Homework 3

1. Solve Kleinberg and Tardos, Chapter 4, Exercise 3. (15 points)
   Assume the greedy algorithm currently in use fits boxes b_1, b_2, …, b_j into the first k trucks. We prove that no other algorithm can fit more boxes into k trucks, i.e., if an algorithm fits boxes b1, b2, …, bi into the first k trucks, then i ≤ j. We prove this claim by induction on k:

- For k = 1: the greedy fits as many boxes into one truck as possible, it is clear that no other algorithm can x more boxes into the truck, thus, i ≤ j.
- Assume it holds for k - 1, i.e., if the greedy algorithm fits boxes b1, b2, …, bj into the first k - 1 trucks, and the other algorithm fits boxes b_1, b_2, …, b_i into the first k - 1 trucks, then i ≤ j.
- For k: the greedy algorithm fits j′ boxes into the first k - 1 trucks, the other algorithm fits i′ boxes into the first k - 1 trucks, and i′ ≤ j′; now, for the k-th truck, the other algorithm packs in boxes bi′+1, …, bi; since i′ ≤ j′, the greedy algorithm is able at least to t all the boxes bj′+1, … ,bi into the k-th truck, and it may be able to fit more

   Rubric:
   15 points for the correct proof.
   - 3 points for establishing the base case (k=1) correctly.
   - 8 points for Inductive Hypothesis and Step
   - 4 points for explaining the k-th step

2. Solve Kleinberg and Tardos, Chapter 4, Exercise 5. (15 points)
   Replaced with Ungraded Problems Q2

3. There are N tasks that need to be completed using 2 computers A and B. Each task *i* has 2 parts that take time: *ai* (first part) and *bi* (second part) to be completed. The first part must be completed before starting the second part. Computer A does the first part of all the tasks while computer B does the second part of all the tasks. Computer A can only do one task at a time, while computer B can do any amount of tasks at the same time. Find an $O(n \log n)$ algorithm that minimizes the time to complete all the tasks, and give a proof of why the solution obtained by the algorithm is optimal. (15 points)

   Sort the tasks in decreasing order of bi. Perform the tasks in that order. Basically computer A does the first parts in that order, and computer B starts every second part after computer A finishes the first part.
   Proof: We show that given solution G is actually the optimal schedule, using an exchange argument. We define an inversion to be a pair of jobs whose order in the schedule does not agree with the order of their finishing times, i.e. job k and l form an inversion if $bk <= b1$ but job k is

scheduled before job 1. We will show that for any given optimal schedule $S \neq G$, we can repeatedly swap adjacent jobs with inversion between them so as to convert S into G without increasing the completion time.

1. Consider any optimal schedule S, and suppose it does not use the order of G. Then this schedule must contain an "inversion", i.e. two jobs Jk and $J1$ so that $J1$ runs directly after Jk but the finishing time for the first job is less than the finishing time for the second one, i.e. $bk <= b1$. We can remove this inversion without affecting the optimality of the solution. Let S' be the schedule obtained from S where we swap only the order of Jk and $J1$. It is clear that the finishing times for all jobs except Jk and $J1$ do not change. The job $J1$ now schedules earlier, thus this job will finish earlier than in the original schedule. The job Jk schedules later, but computer A hands off Jk to computer B in the new schedule S' at the same time as it would handed off $J1$ in the original schedule S. Since the finishing time for Jk is less than the finishing time for $J1$, the job Jk will finish earlier in the new schedule than $J1$ would finish in the original one. Hence our swapped schedule does not have a greater completion time.

2. Since we know that removing inversions will not affect the completion time negatively if we are given an optimal solution that has any inversions in it, we can remove these inversions one by one without affecting the optimality of the solution. When there are no more inversions, this solution will be the same as ours. Therefore the completion time for G is not greater than the completion time for any other optimal schedule S. Thus G is optimal.

Rubric:
7 points for the correct algorithm.

- -3 for not mentioning sorting in descending order of B.
- -2 for not mentioning Computer A follows the sorted order.

Proof rubrics: 8 points for the correct proof.

- -2 points for not defining what the inversions in this case are
- -2 points for not generalizing the proof
- -1 points for not showing that the algorithm is now the same as our greedy algorithm

4. (a) Consider the problem of making change for n cents using the fewest number of coins. Describe a greedy algorithm to make change consisting of quarters(25 cents), dimes(10 cents), nickels(5 cents) and pennies(1 cents). Prove that your algorithm yields an optimal solution. (Hints: consider how many pennies, nickels, dimes and dime plus nickels are taken by an optimal solution at most.)
(b) For the previous problem, give a set of coin denominations for which the greedy algorithm does not yield an optimal solution. Assume that each coin's value is an

integer. Your set should include a penny so that there is a solution for every value of n. (15 points)

(a) Denote the coins values as $c1 = 1$, $c2 = 5$, $c3 = 10$, $c4 = 25$.

1) if $n = 0$, do nothing but return.

2) Otherwise, find the largest coin ck , $1 \le k \le 4$, such that $ck \le n$. Add the coin into the solution coin set S.

3) Subtract ck from n, and repeat the steps 1) and 2) for $n - ck$

For the proof of optimality, we first prove the following claim: Any optimal solution must take the largest ck, such that $ck \le n$. Here we have the following observations for an optimal solution:

1. Must have at most 2 dimes; otherwise we can replace 3 dimes with quarter and nickel.
2. If 2 dimes, no nickels; otherwise we can replace 2 dimes and 1 nickel with a quarter.
3. At most 1 nickel; otherwise we can replace 2 nickels with a dime.
4. At most 4 pennies; otherwise can replace 5 pennies with a nickel.

Correspondingly, an optimal solution must have

- Total value of pennies: ≤ 4 cents.
- Total value of pennies and nickels: $\le 4 + 5 = 9$ cents.
- Total value of pennies, nickels and dimes: $\le 2\times10+4 = 24$ cents.

Therefore,

- If $1 \le n < 5$, the optimal solution must take a penny.
- If $5 \le n < 10$, the optimal solution must take a nickel; otherwise, the total value of pennies exceeds 4 cents.
- If $10 \le n < 25$, the optimal solution must take a dime; otherwise, the total value of pennies and nickels exceeds 9 cents.
- If $n \ge 25$, the optimal solution must take a quarter; otherwise, the total value of pennies, nickels and dimes exceeds 24 cents.

Compared with the greedy algorithm and the optimal algorithm, since both algorithms take the largest value coin ck from n cents, then the problem reduces to the coin changing of $n - ck$ cents, which, by induction, is optimally solved by the greedy algorithm.

Rubrics

- Algorithm: 4 pts
- Claim and proof: 3 pts
- Final proof with induction: 3 pts

(b) Coin combinations = {1, 15, 20} cents coins.Consider this example $n = 30$ cents. According to the greedy algorithm, we need 11 coins: $30 = 1\times20+10\times1$; but the optimal solution is 2 coins $30 = 2\times15$

Rubrics

- Correct example: 5 pts

Ungraded Problems

1. Consider a collection of n ropes which have lengths $L_1, L_2, \ldots, L_n$, respectively. Two ropes of length L and L' can be connected to form a single rope of length $L + L'$, and doing so has a cost of $L + L'$. We want to connect the ropes, two at a time, until all ropes are connected to form one long rope. Design an efficient algorithm for finding an order in which to connect all the ropes with minimum total cost. You do not need to prove that your algorithm is correct.

   Enter all rope segments into a min-heap with the length of the rope segment being its key value. Iteratively pop the 2 shortest ropes and connect them. Then insert the new resulting rope segment (with the key value being the sum of the lengths of the ropes that are connected together) back into the heap. Continue until you are left with only 1 rope segment in the heap.

2. Suppose you want to drive from USC to Santa Monica. Your gas tank, when full, holds enough gas to drive p miles. Suppose there are n gas stations along the route at distances $d_1 \leq d_2 \leq \ldots \leq d_n$ from USC. Assume that the distance between any neighboring gas stations, and the distance between USC and the first gas station, as well as the distance between the last gas station and Santa Monica, are all at most p miles. Assume you start from USC with the tank full. Your goal is to make as few gas stops as possible along the way. Design an efficient algorithm for determining the minimum number of gas stations you must stop at to drive from USC to Santa Monica. Prove that your algorithm is correct. Analyze the time complexity of your algorithm.

   Greedy algorithm: The greedy strategy we adopt is to go as far as possible before stopping for gas. That is, when you are at the i-th gas station, if you have enough gas to go to the (i+1)-th gas station, then skip the i-th gas station. Otherwise stop at the i-th station and fill up the tank.

   Proof of optimality:
   The proof is similar to that for the interval scheduling solution we did in lecture. We first show (using mathematical induction) that our gas stations are always to the right of (or not to the left of) the corresponding base stations in any optimal solution. Using this fact, we can then easily show that our solution is optimal using proof by induction.

   (a) First we show our gas stations are never 'earlier' than the corresponding gas station in any optimal solution:
   Let $g1, g2, \ldots, gm$ be the set of gas stations at which our algorithm made us refuel. Let $h1, h2, \ldots, hk$ be an optimal solution. We first prove that for any indices $i < m$, $hi \leq gi$.

   Base case: Since it is not possible to get to the ($g1$+1)-th gas station without stopping, any solution should stop at either $g1$ or a gas station before $g1$, thus $h1 \leq g1$.

   Induction hypothesis: Assume that for the greedy strategy taken by our algorithm, $hc \leq gc$.

Inductive step: We want to show that $hc+1 \leq gc+1$. It follows from the same reasoning as above. If we start from hc, we first get to gc (IH) and, when leaving gc, we now have at most as much fuel as we did if we had refilled at gc. Since it is not possible to get to $g(c+1)+1$ without any stopping, any solution should stop at either $gc+1$ or a gas station before $gc+1$, thus $hc+1 \leq gc+1$

(b) Now assume that our solution requires m gas stations and the optimal solution requires fewer gas stations. We now look at our last gas station. The reason we needed this gas station in our solution was that there is a point on I-10 after this gas station that cannot be reached with the amount of gas when we left gas station m-1. Therefore we would not have enough gas if we left gas station m-1 in any optimal solution. Therefore, any optimal solution also would require another gas station.

The running time is O(n) since we at most make one computation/decision at each gas station

- Greedy algorithm: 5 pts
- "Stay-ahead argument" of the proof (part a): 8 pts
    - – Induction base: 2 pts
    - – Induction hypothesis: 3pts
    - – Induction step: 3 pts
- Completing the argument of optimality in the proof (part b): 2 points

3. The array A below holds a max-heap. What will be the order of elements in array A after a new entry with value 18 is inserted into this heap? Show all your work.
   A = {15, 12, 11, 8, 7, 9, 3, 2, 4, 2, 1}

Initial Array

Index	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
Element	15	12	11	8	7	9	3	2	4	2	1

Array after Insertimg 18

Index	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**
Element	15	12	11	8	7	9	3	2	4	2	1	18

18>9 (element at index 12/2=6), so swap.

Index	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**
Element	15	12	11	8	7	18	3	2	4	2	1	9

18>11 (element at index 6/2=3), so swap.

Index	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**

Element	15	12	18	8	7	11	3	2	4	2	1	9

18>15 (element at index 3/2=1), so swap.

Index	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**
Element	18	12	15	8	7	11	3	2	4	2	1	9

Final Array = {18, 12, 15, 8, 7, 11, 3, 2, 4, 2, 1, 9}