

CSOR 4231 HW 4

Hongmin Zhu

TOTAL POINTS

98 / 100

QUESTION 1

11 20 / 20

✓ - **0 pts** Correct

- **5 pts** Almost all correct
- **10 pts** Partially correct
- **15 pts** Only some part correct
- **20 pts** Wrong
- **4 pts** Page not selected

QUESTION 2

2 30 pts

2.1 30 / 30

✓ - **0 pts** Correct

- **5 pts** (a)(i) incorrect
- **5 pts** (a)(ii) incorrect
- **5 pts** (b) partially correct
- **10 pts** (b) incorrect
- **2.5 pts** (c) wrong time complexity
- **2.5 pts** (c) no correctness explanation
- **4 pts** (c) algorithm not fully correct (e.g. not returning the subset)
- **10 pts** (c) incorrect

QUESTION 3

3 3 35 / 30

✓ - **0 pts** Correct

✓ + **5 pts** Bonus

- **6 pts** (a) incorrect
- **1 pts** (a) small error
- **4 pts** (a) insufficient
- **6 pts** (b) incorrect
- **2 pts** (b) small error
- **6 pts** (c) incorrect
- **1 pts** (c) small error

- **3 pts** (c) insufficient/incorrect analysis

- **8 pts** (a) empty

- **8 pts** (b) empty

- **8 pts** (c) empty

- **10 pts** (a) nonsense

- **10 pts** (b) nonsense

- **10 pts** (c) nonsense

- **2 pts** pages not selected

QUESTION 4

4 20 pts

4.1 13 / 20

✓ + **6 pts** Correct

- + **3 pts** Correct search
- + **3 pts** Correct complexity $\lg^2 n$
- + **0 pts** Wrong

✓ + **7 pts** B correct

- + **7 pts** C correct
- + **2 pts** B description correct or partially correct
- + **2 pts** B worse case correct
- + **3 pts** B amortized correct
- + **2 pts** C description correct or partially correct
- + **2 pts** C worse case correct
- + **3 pts** C amortized correct
- + **1.4 pts** empty for 4(c)
- + **2 pts** explain c correctly, but did not give the upper bound

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 ( $X$ )
1   $S \leftarrow \text{emptySet}$ 
2   $\text{sort}(X)$ 
3  while  $X$  is not empty do
4     $x_s \leftarrow X[0]$ 
5     $\text{interval} \leftarrow$  all points inside  $[x_s, x_s + 1]$ 
6    add  $\text{interval}$  to  $S$ 
7    remove all points inside  $[x_s, x_s + 1]$  from  $X$ 
8  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 $p \leq x_1 \leq p + 1$. As x_1 is the leftmost point of X , we know that there are no points in $[p, x_1)$. 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 $O(n \log n)$ time, and the *while-loop* goes over every element in X , so it takes $O(n)$ time. The overall running time is $O(n \log n)$.

11 20 / 20

✓ - **0 pts** Correct

- **5 pts** Almost all correct

- **10 pts** Partially correct

- **15 pts** Only some part correct

- **20 pts** Wrong

- **4 pts** Page not selected

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 $compatible(i) + 1, compatible(i) + 2, \dots, i - 1$ as they are all incompatible with current activity i

-the optimal solution must include optimal solution to the problem consisting of compatible activities $1, 2, 3, \dots, compatible(i)$.

(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 $1, 2, 3, \dots, i - 1$.

So, we have:

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

(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)

```

1  if  $i == 1$ 
2    if  $A[i].start \geq A[0].finish$ 
3      return 0
4    return -1
5   $l \leftarrow 0$ 
6   $r \leftarrow i - 1$ 
7  while  $l \leq r$  do
8     $mid \leftarrow (l + r)/2$ 
9    if  $A[i].start == A[mid].finish$ 
10     return  $mid$ 
11   else if  $A[i].start < A[mid].finish$ 
12      $r \leftarrow mid - 1$ 
13   else
14      $l \leftarrow mid + 1$ 
15  return  $r$ 

```

MAXWEIGHT (A)

```

1  sort( $A$ ) by finishing time
2   $optimal[0] \leftarrow 0$ 
3  for  $i \leftarrow 1$  to  $n$ 
4     $compatible(i) \leftarrow \text{FINDCOMPATIBLE}(A, i)$ 
5     $optimal[i] \leftarrow \max(optimal[compatible[i]] + A[i].w, optimal[i - 1])$ 
6  return  $optimal[n]$ 

```

FINDSUBSET ($optimal, A, result, i$)

```

1  if  $i == 0$ 
2    return
3  if  $optimal[compatible[i]] + A[i].w > optimal[i - 1]$ 
4     $result.add(A[i])$ 
5    FINDSUBSET( $optimal, A, result, compatible(i)$ )
6  else
7    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 $compatible(i) + 1, compatible(i) + 2, \dots, i - 1$ as they are all incompatible with current activity i

-the optimal solution must include optimal solution to the problem consisting of compatible activities $1, 2, 3, \dots, compatible(i)$.

(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 $1, 2, 3, \dots, i - 1$.

and we take the maximum of these two cases,

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

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 $O(n \log(n))$ time, and the *for-loop* takes $O(n \log(n))$, so the overall running time is $O(n \log(n))$. The recursive calls in FINDSUBSET outputs the subset, so its running time is $O(n)$.

2.1 30 / 30

✓ - **0 pts** Correct

- **5 pts** (a)(i) incorrect
- **5 pts** (a)(ii) incorrect
- **5 pts** (b) partially correct
- **10 pts** (b) incorrect
- **2.5 pts** (c) wrong time complexity
- **2.5 pts** (c) no correctness explanation
- **4 pts** (c) algorithm not fully correct (e.g. not returning the subset)
- **10 pts** (c) incorrect

Problem 3

(a)

Pseudo-code:

```

COMPUTEDISTANCE(A, B, idx, distance, f)
1  if (idx > n)
2      cost ← 0
3      for i ← 1 to n
4          cost ← cost + abs(A[i] − B[f[i]])
5      distance ← min(distance, cost)
6  else
7      for j ← 1 to m
9          if (j ≥ f[idx − 1])
10             f[idx] ← j
11             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 $C(i, j) = \min(|a_i - b_j| + C(i-1, j), C(i, j-1))$.

(c)

Pseudo-code:

```

COMPUTEDISTANCE(A, B
1  grid ← n * m matrix
2  grid[1][1] ← |A[1] − B[1]|
3  for col ← 2 to m do
4      grid[1][col] ← min(|A[0] − B[col] |, grid[0][col − 1])
5  for row ← 2 to n do
6      grid[row][1] ← grid[row − 1][1] + |grid[row][1]|
7  for row ← 3 to n do
8      for col ← 3 to m do
9          grid[row][col] ←
              min(|A[row] − B[col] | + grid[row − 1][col], grid[row][col − 1])

```

```
10 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 $O(nm)$.

33 35 / 30

✓ - 0 pts Correct

✓ + 5 pts Bonus

- 6 pts (a) incorrect
- 1 pts (a) small error
- 4 pts (a) insufficient
- 6 pts (b) incorrect
- 2 pts (b) small error
- 6 pts (c) incorrect
- 1 pts (c) small error
- 3 pts (c) insufficient/incorrect analysis
- 8 pts (a) empty
- 8 pts (b) empty
- 8 pts (c) empty
- 10 pts (a) nonsense
- 10 pts (b) nonsense
- 10 pts (c) nonsense
- 2 pts pages not selected

Problem 4

(1)

Pseudo-code:

```

SEARCH( $A$ ,  $target$ )
1   for  $i \leftarrow 0$  to  $k - 1$  do
2        $pos \leftarrow binarysearch(A_i, target)$ 
3       if  $pos \neq 0$  do
4           return  $(i, pos)$ 
5   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( $A$ ,  $target$ )
1    $B[0] \leftarrow target$ 
2   for  $i \leftarrow 0$  to  $k - 1$  do
3       if  $A[i]$  is full do
4            $B[i + 1] \leftarrow combine(A[i], B[i])$ 
5           empty  $A[i]$ 
6       else do
7            $A[i] \leftarrow B[i]$ 
8       return
9    $A[k] \leftarrow B[k]$ 

```

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, \dots, 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, \dots, 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 $n, \langle n_{k-1}, n_{k-2}, \dots, n_0 \rangle$, 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: $T \leq \sum_{i=0}^{k-1} \lfloor \frac{x}{2^i} \rfloor 2^i \leq xk = xO(k) = xO(\log n)$, so the amortized running time for each operation is $xO(\log n)/x = O(\log n)$.

(3)

Pseudo-code:

```

DELETE( $A$ ,  $target$ )
1  for  $i \leftarrow 0$  to  $k - 1$  do
2    if  $A[i]$  is not empty do
3       $A_s \leftarrow A[i]$ 
4      break
5   $i, pos \leftarrow \text{SEARCH}(A, target)$ 
6  remove the target of  $A[i][pos]$ 
7  get a value from  $A_s$  and insert this value to the right place of  $A[i]$ 
8  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 $O(k) = O(\log n)$ 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 $O(k) = O(\log n)$ time with binary search, finally we break down array A_s to several smaller arrays with time of $O(2^s)$ which is $O(\log n)$ time in worst case. So in the worst case, the running time is $O(\log^2(n)) + 3O(\log n)$, which is $O(\log^2(n))$.

Amortized time. empty.

4.1 13 / 20

✓ + 6 pts Correct

+ 3 pts Correct search

+ 3 pts Correct complexity $\lg^2 n$

+ 0 pts Wrong

✓ + 7 pts B correct

+ 7 pts C correct

+ 2 pts B description correct or partially correct

+ 2 pts B worse case correct

+ 3 pts B amortized correct

+ 2 pts C description correct or partially correct

+ 2 pts C worse case correct

+ 3 pts C amortized correct

+ 1.4 pts empty for 4(c)

+ 2 pts explain c correctly, but did not give the upper bound