the inclusion of a job maintains independence in constant time; we merely keep track of S_d as elements are added, and check only the new candidate element to determine its inclusion. Thus, the remaining work done by MGA requires only $\mathcal{O}(n)$ time, so the total runtime of MGA, and therefore also of ALG, is $\mathcal{O}(n \log n)$. □

Lemma 3.1. *If S is a good set, the earliest-deadline-first ordering completes every job in S.*

Proof. Let $S = \{s_1, \ldots, s_k\}$ be any ordering that completes every job in S. If this ordering is not the earliest-deadline-first ordering, there is some pair of jobs s_i and s_j so that $i < j$ and $d_{s_i} > d_{s_j}$. Exchanging the positions of these two jobs does not cause any jobs to become overdue, since s_j is completed earlier than it previously was, s_i is due after s_j (which was not overdue in its original position), and the other jobs are unaffected. Since we can perform these exchanges until no more such pairs exist, we can transform any witness ordering of a good set into the earliest-deadline-first ordering without causing any jobs to become overdue. □

Lemma 3.2. *If S is a maximum-weight good set, then any ordering of J that completes every job in S yields the maximum bonus amount.*

Proof. Let $J = \{j_1, \ldots, j_n\}$ be an ordering of J that completes every job in S. Suppose toward a contradiction that there is some other ordering $J' = \{j'_1, \ldots, j'_n\}$ of the same jobs that yields a greater sum of bonuses than J. Let $Q = \{j' \in J' \mid j' \text{ is completed on time in } J'\}$. Then Q is a good set, as witnessed by the ordering given by removing $J' \setminus Q$ from J' and reindexing its remaining elements in the order they originally appeared. But, since the removed elements were not completed on time, they did not contribute to the sum of bonuses yielded by J'. Thus, Q is a good set yielding the same sum of bonuses as J', and in particular it yields more than J, which completed every item in the maximum-yield good set. Thus Q contradicts the maximality of S, and therefore there must be no ordering of J' that yields more in total than J. □

Claim 2: ALG returns an ordering that yields the maximum possible bonus.

Proof. By construction, the ordering returned by ALG includes the maximum-weight good set, in earliest-deadline-first order. By Lemma 3.1, every job in this maximum-yield set is completed. By Lemma 3.2, this implies that ALG gives the ordering that maximizes the total bonus yield. □

4 Solution to Question 4

4.1 Dynamic Approach

Definition 4.1.1: *Maximal Sequence Sum of $\{a_1, \ldots, a_i\}$ is the optimal sequence sum when we're at the a_i entry*
Definition 4.1.2: *S_i = the optimal sequence sum of $\{a_1, \ldots, a_i\}$ including a_i*
Algorithm:

create an array S of size $n+1$ and make the $0th$ entry 0
$maximum = 0, maximum\_index = 0$
for $i \in [n]$ **do**
  $S[i] = max\{S[i-1], 0\} + a_i$
  if $S[i] > maximum$ **then**
    $maximum = S[i], maximum\_index = i$
  end if
end for
Scan from maximum_index to its left until we hit a non-positive entry of S
The entries scanned(not including the non-positive one) is the MSS of the whole sequence

Proof. Whenever we go to the next number a_i we either chose to or not to include it in the Maximal Sequence Sum so far at a_{i-1}. So we have this two cases: case 1: a_i included; case 2: a_i not included.

For case 1, we have to add a_i to our Maximal Sequence Sum of $\{a_1, \ldots, a_{i-1}\}$, which is S_i by definition; For case 2, we know if a_i is not included, then the Maximal Sequence Sum will be 0 at a_i because the sequence sum has to be sum of contiguous numbers. In this case, Maximal Sequence Sum is always 0 so that we utilize this property and don't have to create another array.

Obviously the Maximal Sequence Sum at a_i is the larger one of case 1 and case 2 as we only have this two cases. Therefore, at any a_i we have this $2 * n(1 * n$ in actual operation) table that tracks the Maximal Sequence Sum. The maximum value of this table then is the Maximum Sequence Sum. The sequence should start from the last number that's not included, at which the Maximal Sequence Sum is 0. a Maximal Sequence Sum of 0 suggests $S_i <= 0$. So it's reasonable for our algorithm to stop scanning at that entry and return the sequence. □

Time Complexity: It takes $\mathcal{O}(n)$ to create a length-$(n+1)$ array and $\mathcal{O}(n)$ to traverse n numbers. And we have to scan at most $n+1$ entries of array S which also takes $\mathcal{O}(n)$ to conduct. Therefore, the time complexity is $\mathcal{O}(n)$.

4.2 Divide-and-Conquer Approach

Algorithm:

1.Divide the Sequence into equal halves, get two sub-MSS MSS_l and MSS_r
2.Keep adding nubmers from the boder of MSS_l and MSS_r to its left until we hit the begining of MSS_l, record the maximum sum as MSS_{bl}
3.Keep adding nubmers from the boder of MSS_l and MSS_r to its right until we hit the end of MSS_r, record the maximum sum as MSS_{br}
4.return $max\{MSS_l, MSS_r, MSS_{bl} + MSS_{br}\}$

Proof. When we merge the two sequences it's possible for us to get a even larger MSS than MSS_l and MSS_r. And we can only do this by crossing the middle. Because if we can add more numbers from the two "sides" then we'd get a larger MSS_l or MSS_r which results in contradiction.

From above we can conclude that there're 4 cases:

- MSS_l+ some sequence to its right in the "middle"

- MSS_r+ some sequence to its left in the "middle"
- MSS_l+ "middle" sequence + MSS_r
- some sequence in the "middle" and is not right next to MSS_l or MSS_r

It's obvious that our algorithm covers all the cases, and it chooses the maximum of MSS_l, MSS_r and $MSS_{bl} + MSS_{br}$. So if there's a larger sequence crossing the middle we'll find and return that

□

Time Complexity: Every time we divide the problem into 2 subproblems of half size. And the merge step scan from the middle to the two sides which takes linear time. So we get $T(n) = T(n/2) + O(n)$. According to Master's Theorem, the total time complexity is $O(nlogn)$

5 Solution to Question 5

Let us suppose, input array $P = \{P_1, \ldots, P_n\}$ and we need to find Longest decreasing subsequence (LDS). We will use the concept of dynamic programming where we will find the LDS for $\{P_i, \ldots, P_n\}$ given that we know LDS for $\{P_{i+1}, \ldots, P_n\}$
Here is our proposed algorithm:
Algorithm:
Consider three arrays as follows:

- input array $P = \{P_1, \ldots, P_n\}$
- dynamic programming array $D = \{D_1, \ldots, D_n\}$ with all elements initialized to 1 and where D_i is the longest decresing sub-sequence length from P_i to P_n **WITH** P_i **INCLUDED**
- predecessor array $S = \{S_1, \ldots, S_n\}$ with all elements initialized to -1 and where S_i is the index of the element that is the predecessor to P_i in the longest decreasing subsequence from P_i to P_n

Now, algorithm is as follows:

---

for $i \in \{n-1, n-2, \ldots, 1\}$ **do**
  for $j \in \{n, n-1, \ldots, i+1\}$ **do**
    if $P_i > P_j$ and $D_i > 1 + D_j$ **then**
      $D_i += 1, S_i = j$
    end if
  end for
end for
Find the the index k of the maximum element in array D
append P_k to the result sub-sequence, find index of its predecessor from S_k to append the element to final list. Keep appending the predecessors until $S_x = -1$, at which point we will return since it is the last element of resulting LDS.

Time complexity: We do the outer loop and inner loop roughly n times, which suggests complexity of $\mathcal{O}(n^2)$. Traversing the array D and S both take $\mathcal{O}(n)$. So we have $T(n) = \mathcal{O}(n^2) + \mathcal{O}(n) = \mathcal{O}(n^2)$

Accuracy explanation:

proof of concept: We know that for every iteration on P_i, D_{i+1} to D_n contains length of LDS including that particular D_j. So, when we traverse from P_n to P_{i+1}, for any element P_j, if LDS is $\{P_j, P_l, P_o\}$ and if $P_i > P_j$, then $P_i > P_j > P_l > P_o$. Thus LDS at P_i is $\{P_i, P_j, P_l, P_o\}$ which has length 1 greater than LDS length at P_j. Also, since we only update the D_i when new length is greater than the existing one, we will find the sequence with the longest LDS at D_i with element at index i included. Once we reach the max possible length of LDS, we can produce the final sequence by traversing through the predecessor array S, since for every time we update D_i, S_i contains the next element of the LDS starting from P_i

proof of termination: Since, we have intialized S array with -1, for every S_x the value is either -1 or an index from P array (1 to n).

Case 1: $S_i = -1$; This means that this element has no element to its right which is smaller than P_i. So, this must be the smallest element of any LDS which includes it and thus our algorithm should terminate here which it does.

Case 2: $S_i \in \{1, 2, \ldots, n\}$; Then we recurse to its next smaller element and if it's the smallest we terminate. Else we go to next smaller.

6 Solution to Question 6

6.1 Solution H

Algorithm:

Create a list $c[n]$ which stores $(c_i + i - 1, i)$ in its $i - th$ entry. Index starts from 1
Sort the list by the first element of each tuple, get the sorted array $c_{sorted}[n]$
create an empty list L, a variable $i = 0$
for $c' \in c_{sorted}[n]$ **do**
  if the second element of $c' > i$ **then**
    L.append(c'), update $i = c'$
  end if
end for

Definition 6.H.1: $Cost(i, j) = c_j + \sum_{j \in [n]} u_j$. This is the cost of storing file on server S_i plus the cost of some user(s) accessing file on server S_j

Proof. Suppose a user wants to access server S_i. The user can either find the file on S_i or on some S_j where $j > i$. The $Cost(i, j) = c_j + (j - i)$. Consider the condition where we have to find the file elsewhere. Then for the cost to be cheapest, we have to satisfy $Cost(i, j) < Cost(i, i)$, i.e. $c_j + (j - i) < c_i$ and $Cost(i, j)$ is the minimum $\forall j > i$. It's not hard to find for all access between S_i and S_j, the cheapest cost will always be achieved by saving the file on S_j. Because if not, there exists $Cost(m, n) < Cost(m, j)$ where $i < m \leq n < j \Rightarrow c_n + (n - m) < c_j + (j - m) \Rightarrow c_n + (n - m) - i + i < c_j + (j - m) - i + i \Rightarrow c_n + (n - i) < c_j + (j - i) \Rightarrow Cost(i, n) < Cost(i, j)$. This contradicts to our assumption that $Cost(i, j)$ is the minimum

For our algorithm, we first sort the list $c[n]$. The element in it is well designed so that $\forall Cost(i,j) < Cost(i,i), c[j] < c[i]$. Therefore, everytime we get an element out of $c_{sorted}[n]$, we don't consider any server with a subscript less than the current subscirpt. Finally we may get an $L = \{l_1, l_2, \ldots, l_m\}$. From the proof above we know access to S_{l_i} can always find the file at its corresponding interval. Since the user won't access from right to left we are assured the cost is always cheapest. □

Time Complexity: Creating and intializing the list costs $\mathcal{O}(n)$. Sorting the list costs $\mathcal{O}(nlogn)$. Traverse the sorted list costs $\mathcal{O}(n)$. So the overall complexity of this algorithm is $O(nlogn)$

6.2 Solution M

Given n servers with their copying cost, we want to find the optimal number of servers and the optimum combination of servers such that our cost function is minimized.
We will use the concept of dynamic programming similar to how we used in the previous question.
For any server $S_i \in S_1, \ldots, S_n$, we will find the optimal soultion for sub-list of servers of S_i to S_n **with S_i included** using already found optimal solution at a server later than i and before n.
Here is how it works:
for $i \in (1, n)$ after having chosen S_i let us suppose in the optimal solution for S_i, the next server chosen is S_k where $k \in (i+1, n)$ and it is given that we know optimal solutions for all k.
So, optimum cost function will be:

$opt\_cost(i) = C_i + \min_{\forall k \in (i+1,n)}(\frac{(k-i)(k-i-1)}{2} + opt\_Cost(k))$

Once, we have found $opt\_cost(i) \forall i \in (1, n)$ we can traverse the whole opt_cost array to find the minimum cost.
Algorithm: Consider two additional arrays , apart from S and C:
minimum cost array opt_cost : length n with all elements initialized to ∞ and where
opt_cost(i) gives us the minimium cost of storing the file on S_i to S_n servers with S_i chosen.
and optimal combination $opt\_servers$ array
where $opt\_servers$(i) gives us the combination of selected servers for optimal solution including S_i.
Now,

1. For i = n to 1:
2. **Base case**
   if i==n: opt_cost(i) = $c_n = 0$ end if
   else: **Dynamic case:**
   for j = i+1 to n:
   min = ∞
   $min\_index = j$
   if ($\frac{(j-i)(j-i-1)}{2}$+opt_Cost(j)) < min :
   min = ($\frac{(j-i)(j-i-1)}{2} + opt\_Cost(j)$)
   min_index = j
   end if
   end for

$opt\_Cost(i) = (\frac{(min\_index-i)(min\_index-i-1)}{2} + opt\_Cost(min\_index))$
$opt\_servers(i) = S_i$ + $opt\_servers(min\_index)$ (**append strings**)
end else

3. Once we are done with the iterations, traverse the whole array *opt_cost* to find the index(k) of the min cost.

4. output opt_cost(k) and opt_servers(k)

Complexity Analysis:
determining entire array of *opt_cost* using looping on i and j: O(n^2)
traversing whole array to find the minimum cost: O(n)
So time complexity = O(n^2) + O(n) = O(n^2)