CSOR W4231: Analysis of Algorithms (sec. 001) - Problem Set #4

Hongmin Zhu (hz2637) - hz2637@columbia.edu

October 30, 2019

Problem 1

Algorithm:

We apply *greedy* approach to this problem. First, sort the given set of points X in ascending order, and then the algorithm starts to select the smallest point x and builds an unit-length interval $[x_s, x_s + 1]$ from it. All the points inside $[x_s, x_s + 1]$ will be removed from X. The algorithm continues to select the smallest point and build an interval repeatedly until the set X is empty.

Pseudo-code:

INTERVALS (X)
$S \leftarrow emptySet$
$sort(X)$
while X is not empty do
  $x_s \leftarrow X[0]$
  $interval \leftarrow$ all points inside $[x_s, x_s + 1]$
  add $interval$ to S
  remove all points inside $[x_s, x_s + 1]$ from X
return S

Analysis:

Correctness. The set X is sorted in ascending order. We know $[x_1, x_1 + 1]$ is the interval starting from the smallest point in X, suppose the optimal set S_{opt} contains an interval $[p, p + 1]$ that covers point x_1, so $p \leq x_1 \leq x_1 + 1$. As x_x is the leftmost point of X, we know that there are no points in $\big[p, x_1\big)$. Therefore, we can simply replace $[p, p + 1]$ with $[x_1, x_1 + 1]$, which means $[x_1, x_1 + 1]$ itself is an optimal interval.

The optimal solution for sub-problem can be built by solving the problem with all points in $[x_1, x_1 + 1]$ removed from X. The *greedy* approach solves the sub-problem by an identical way and yields the optimal solution S'_{opt}, so overall we get solution $S' \cup [x_1, x_1 + 1]$, which is the optimal one for the entire problem.

Time Complexity. sort takes $O(nlogn)$ time, and the *while-loop* goes over every element in X, so it takes $O(n)$ time. The overall running time is $O(nlogn)$.

Problem 2

(a)
(i) counterexample:
$a_i : (1, 9), (7, 11), (10, 15)$
$w_i : 10, \quad 20, \quad 15$
if we select an activity of largest weight, we would choose $Q : (7, 11)$ which gives us the total weight $w(Q) = 20$. However, the maximum weight should be $w(Q) = 25$ with $Q : (1, 9), (10, 15)$.

(ii) counterexample:
$a_i : (1, 9), (7, 11), (10, 15)$
$w_i : 10, \quad 20, \quad 5$
if we select an activity of earliest finishing time, we would choose $Q : (1, 9), (10, 15)$ which gives us the total weight $w(Q) = 15$. However, the maximum weight should be $w(Q) = 20$ with $Q : (7, 11)$.

(b)
From question (a), we can see that *greedy* solution can fail with either considering the largest weight or the earliest finishing time. We can then consider using *dynamic programming*.

The activities are already sorted by finishing time, suppose *compatible(i)* means the largest index $j < i$ such that activity j is compatible with i. So we have the following two cases:

(1) Case 1: we select current activity i:
-then we can't choose activities from $compatible(i) + 1, compatible(i) + 2, ..., i - 1$ as they are all incompatible with current activity i
-the optimal solution must include optimal solution to the problem consisting of compatible activities $1, 2, 3..., compatible(i)$.

(2) Case 2: we don't select current activity i:
-then the optimal solution must include optimal solution to the problem consisting of compatible activities $1, 2, 3..., i - 1$.

So, we have:

$$W(i) = \begin{cases} 0 & i = 0 \\ \min\left(W(compatible(i)) + w_i, W(i-1)\right) & otherwise \end{cases}$$

(c)

Algorithm:

Use *Bottom-Up dynamic programming* to solve this problem. Create an array to store the optimal solution, i.e. the largest total weight of each sub-problem and use a *for-loop* to calculate the current optimal solution.

Pseudo-code:

```
findCompatible(A, i)
if i == 1
   if A[i].start ≥ A[0].finish
      return 0
   return -1
l ← 0
r ← i − 1
while l ≤ r do
   mid ← (l + r)/2
   if A[i].start == A[mid].finish
      return mid
   else if A[i].start < A[mid].finish
      r ← mid − 1
   else
      l ← mid + 1
return r
```

```
maxWeight (A)
sort(A) by finishing time
optimal[0] ← 0
for i ← 1 to n
   compatible(i) ← findCompatible(A, i)
   optimal[i] ← max(optimal[compatible[i]] + A[i].w, optimal[i − 1])
return optimal[n]
```

```
findSubset (optimal, A, result, i)
if i == 0
   return
if optimal[compatible[i]] + A[i].w > optimal[i − 1]
   result.add(A[i])
   findSubset(optimal, A, result, compatible(i))
else
   findSubset(optimal, A, result, i-1)
```

Analysis:

The algorithm uses *dynamic programming* to solve this optimization problem. That input activities are sorted by the finishing time and the function MAXWEIGHT will use a *for-loop* to go through the input. For current activity i, we use FINDCOMPATIBLE to get the largest index $j < i$ such that activity j is compatible with i and consider two cases:

(1) Case 1: we select current activity i:
-then we can't choose activities from $compatible(i) + 1, compatible(i) + 2, ..., i - 1$ as they are all incompatible with current activity i
-the optimal solution must include optimal solution to the problem consisting of compatible activities $1, 2, 3..., compatible(i)$.

(2) Case 2: we don't select current activity i:
-then the optimal solution must include optimal solution to the problem consisting of compatible activities $1, 2, 3..., i - 1$.

and we take the maximum of these two cases,

$$W(i) = \begin{cases} 0 & i = 0 \\ \min\left(W(compatible(i)) + w_i, W(i-1)\right) & otherwise \end{cases}$$

so that the overall optimal solution consists of optimal solutions for sub-problems, which makes this *dynamic programming* algorithm correct. The function FINDSUBSET outputs the subset.

Running time. The function FINDCOMPATIBLE uses *binary search* to find the index, so the time complexity is $O(log(n))$. The function MAXWEIGHT sorts the input in $O(nlog(n))$ time, and the *for-loop* takes $O(nlog(n))$, so the overall running time is $O(log(n))$. The recursive calls in FINDSUBSET outputs the subset, so its running time is $O(n)$.

Problem 3

(a)
Pseudo-code:

```
computeDistance(A, B, idx, distance, f)
if (idx > n)
   cost ← 0
   for i ← 1 to n
      cost ← cost + abs(A[i] − B[f[i]])
   distance ← min(distance, cost)
else
   for j ← 1 to m
      if (j ≥ f[idx − 1])
         f[idx] ← j
         computeDistance(A, B, idx + 1, distance, f)
```

(b)
For $C(i, j)$, the problem we should consider is there are two integers a_i, b_j, whether we match these two or not, all the previous integers are already be perfectly matched. So for a_i, b_j, there are two cases:

(1) we match a_i, b_j, then we can know from the definition of f that a_{i-1} can also choose to match b_j, so the sub-problem here is $C(i-1, j)$, $C(i, j) = |a_i - b_j| + C(i-1, j)$.
(2) we don't match a_i, b_j, therefore, $a_k, k < i$ cannot match b_j either, the sub-problem here is $C(i, j-1)$, $C(i, j) = C(i, j-1)$.

So $C(i, j) = \min\left(|a_i - b_j| + C(i-1, j), C(i, j-1)\right)$.

(c)
Pseudo-code:

```
computeDistance(A, B
grid ← n ∗ m matrix
grid[1][1] ← |A[1] − B[1]|
for col ← 2 to m do
   grid[1][col] ← min(|A[0] − B[col]|, grid[0][col − 1])
for row ← 2 to n do
   grid[row][1] ← grid[row − 1][1] + |grid[row][1]|
for row ← 3 to n do
   for col ← 3 to m do
      grid[row][col] ←
            min(|A[row] − B[col]| + grid[row − 1][col], grid[row][col − 1])
```

return $grid[n][m]$

Analysis:

The algorithm uses *dynamic programming* to calculate the distance, the initialization takes $O(n+m)$ time, and the calculation takes two *for-loop* in $O(nm)$ time, so the total running time is $O(nm)$.

Problem 4

(1)
Pseudo-code:

SEARCH(A, *target*)
for $i \leftarrow 0$ to $k-1$ do
  $pos \leftarrow binarysearch(A_i, target)$
  if $pos \neq 0$ do
    return (i, pos)
return $None$

Analysis:

We linearly go through each array A_i, and use *binary search* to search it, if current array contains target value, return index, otherwise we continue to search next array. So the worst case is we need to binary search all the array A_i.

Running time. We know that array A_i has length of 2^i, binary searching this array will take $O(log(2^i))$ time, which is $O(i)$ time. And i ranges from 0 to $k-1$, and $k = \lceil log(n+1) \rceil$, so in total, the worst case running time is $\sum_{i=0}^{k-1} O(i)$, which is $O(log^2(n))$.

(2)
Pseudo-code:

INSERT(A, *target*)
$B[0] \leftarrow target$
for $i \leftarrow 0$ to $k-1$ do
  if $A[i]$ is full do
    $B[i+1] \leftarrow combine(A[i], B[i])$
    empty $A[i]$
  else do
    $A[i] \leftarrow B[i]$
    return
$A[k] \leftarrow B[k]$

Analysis:

To insert a new element, the algorithm creates a new Array A_0 with size of 1. If the original A_0 of the data structure is already full, then we combine these two $A_[0$ into one array A_1. If the original A_1 of the data structure is already full, we combine two A_1 into array A_2. The algorithm repeats this procedure until the combination is no longer needed. We know that combine two sorted array into a bigger array can be done linearly in the total length of lists, so assume the algorithm combines arrays $A_0, A_1, ..., A_{m-1}$ into A_m, the running time is $O(2^m)$, the worst case is the algorithm needs to combine all the arrays $A_0, A_1, ..., A_{k-1}$ into A_k, so the worst case running time is $\sum_{i=0}^{k-1} 2^i = O(2^k) = O(n)$.

Amortized time. From the binary representation of $n, < n_{k-1}, n_{k-2}, ..., n_0 >$, we can know that every time the algorithm combine two arrays into one bigger array, n_i flips. To be specific, n_0 flips every time, n_1 flips every $2th$ time,...,n_{k-1} flips every $2^k th$ time. So for total running time for x insert operation is: $T \leq \sum_{i=0}^{k-1} \lfloor \frac{x}{2^i} \rfloor 2^i \leq xk = xO(k) = xO(logn)$, so the amortized running time for each operation is $xO(logn)/x = O(logn)$.

(3)
Pseudo-code:

DELETE(A, $target$)
for $i \leftarrow 0$ to $k-1$ do
  if $A[i]$ is not empty do
    $A_s \leftarrow A[i]$
    break
$i, pos \leftarrow$ SEARCH(A, $target$)
remove the target of $A[i][pos]$
get a value from A_s and insert this value to the right place of $A[i]$
break down A_s to several smaller arrays

Analysis:

The algorithm finds the first array A_s that is not empty with smallest index, which takes $O(k) = O(logn)$ time in worst case, then it uses SEARCH to find the right array with target value, which takes $O(log^2(n))$ in worst case. We delete the target value, and swap a value from S and insert it to the right place, since we need to loop over this array to find the right place, the worst case is that this array is A_{k-1}, which takes $O(k) = O(logn)$ time with binary search, finally we break down array A_s to several smaller arrays with time of $O(2^s)$ which is $O(logn)$ time in worst case. So in the worst case, the running time is $O(log^2(n)) + 3O(logn)$, which is $O(log^2(n))$.

Amortized time. empty.