Sum Yi Li
(UID: 505146702)

Discussion 1A
10:00am - 11:50am
Goldstein Orpaz

① Exercise 13, page 194

- The efficient algorithm is to arrange the job in decreasing order of w_i/t_i. which is the weight of the job divided by the time it takes to complete.
- Let assume there is some other schedule. Our original greedy algorithm schedule is job j scheduled before job i
- The other solution is job i scheduled before job j for any pairs of job i, j.
- According to our greedy algorithm, job j's ration w_j/t_j is greater than or equal to job i's ratio w_i/t_i. ($w_j/t_j \geq w_i/t_i$)
- In order to show our greedy algorithm is optimal, we can show that if we swap the order of job i & job j and it does not affect the weighted sum of the completion time
- We can iteratively swap the position for any pair of job i & j until there are no other schedule to be considered, except what is left is our greedy algorithm solution.
- Let assume that each job's completion time is identical & let the time it takes to get to job i & j is F.
- The contribution time according to the greedy algorithm is $w_j(F+t_j) + w_i(F+t_j+t_i)$
- The contribution time according to the other solution is $w_i(F+t_i) + w_j(F+t_i+t_j)$

- The difference between the two is

$$\cancel{w_j F} + \cancel{w_j t_j} + \cancel{w_i F} + w_i t_j + \cancel{w_j t_i} - \cancel{w_i F} - \cancel{w_i t_i} - \cancel{w_j F} - w_j t_i - \cancel{w_j t_j}$$

$$= w_i t_j - w_j t_i$$

- According to our assumption, $w_j/t_j \geq w_i/t_i$, the difference is greater than zero.
- We can conclude that the swapping between job i & job j have no effect on the contributed completion time

Time Analysis:

- There are n jobs, takes $O(n)$ to go through
- Each job can be pair up with n-1 other job
- The total time it takes $O(n \cdot n-1) \approx \boxed{O(n^2)}$

② Ex. 17 (page 197)

- First, let assume that there are a total of n intervals, which is denoted by $A_1, \dots, A_n$
- For a fixed interval b, we want to find maximum size of interval that contains the interval A_b
- Assume there is a specific time point y locates at the interval A_b. In order to accept as many non-overlapped jobs as possible, we will first remove the interval A_b and all other intervals that overlap the interval A_b
- For the remaining intervals which does not contain the time point y, we can chop the intervals at time point y in order to get a set of intervals which we can apply the concept and strategy of the interval scheduling.
- The intervals that we are working with are in ordered based on their own end time.
- We are try to find solution that contains interval A_b with maximum size when b is from 1 up to n $(b = 1, \dots, n)$. Therefore, we claim the largest one to be our solution.
- Let assume that the optimal solution contains a series of intervals C, so there is an interval $A_b^{(b>0)}$ inside C. But if set of intervals C contains A_b which means C is a optimal solution that contains

interval A_b with maximum size.

- Our greedy algorithm solution is the same size as C.

Time analysis :

- when we first generate all the intervals that doesn't overlap with each order to prepare for the Interval Scheduling Problem takes $O(n)$
- In order to find the maximum size of all possible interval solution takes $O(n)$
- Total : $\boxed{O(n^2)}$

③ Exercise 3, page 246

- For this algorithm, we are gonna use the divide & conquer technique to solve this problem.
- First, we will divide all the cards into two sets which is first set contains $n/2$ while the second set contain the second $n/2$ cards.
- The algorithm will be run recursively on both sides.
- The algorithm is stated as follows :

• If we find a set of equivalent cards that consist of more that half of the cards, the algorithm will return one of the card in the set (respect to a card X)

• If equivalent cards exceed $n/2$ among all the cards, either one side or two side will have more than half of the cards that are same as X. Therefore, one among two recursive calls will give us a card that is equivalent to card X

• If the number of cards in our subset A is one, return the card

• If the number of cards in our subset A is two, we need to check if both cards are equal to each other. If they are equal, then just return either one of them.

- Let assume that A_1 is the first set of $n/2$ cards
- Let assume that A_2 is the second set of $n/2$ cards.
- We will call the algorithm to run recursively on set A_1
- If there is a card return at the end of all recursive calls, then we will compare this card with all the rest of the cards.
- If there is no card has been returned at the end of all the recursive calls on set A_1, then we will run recursively on set A_2.
  - If there is a card return at the end of all the recursive calls, then we will compare the card with the rest of the card.
- Return a card as the majority equivalent cards if we have found one.

Time Analysis

- The number of tests that are required for the algorithm based on a set of n cards are $S(n)$
- There is one recursive call on each side. If we go outside of the recursive call, the algorithm will have at most $2n$ comparisons between cards
- The total time it takes to run this algorithm is

$$2S(n/2) + 2n \approx \boxed{O(n\log n)}$$

④ Exercise 7 on page 248

- Assume N is a set that contains nodes are located along the border of the graph G.
- Let's say graph G has a specification that G has a node x that is not in set N which is adjacent to some arbitrary node in N, but smaller than all the nodes in N.
- If G satisfy the specified property, the outer minimum (not local minimum) does not lay on the border of N. Therefore, we can say there is at least one local minimum which are not located on the border of N.
- First, let G satisfy the property. If we don't consider the nodes that are on the border, we will let W be the set that contains nodes that are located at the middle row & column of the graph G.
- Then, we combine our nodes from set W & set N into a new set called set Z. We will remove set Z from graph G, then G will be divided into four grids.
- We want to find all the nodes that are adjacent to set Z and denoted that as set D.
- We want to find a node e in the union of set Z & set D. We have 2 possibilities.

- The first case is if node e is in set W, then node e is inside local minimum. because node e is the smallest among all the neighbors of node e from the union of set Z & D.
- The second case is node e is in set D.
- Let define smaller grid called SG that consist of node e & also the part that is between itself & set Z.
- The smaller grid SG also satisfy the property because node e is the smallest among all the nodes that are located at the border of SG & node e is adjacent to the border of SG.
- Therefore, the algorithm will keep recursively search for a local minimum that is inside in grid SG.
- In order to find a local minimum from grid graph G, we will find a node a that has the smallest value & locates on the border of set N
- We find the local minimum if the node a locates at the corner.
- If node a is smaller than node e, then node a is the local minimum.

(5)

1) Let arr be the array that store the sorted element.

2) int n = sizeof (arr) / sizeof (arr[0])

   countRotations (arr, 0, n-1)

3) countRotations is defined as follows:

   countRotations (int arr, int low, int high) :

     if high < low → when the array is not rotated at all
       return 0

     if high == low → if there is only one element left
       return low

     int middle = low + (high - low) / 2 → find the middle

     if (middle < high && arr[middle + 1] < arr[middle]) check if element (middle + 1) is minimum element
       return (mid + 1)

     if (mid > low && arr[mid] < arr[mid - 1]) → check if middle itself is the minimum element
       return mid

     if (arr[high] > arr[mid]) → decide whether to go left or right
       return countRotations (arr, low, middle - 1)

     return countRotations (arr, middle + 1, high)

Time Analysis

Since we incorporate the concept of binary search in to this algorithm, each stage of the search, we split the input in half, we successively reduce the size of the problem, so the time complexity is $\boxed{O(\log n)}$

(6) Extract the minimum

1) First, we copy the last value in the heap to the root to extract & remove the minimum

2) We gonna decrease the heap size by 1

3) We will shift down the root value according to the following rules :

- if current node has no children, the shift ends.
- if current node has one child ; check if the heap property is broken which is the child node is smaller than the parent node, then swap current node's value and child value ; sift down the child.
- if the current node has two children, find the smallest of them. If the heap property broken, then swap the current node's value & the selected child value ; shift down the child.

Time Analysis

The extracting the minimum number required $O(1)$. The height of the tree is $\log n$. The worst case is we need to compare & swap for every pair of parent & child node. Each swap cost $O(1)$. Therefore, the time complexity is also $\boxed{O(\log n)}$

⑥ Insert a new number in the balanced heap

1) Increase the size of heap by 1, n = n+1; n is the size of the heap

2) Insert the element at the end of Heap;
   arr[n-1] = new element value

3) Heapify the new node with a bottom up search.
   heapify(arr, n, n-1); arr is my array of n nodes.

4) My heapify function is gonna heapify the ith node in a Heap of size n; n-1 is my ith node to start with

5) First, let's find the parent of my newly inserted node
   int parent = (i-1)/2

6) If current node is greater than its parent, swap both of them & call heapify again for the parent

```
if (arr[parent] > 0)
    if (arr[i] > arr[parent])
        swap(arr[i], arr[parent])
        heapify(arr, n, largest)
```

Time Analysis

Since the heap has a complete binary tree structure, the height of the tree is logn. In the worst cost, the newly inserted element at the bottom has to be swapped at every level from bottom to top up, swap is needed on every level. Therefore the maximum number of times the swap needed to be performed is

$\boxed{O(\log n)}$

⑥ Change a number in a balanced heap

1) Replace the element to be changed by the the provided number.
2) The new number might not follow the heap property. we need to heapify
3) arr[index] = new element ; index is the position of the element to be changed.
4) In my heapify function (int arr[], int n, int i)

   heap size → n, index → i

   - initialize largest as my position of the element to be changed

```
int largest = i
int l = 2*i+1
int r = 2*i+2
if(l<n && arr[l]>arr[largest])
      largest = l
if (r<n && arr[r] > arr[largest])
      largest = r
if (largest != i)
      swap(arr[i], arr[largest])
      heapify (arr, n, largest)
```

Time Analysis

The replacing act of the element cost O(1). The heights of the tree is log n. The worst case is we need to compare & swap for every pair of parent & child node. Each swap cost O(1). Therefore, the time complexity is $\boxed{O(\log n)}$