HW4 CS 180

ASHER CHRISTIAN 006-150-286

1. Exercise 13 P 194

For the solution to this problem I propose completing the jobs in order or decreasing weight to time ratio, that is complete the job with the highest weight to time ratio first and then proceeding until the job with the lowest weight to time ratio. I prove the efficiency of my algorithm as follows.

Consider any arbitrary ordering and consider any two jobs C_i and C_j such that C_i immediately precedes C_j but the weight time ratio of C_i is less than C_j that is $\frac{w_i}{t_i} < \frac{w_j}{t_j}$ and let t be the time that C_i starts. then

$$C_i = t + t_i.$$

$$C_j = t = t_i + t_j.$$

and consider the partial sum difference if the two terms are swapped noting that every other element in the sum would be unaffected

$$\begin{aligned}(t+t_i)w_i + (t+t_i+t_j)w_j& \\ t(w_i+w_j) + t_iw_i + t_jw_j + t_iw_j &> t(w_1+w_j) + t_iw_i + t_jw_j + t_jw_i \\ &= (t+t_j)w_j + (t+t_j+t_i)w_i\end{aligned}$$

The inequality following from the fact that

$$\begin{gathered}\frac{w_j}{t_j} > \frac{w_i}{t_i} \\ t_iw_j > t_jw_i\end{gathered}$$

Thus if C_i and C_j were swapped so that

$$C_j = t + t_j.$$

$$C_i = t + t_j + t_i.$$

the overall sum contributed by the two jobs would be lesser. Thus any possible ordering of jobs can be rearranged in a finite amount of steps into an ordering that complies with my proposal in a way taht only decrease the total weighted sum of the costs and so my solution is optimal.

Date: 07.02.25.

2. Exercise 17 P 197

Sort the elements in increasing order of end time. Then for each job fix it and run the interview scheduling problem as before picking the first element to complete after the element. Thus we will create optimal solutions for each fixed job. Return the element that has the most jobs. Any optimal solution will contain at least one job and after fixing this job our algorithm will then find the rest of the optimal solution by the optimality discussed in class, therefore our algorithm would have found the solulution. Additionally since sorting takes $O(n \log n)$ and after fixing a job finding the ordering takes $O(n)$ which is done n times the overall complexity of this algorithm is $O(n^2)$

3. Exercise 3 P 246

I propose the following divide and conquer algorithm

```
• Merge Data(LeftSets, RightSets, LeftMaxes, RightMaxes)
  – For the two maxes in left compare the representatives with a representative
    from every element of right merging the two into the same group if possible
  – Do the same with the two right maxes in comparison to the left population
  –  Return the two largest of the original left maxes and right maxes
• Get Largest Groups(Credit Cards)
  – arbitrarily partition the set of credit cards into two groups and initialize
    every element into its own set for union find datastructure
  – If credit cards has two elements or one element return a list with either
    only one element if the two elements are equivalent or a list with two elements
    one for each element
  – Get Largest Groups(First Half) and assign it to a variable:  Left
  – Get Largest Groups(Second Half) assign it to a variable:  Right
  – Merge(Left half, Right half, Left, Right) and return the output
• Find N/2 (Credit Cards)
  – Run Get largest Groups(Credit Cards) and check if either of the unions outputted
    has more than $\frac{n}{2}$ elements.  If it does return true else return false
```

At the base case this algorithm obviously returns the two largest sets $\frac{n}{2}$ or greater of equivalency classes in the data Then for each additional set if the data were to have $\frac{n}{2}$ elements it must have at least $\frac{n}{4}$ elements in one of the two sublists which means it must be part of one of the four max elements of the two sublists so in searching those two sublists we guaarantee that we preserve any element that has $\frac{n}{2}$ elements. Thus inductively at each stage we are guaranteed any group that has at least $\frac{n}{2}$ elements so at the end of the algorithm if there is an equivalency class of $\frac{n}{2}$ or more elements it is returned. Additionally the merging task takes $O(n)$ time because for each of the four maxes possible a total of n correspondences are checked. additionally at each level two recurrences are called each with half the input so the total time is proportional to $O(n \log n)$

4. Exercise 5 P 248

I propose the following divide and conquer algorithm

```
• Merge(LeftIntervals, RightIntervals)
  –  Intervals given as arrays of intervals given in sorted order corresponding
    to line segments that are the highest in the corresponding intervals
  – Maintain a new array of intervals to be returned
  – Iterate through both intervals using Lp as the left interval pointer and
    Rp as the right interval pointer
```

```
        * Check where Lp and Rp intersect
        * If they intersect somewhere not in the interval of the two then add the
          greater interval to the list
        * If they intersect in the interval add two intervals one for the greater
          one up to the intersection then one for the greater one after the intersection
        * Advance the pointers corresponding to the ones checked
    – return the interval list
• Get Intervals(lines)
    – If there are two lines return the intervals that they each are maximized
      assuming unequal slope
    – If there is one line return it on the interval of the entire real line
    – arbitrarily partition the set of lines into two parts
    – Get Intervals(Left partition)
    – Get Intervals(Right partition)
    – Merge intervals
    – Return merged intervals
• Find lines
    – Get intervals on the lines and then return the lines correspondign to those
      intervals
```

5. Exercise 5 HW

Suppose you are given an array of sorted integers that has been circularly shifted k positions to the right. For example taking (1 3 4 5 7) and circularly shifting it 2 position to the right you get (5 7 1 3 4). Design an efficient algorithm for finding K. Iterate trhough the array checking every element with the next element if there is an inversion then return the index of the element that is inerted plus one. If the entire array is iterated through without inversion then the array was not shifted. This algorithm works because if the array was shifted left k times then the k element index of the shifted array is less than the $k-1$ element and everything else is in order so it suffices to find this k which is what the algorithm does.
This algorithm is trivial, and in fact I propose a different algorithm that is faster.

```
• Let x be the first element in the list
• Check if there are no inversions by checking if x is less than the last element
• Perform a binary search with the following search criteria:  If the element
  is less than the previous element it is an inversion and return its index-1,
  if it is greater than x search the right half, if the element is less than
  x search the left half
```

6. Exercise 6 HW

Given two sorted arrays of size m and n respectively, you are tasked with finding the element that would be at the k-th position of the final sorted array. Note that a linear time algorithm is trivial (and therefore we are not interested in). I propose the following algorithm

```python
    import math
a = [81, 86, 117, 128, 134, 168, 202, 230, 344, 347, 363, 426,
    444, 468, 543, 575, 605, 632, 682, 694, 695, 735, 763, 794,
     809, 811, 819, 826, 841, 855, 895, 924, 925, 952]
b = [57, 99, 104, 105, 111, 175, 284, 292, 310, 314, 317, 332,
    417, 426, 548, 649, 715, 722, 733, 773, 820, 886, 917, 924,
     926, 937, 950, 967, 988]
```

```
k = 47



def f( prevdist, k , x, Left, Right):
    if(k == 1):
        return min(Left[0], Right[0])
    x = int(x)
    y = k - x
    print(Left[x-1],Right[y-1])
    dist = 0
    if Left[x-1] >= Right[y-1]:
        if len(Right) > y:
            if Left[x-1] <= Right[y]:
                return Left[x-1]
            else:
                dist = min(math.ceil((1/2)*prevdist), len(Right
                    ) - y)
                x = x - dist
        else:
            return Left[x-1]
    else:
        if len(Left) > x:
            if Right[y-1] <= Left[x]:
                return Right[y-1]
            else:
                dist = min(math.ceil((1/2)*prevdist), len(Left)
                     - x)
                x = x + dist
        else:
            return Right[y-1]
    return f(dist, k , x , Left, Right)


if(len(b) > len(a)):
    c = a
    a = b
    b = c


for i in range(1,len(a)+len(b)+1):
    c = max(math.ceil(1/2 * i), i - len(b))
    print(f(math.ceil(1/2 * i), i, c, a, b))
```

Replacing k with the index necessary and a and b with the arrays in sorted order as given This algorithm works because at each step of the algorithm it correctly determines if the element is the kth element as well as determining if the correct element is farther or further behind in the left list while remembering all interval sizes. It also decreases its search space by a factor of 2 every iteration and thus completes in $O(\log n)$ time. .