Solutions - Problem Set #4

Alice Bob (uni). Collaborators: A. Turing. - uni@columbia.edu

October 26, 2019

1 Problem 1

Idea

In order to minimize the number of intervals needed, we should place intervals as sparse as possible. Also, every interval should cover as many points as possible.

Given a set of points (call it X), the first thing we need to do is to sort all points in X into ascending order. Next we create an interval $I = [min(X), min(X)+1]$. We say that a point x_i is covered by I if and only if $x_i \geq min(X)$ and $x_i \leq min(X) + 1$. We no longer need to consider points that have already been covered by I since we want our intervals to be sparse, so we can just remove them and update X. We continue adding new intervals until there is no element in X.

Pseudocode

```
function findIntervals(X)
    X = quicksort(X)
    result = []
    index = 1
    while (index <= X.length) do
        start = X[index]
        result.append(Interval(start, start + 1))
        while (index <= X.length and X[index] <= start + 1) do
            index = index + 1
    return result
```

Correctness

In order to prove the correctness of a greedy algorithm, we should prove two things: (from lec11)

- Exchange argument: there exists some optimal solution that includes the first greedy choice.

- Optimal substructure property: given the first greedy choice, an optimal solution for the remainder of the problem leads to an overall optimal solution.

First we prove the exchange argument. Suppose S^* is an optimal solution and the first interval (the interval with minimum start point) in S^* is $[a, a+1]$. Since S^* is a valid solution, we must have that $a \leq min(X)$. Since there is no point in X that is less than $min(X)$, it is safe to shift $[a, a+1]$ to be $[min(X), min(X)+1]$ (our first greedy choice) because all points covered by $[a, a+1]$ can be covered by $[min(X), min(X)+1]$. This operation does not introduce any new interval, thus replacing $[a, a+1]$ by $[min(X), min(X)+1]$ yields another optimal solution. We have proved that we can always find some optimal solution that includes our first greedy choice.

Second we prove the optimal substructure property. Suppose X is the set of all points and Y is the set of all points that are not covered by the first interval for X (call it I_1). Suppose S^*_{sub} is an optimal solution for Y.

We can always prove the optimal subproblem property by contradiction. Suppose S_{sub} is a valid but non-optimal solution for Y (the size of S_{sub} is greater than S^*_{sub}) and $S_{sub} \cup I_1$ is an optimal solution for X.

Since the size of S_{sub} is greater than S^*_{sub}, the size of $S_{sub} \cup I_1$ must be greater than the size of $S^*_{sub} \cup I_1$, which contradicts to the fact that $S_{sub} \cup I_1$ is an optimal solution (valid and containing minimum number of intervals). Thus we have shown that we can never find an optimal solution for X by combining our first greedy choice and a non-optimal solution for Y.

By combining the exchange argument and optimal substructure property, we can prove the correctness of our algorithm.

Time Complexity

- We can sort all points in X into ascending order in $O(n \log n)$. For example, use heapsort or quicksort.
- We iterate over all points only once and create at most n intervals, thus the running time for creating intervals for sorted X is $O(n)$.
- Total running time: $O(n \log n) + O(n) = O(n \log n)$.

2 Problem 2

In this problem, we have n activities $a_1, \dots, a_n$, where each a_i is associated with a starting time, a finish time, and a positive weight: $a_i = (s_i, t_i, w_i) \in [0, \infty) \times [0, \infty) \times (0, \infty)$. Recall that two tasks a_i, a_j are *non-overlapping* if $t_i \leq s_j$ or $t_j \leq s_i$.

The goal is to select a subset $Q \subseteq \{a_1, \dots, a_n\}$ of non-overlapping tasks maximizing the total weight $w(Q) \stackrel{\text{def}}{=} \sum_{i:a_i \in Q} w_i$. (As opposed to the standard, unweighted version of this problem, which asked to maximize $|Q|$.)

(a) Counterexamples to greedy algorithms.

Select the largest weight: Consider the set of activities defined by (i) $a_n = (0, n, 2)$ and (ii) $a_i = (i-1, i, 1)$ for $1 \leq i \leq n-1$ (that is, one very long activity with weight 2, and many non-overlapping activities each with weight 1).

By this rule, the greedy algorithm will select the task a_n; and then stop, as it overlaps with every other activity. Thus, its solution will be $Q = \{a_n\}$, which has weight $w(Q) = 2$. However, the optimal solution is $Q^* = \{a_1, \dots, a_{n-1}\}$, which has weight $w(Q^*) = n - 1$.

Select the earliest finish time: Consider now the set of activities defined by (i) $a_n = (0, n, 2n)$ and (ii) $a_i = (i-1, i, 1)$ for $1 \leq i \leq n-1$ (that is, one very long activity with *very* big weight but very late finish time, and many non-overlapping activities each with weight 1).

By this rule, the greedy algorithm will select the task a_1, then a_2, etc.; and then stop after selecting a_{n-1}, as a_n now overlaps with the previously selected activities. Thus, its solution will be $Q = \{a_1, \dots, a_{n-1}\}$, which has weight $w(Q) = n - 1$. However, the optimal solution is $Q^* = \{a_n\}$, which has weight $w(Q^*) = 2n$.

(b) We assume for this question that the activities are sorted by earliest finish time, that is that $t_1 \leq t_2 \leq \cdots \leq t_n$. Defining

$$W(i) \stackrel{\text{def}}{=} \max_{\substack{Q \subseteq \{a_1, \dots, a_i\} \\ Q \text{ non overlapping}}} w(Q)$$

for any $i \in [n]$, we need to compute, for any given $i \in [n]$, the value of $W(i)$ using the values $W(1), \dots, W(i-1)$. (As a first obvious remark, note that $W(0) \leq W(1) \leq W(2) \leq \cdots \leq W(n)$.)

Define $W(0) = 0$ and $s_0 = t_0 = 0$ for convenience, and given $i \in [n]$ let j^* denote the biggest index $j \leq i$ such that a_j and a_i are non-overlapping; i.e., because the activities are sorted,

$$j^* \stackrel{\text{def}}{=} \min\{0 \leq j \leq i-1 : t_j \leq s_i\}.$$

To see why this helps, consider an optimal set $Q_i^* \subseteq \{a_1, \dots, a_i\}$ of non-overlapping activities among the first i, that is such that $W(i) = w(Q_i^*)$. There are now two options:

- $a_i \notin Q_i^*$, in which case $Q_i^* \subseteq \{a_1, \dots, a_{i-1}\}$ from which $W(i) = w(Q_i^*) \leq W(i-1)$ and therefore $W(i) = W(i-1)$.
- $a_i \in Q_i^*$: in this case, besides a_i the set Q_i^* can only contain activities a_j with $j \leq j^*$, since by definition of j^* the other ones are overlapping with a_i. It follows that, in this case, we have $W(i) = W(j^*) + w_i$.

Combining the two cases, we get that

$$W(i) = \max(W(i-1), W(j^*) + w_i). \tag{1}$$

Moreover, we remark that computing j^* given i takes $O(\log i)$ (as the activities are sorted by t_j) time, and thus so does computing $W(i)$ by the above formula.

(c) The previous question suggests a natural dynamic programming (DP) approach to compute the optimal value, which will be $W(n)$ – with the subtlety that are also asked to return a set $Q \subseteq \{a_1, \dots, a_n\}$ of non-overlapping activities achieving this optimum.

This can be easily achieved as follows: we first sort the activities by earliest finish time, to be able to use the previous question (this takes time $O(n \log n)$. Assuming henceforth that the list of activities is sorted, we will maintain a list of pairs $((W(i), Q_i))_{0 \leq i \leq n}$, where $W(i)$ is as in (b) and $Q_i \subseteq \{a_1, \dots, a_i\}$ will be a set achieving this value.

We initialize $W(0) = 0$ and $Q_0 = \emptyset$; from there, we loop from 1 to n (in this order), computing at step i the pair $(W(i), Q_i)$ based on the precomputed pairs $(W(j), Q_j)$ for $0 \leq j \leq i-1$, as per Eq. (1). More specifically, at step i we:

(a) compute $j^* \leftarrow \min\{0 \leq j \leq i-1 : t_j \leq s_i\}$;

(b) set $W(i) \leftarrow \max(W(i-1), W(j^*) + w_i)$;

(c) depending on which of the two terms achieved the maximum, we set Q_i to be either Q_{i-1} or $Q_{j^*} \cup \{a_i\}$.

The full algorithm is given in Figure 1.

Its correctness is immediate based on (b); as for the time complexity, it is

$$O(n \log n) + \sum_{i=1}^{n} O(i) = O(n \log n) + O(n^2) = O(n^2).$$

Algorithm 1 Dynamic programming approach

Require: set of activities $\{a_1, \ldots, a_n\}$

1: sort the activities by earliest finish time ▷ $O(n \log n)$ time
2: Set $(W(0), Q(0) \leftarrow (0, \emptyset)$
3: **for** $i = 1$ **to** n **do**
4:     Compute $j^* \leftarrow \min\{0 \leq j \leq i-1 : t_j \leq s_i\}$ ▷ $O(\log i)$ time
5:     **if** $W(i-1) \geq W(j^*) + w_i$ **then**
6:         Set $(W(i), Q(i)) \leftarrow (W(i-1), Q(i-1))$ ▷ Assigning $Q(i)$: time $O(i)$
7:     **else**
8:         Set $(W(i), Q(i)) \leftarrow (W(j^*) + w_i, Q(j^*) \cup \{a_i\})$ ▷ Assigning $Q(i)$: time $O(i)$
9:     **end if**
10: **end for**
11: **return** $Q(n)$.

Improving the time complexity. Note that this can be improved to $O(n \log n)$ as follows: instead of storing the full set $Q(i)$ for each i, one can store a pair (p, b) where p is the value of the previous index, and $b \in \{0, 1\}$ indicates which of the two cases holds when taking the maximum in Steps 5–8. That is, with the above notations, at step i the pair (p, b) will be set to either $(j^*, 0)$ or $(i-1, 1)$ (that is b indicates whether a_i belongs to the optimal set or not, and p is the index of the previous $Q(j)$ to consider). In this case, the total time complexity will be $O(n \log n) + \sum_{i=1}^{n} O(\log i) = O(n \log n) + O(n \log n) = O(n \log n)$, and to return the final set $Q(n)$ it is sufficient to "follow" the path indicated by the pairs, adding the elements to the set as we go.

3 Problem 3

Naive Method

Let $\mathcal{F}$ be the set of all functions $f\colon [n] \mapsto [m]$.
Since there are m possible values for each $f(i)$, $|\mathcal{F}| = m^n$.
Time to calculate cost for a mapping (and check it is non-decreasing): $O(n)$.
Hence the naïve method has complexity of $O(n \cdot m^n)$

Part B

Calculating, $C(i, j)$ using $C(s, t)$ where $s < i, t \leq j$.

- Let's assume in optimal mapping $C(i, j)$, index i maps to index k_0 in second series, i.e. $f(i) = k_0$.
- In the optimal mapping, any index $i' \in [1, i-1]$ can only map to some value in $[1, k_0]$ since $f(i_1) \leq f(i_0)$ whenever $i_1 < i_0$.
- This implies
$$C(i, j) = |a_i - b_{k_0}| + C(i-1, k_0)$$
.
- We can calculate k_0 and $C(i, j)$ as follows,
$$C(i, j) = \min_{1 \leq k \leq j} \{|a_i - b_k| + C(i-1, k)\}$$

Part C

We can design a simple Dynamic programming solution to solve the above problem.

```
Matrix = n * m matrix to store optimal cost
// Matrix[i][j] = C(i,j)

// Initialize Matrix
cost = |A[1] - B[1]|
for j in {1..n}
    cost = min(cost, |A[1] - B[j]|)
    Matrix[1][j] = cost

cost = 0
for i in {1..n}
    cost = cost + |A[i] - B[1]|
    Matrix[i][1] = cost
```

```
function calculateCost(i, j):
    if Matrix[i][j] exists:
        return Matrix[i][j]

    value = Int.MAX
    for k in {1..j}
        sub_cost = calculateCost(i-1, k)
        value = min {value, |A[i] - B[k]| + sub_cost)

    Matrix[i][j] = value
    return value

function main:
    return calculateCost(n,m)
```

The value of Matrix[n][m] is our final answer. The correctness of this algorithm is proved in Part B.

Running time. Since it takes at most $O(m)$ time to update single cell of a $n \times m$ matrix. The total runtime is $O(nm^2)$.

Bonus

Alternate method to update $C(i, j)$ is,

$$C(i, j) = \min\{C(i, j-1), |a_i - b_j| + C(i-1, j)\}$$

The Runtime in that case would be $O(nm)$ as it takes $O(1)$ time to update each cell now.

Part D

To output the final mapping we can store an additional $n * m$ matrix and update it while calculating the optimal cost in Part C. Update the calculateCost function in section C to update the mapping matrix as follows,

$$\text{mapping}(i, j) = k_0$$

where

$$C(i, j) = |a_i - b_{k_0}| + C(i-1, k_0)$$

This can be done while calculating optimal cost.

Since in optimal mapping if

$$f(i) = k_0 \implies \forall i' < i; f(i') \leq k_0$$

That is, $f(i-1) \in [1, k_0]$ where $f(i) = k_0$. Hence to find $f(i-1)$ given $f(i) = k_0$ we only need to check mapping$(i-1, k_0)$ as $f(i-1) \in [1, k_0]$.

Therefore we can generate the final mapping output as follows,

```
function generateMapping():
    k_0 = m
    for i in {n..1}:
        k_0 = Mapping[i][k_0]
        print i, k_0
```

Run Time: $O(n)$.

Since we can update Mapping matrix while calculating the $C(i, j)$ without increasing the complexity. The asymptotic run time of the algorithm remains the same.

4 Problem 4

(a) Search each of the k arrays using binary search. The time to search array A_i is $O(i)$ so the total time it takes is $\sum_{i=1}^{k} i = \frac{k(k+1)}{2} = O(k^2) = O(\lg^2 n)$

(b) When we insert a new element x, we add 1 to the binary representation of n. We locate the first n_i that is 0 and then merge $A_{i-1}, \ldots, A_1$ together with x and call it A_i. We can find the first 0 in $O(\lg n)$ time by scanning from the beginning. The worst case for merging is we have to merge $A_1, \ldots, A_k$. We first merge A_1 with A_2, and then merge that with A_3, etc. The time to merge $A_1, \ldots, A_k$ would take $\sum_{i=0}^{k-1} 2^i = O(2^k) = O(n)$. Thus, for insert, the worst-case time is $O(n)$.

To analyze the amortized time, recall that when inserting n numbers, the number of times n_i flips is $\lfloor \frac{n}{2^i} \rfloor$ and whenever it flips from 1 to 0, it takes $O(2^i)$ time to merge A_i with the merged list of $A_{i-1}, \ldots, A_1$. Thus, the contribution of A_i to the runtime is $O(n)$ so the total contribution of all $A_1, \ldots, A_k$ is $O(n \lg n)$. Hence, we get a $O(\lg n)$ amortized cost per operation.

(c) To implement DELETE, we first run SEARCH to find our element x. Next, we find the first instance of 1 in our binary representation, suppose this is at n_i. We need to change n_i to 0 and change $n_{i-1}, \ldots, n_0$ to 1, meaning we need to take A_i, remove an element y (say the top element) and then divide the rest into $A_{i-1}, \ldots, A_0$. Next, we move y to the array that x came from.

Finding the smallest nonzero n_i takes $O(\lg n)$. SEARCH takes $O(\lg^2 n)$. Putting y into the array that x belongs takes $O(n)$ time and dividing A_i into $A_{i-1}, \ldots, A_0$ takes $O(n)$ time, so in total we have $O(n)$ worst case running time.