Due: Thu Apr 16th at the start of class

Homework #4

CMSC351 - Spring 2013

PRINT Name _____:

- *Grades depend on neatness and clarity.*
- *Write your answers with enough detail about your approach and concepts used, so that the grader will be able to understand it easily. You should ALWAYS prove the correctness of your algorithms either directly or by referring to a proof in the book.*
- *Write your answers in the spaces provided. If needed, attach other pages.*
- *The grades would be out of 16. Four problems would be selected and everyone's grade would be based only on those problems. You will also get 4 bonus points for trying to solve all problems.*

1. Given two sets S_1 and S_2, and a real number x, find whether there exists an element from S_1 and an element from S_2 whose sum is exactly x. The algorithm should run in average time $O(n)$, where n is the total number of elements in both sets. (Try to use the data structures you have learned in the lectures.)

   Use a Hash data structure to store all the members of S_1. For every member u of S_2, we simply search for $x - u$ in the data structure. In a hash data structure the running time of insertion and search are O(1) in average , thus the algorithm runs in O(n) time in average.

2. [Prob 4.22,Pg 88] Determine the general structure of a binary search tree formed by inserting the numbers 1 to n in order (you may draw a general structure, but explain it properly). What is the height of this tree?

The tree is a path of length $n - 1$. Hence its height is also $n - 1$.

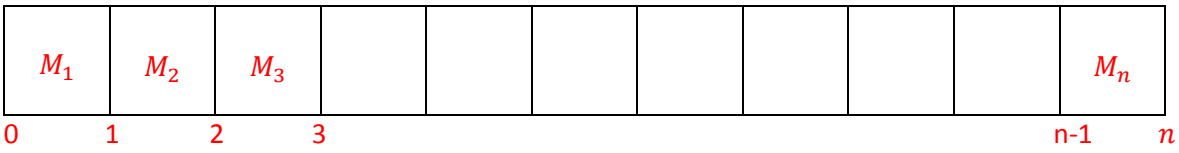
3. [Prob 4.16,Pg 88] Design a data structure that supports the following operations. Each operation should take $O(\lg n)$ time, where n is the number of elements in the data structure. Explain the required process for doing each operation and prove that it takes $O(\lg n)$ time.
   a) $Insert(x)$: Insert the key x into data structure only if it is not already there.
   b) $Delete(x)$: Delete the key x, if it is there!
   c) $Find\_Smallest(k)$: Find the kth smallest key in the data structure.

A balanced binary search tree (e.g. AVL tree) can perform the first two types of commands. We modify the binary search tree such that we can also perform the third type of command in O(lg n).

In each node of the tree v, store a number ℓ_v which is equal to the number of vertices in the left subtree of that node. Insertion and Deletion of an AVL tree can be easily adjusted to update these values.
To perform the last command, we implement a recursive function $fs(v,k)$ which finds the $k$the element in the subtree rooted at node v. Clearly the answer to operation (c) would be $fs(root,k)$. Now when we want to find the kth smallest element, we simply check ℓ_v. If $\ell_v = k-1$, then node v is the answer to $fs(v,k)$. If $\ell_v < k-1$, the answer would be in the right subtree of v. Thus the answer would be $fs(right\_child\_of\_v\ , k-\ell_v-1)$. Similarly, if $\ell_v > k-1$, the answer would be $fs(left\_child\_of\_v, k)$.

4. **Scheduling jobs intervals with penalties:** For each $1 \leq i \leq n$ job j_i is given by two numbers d_i and p_i, where d_i is the deadline and p_i is the penalty. The length of each job is equal to 1 minute. We want to schedule all jobs, but only one job can run at any given time. If job *i* does not complete on of before its deadline d_i, we should pay its penalty p_i. Design a greedy algorithm to find a schedule which minimizes the sum of penalties.

Observation: We can assume that all jobs finish after n minutes. Suppose not. So there is an empty minute starting at say $0 \leq k \leq n-1$ (where no job is scheduled) and there is a job j_i for some $1 \leq i \leq n$ which is scheduled some time after n minutes. If we schedule job j_i to start at minute k, then this can only be a better solution since everything remains same except j_i might now be able to meet its deadline.



We assign time intervals M_i for $1 \leq i \leq n$ where M_i starts at minute $i-1$ and ends at minute i. The greedy algorithm is as follows:

- Arrange the jobs in decreasing order of the penalties $p_1 \geq p_2 \geq \cdots \geq p_n$ and add them in this order
- To add job j_i, if any time interval M_l is available for $1 \leq l \leq d_i$, then schedule j_i in the last such available interval. Else schedule j_i in the first available interval starting backwards from M_n

Let S be the greedy schedule and suppose it schedules job j_i at time t_i for each $1 \leq i \leq n$. We show by induction that there is an optimal schedule T_r that agrees with S on the schedule of the first r intervals for $1 \leq r \leq n$.

Base Case: The base case is when $r = 1$. Let S′ be an optimal schedule. Suppose S' schedules j_1 at time $t \neq t_1$. By our observation above, S' must schedule some job j_k at time t_1. Let T_1 be the schedule obtained by swapping times of j_1 and j_k in S′. Note that T_1 makes the greedy choice to schedule j_1. In each of the following 3 cases we show that penalty of T_1 is at most that of S′.

- $d_1 \geq t_1$ and j_1 does not incur penalty in S′. This implies $t_1 > t$ since t_1 was the last available slot for j_1 in greedy algorithm. So in T_1, j_1 still does not incur penalty and j_k gets scheduled at t which is earlier than t_1.
- $d_1 \geq t_1$ and j_1 incurs penalty in S′. This implies $t > d_1 \geq t_1$. In T_1 we do not pay penalty for j_1 since $d_1 \geq t_1$. Worst case we might pay the penalty p_k for j_k. So the net difference is penalty is $p_k - p_1 \leq 0$.
- $d_1 < t_1$. Hence there is no way to schedule j_1 without incurring penalty. Hence j_1 pays p_1 in both S′ and T_1. In greedy algorithm since t_1 is chosen as last available slot, we have $t_1 > t$. In T we schedule j_k at t instead of t_1, which can only reduce the penalty of j_k.

Inductive Hypothesis: There is an optimal schedule T_r that agrees with S on the schedule of the first r intervals.

Inductive Step: If $r = n-1$, then by the Observation above the job j_n must be scheduled in the only remaining slot from 1 to n. If $r < n-1$ then consider the optimal schedule T_r. Let it schedule the job j_{r+1} at time $t \neq t_{r+1}$. By Observation above, T_r schedules a job say j_h at time t_{r+1}. Since T_r agrees with S on first r jobs we have $h > r+1$. Let T_{r+1} be the schedule that

swaps times of j_h and j_{r+1}, $i.e.$, schedules j_h at time t and j_{r+1} at time t_{r+1}. Note that T_{r+1} agrees with S on first $r+1$ jobs. A very similar argument as in the base case shows that penalty of T_{r+1} is at most as much as the penalty of T_r. Then the optimality of T_r implies the optimality of T_{r+1} as well.

5. **Making Change**: Suppose we want to make change for n cents and the only denominations allowed are 1, 10 and 25 cents.
    a. Find an example such that the greedy algorithm does not find the minimum number of coins required to make change for n cents (give a concrete counterexample).
    b. Give a $O(n)$ dynamic programming algorithm to find the minimum number of coins required to make change for n cents.

Let C_n be the least number of coins needed to make change for n cents
For $1 \leq n \leq 9$ we have $C_n = n$ since the only way is to use n pennies
For $10 \leq n \leq 24$ we have $C_n = C_{n-10} + 1$ since now we must use a dime
For $25 \leq n$ we have $C_n = \min(C_{n-10} + 1,\ C_{n-25} + 1)$ since now we must use either a dime or a quarter

There is also a simple $O(1)$ algorithm !!

6. **Weighted Interval Scheduling:** Consider a set of n intervals where each interval is given by (s_i, t_i) Where s_i is the start time and t_i is the finish time. In addition each interval also has a weight given by w_i. Give a dynamic programming algorithm to find the maximum weight of a non-conflicting set of intervals.

   First sort the intervals in increasing order of finishing times. This takes O(n log n) time. Let $1 \leq p_j \leq j-1$ denote the largest index of an interval which can be scheduled with j. Since we have sorted the intervals, we can find the indices p_j for each $1 \leq j \leq n$ in O(n) time.

   For each $1 \leq j \leq n$ let OPT_j denote the maximum weight possible from only the intervals 1,2, … j. Clearly $OPT_1 = w_1$.
   For each $2 \leq i \leq n$ we have $OPT_i = \max\ (\ OPT_{i-1},\ w_i + OPT_{p(j)})$ since you can either have the interval i in your optimal schedule (in which case the max weight you can gain from earlier intervals is $OPT_{p(j)}$) ; or the interval i is not present in the optimal schedule (in which case the max weight you can gain from earlier intervals is OPT_{i-1} by definition). Hence each OPT_i can be found in constant time (it only involves making one comparison). Therefore, the answer which is OPT_n can be calculated in $O(n)$ time.

   Note that the total time taken is $O(n.\log n\ ) + O(n) + O(n) = O(n.\log n\ )$

7. **Scheduling a Class:** A professor needs to choose a sequence of locations to conduct a class, one for each day. The days are numbered $1,2,\dots,n$. The two possible locations are AVW (A.V. Williams) and $CSIC$. For each $1 \leq i \leq n$, the cost of conducting the class in AVW on day i is A_i and the cost of conducting the class in $CSIC$ on day i is C_i. The cost of moving from AVW to $CSIC$ (and vice versa) is some constant M. Give an $O(n)$ algorithm which computes the cost of an optimal schedule of the class.

   For each $0 \leq i \leq n$ and $X \in \{AVW, CSIC\}$ let $T(i,j)$ denote the cost of an optimal schedule for the first i days given that the last day the class is held in X.
   Clearly, $T(0, AVW) = 0 = T(0, CSIC)$. For each $1 \leq i \leq n$, we have

$$T(i, AVW) = \min\ (T(i-1, AVW) + A_i\ ;\ T(i-1, CSIC) + C_i + M\ )$$
$$T(i, CSIC) = \min(\ T(i-1, CSIC) + C_i\ ;\ T(i-1, AVW) + A_i + M\ )$$

   The final answer for our problem is $\min\ \{\ T(n, CSIC); T(n, AVW)\ \}$

   [This problem was taken from the Kleinberg-Tardos book]