CSC 545 **Homework 3 (version 1)** **Spring 2022**

This homework is due Thursday, April 14, at 3:30pm MST. Please upload a single `PDF` file containing your submission (ensuring scans of handwritten work are legible) to `Gradescope` by that time.

The questions are drawn from the material in the lectures, and in Chapters 16 and 17 of the text, on *greedy algorithms* and *amortized algorithms.*

The homework is worth a total of 100 points. When point breakdowns are not given for the parts of a problem, each part has equal weight.

For questions that ask you to design a *greedy* algorithm, prove that your algorithm is correct making use of a lemma of the following form:

> **Lemma** *If a partial solution P is contained in an optimal solution, then the greedy augmentation of P is still contained in an optimal solution.*

Prove the lemma using an exchange-style argument, where you transform the optimal solution to contain the augmentation and argue that this transformation does not worsen the solution value.

Please remember to start each problem on a *new page*, and mark the *corresponding pages* in your submission for each homework problem on `Gradescope`. Conciseness counts!

(1) **(Continuous knapsack)** (20 points) In the Continuous Knapsack problem, the input is a collection of n *items* indexed $1, 2, \ldots, n$, each with an associated *weight* w_i and an associated *value* v_i, together with a weight *capacity* k. The output is a collection of real-valued fractional amounts $0 \le f_i \le 1$ to take of each item i, such that the total weight taken for the items $\sum_{1 \le i \le n} f_i\, w_i \;\le\; k$ does not exceed the capacity, and the total value taken for the items $\sum_{1 \le i \le n} f_i\, v_i$ is maximum.

Design a *greedy algorithm* for Continuous Knapsack that finds an optimal solution in $O(n \log n)$ time, and prove that it is *correct* using a greedy augmentation lemma of the form stated above.

(2) **(Minimizing average completion time)** (30 points) Suppose you are given a collection of n tasks that need to be scheduled. With each task, you are given its duration. Specifically, task i takes t_i units of time to execute, and can be started at any time. At any moment, only one task can be scheduled.

The problem is to determine how to schedule the tasks so as to minimize their *average completion time.* More precisely, if c_i is the time at which task i completes in a particular schedule, the average completion time for the schedule is $\frac{1}{n} \sum_{1 \le i \le n} c_i$.

(a) (30 points) Design an efficient *greedy* algorithm that, given the task durations t_1, t_2, …, t_n, finds a schedule that minimizes the average completion-time, assuming that once a task is started it must be run to completion.

Analyze the running time of your algorithm, and prove that your algorithm is correct using a lemma of the required form.

(b) **(bonus)** (10 points) Suppose with each task we also have a *release time* r_i, and that a task may not be started before its release time. Furthermore, tasks may be *preempted*, in that a scheduled task can be interrupted and later resumed, and this can happen repeatedly.

Design an algorithm that finds a schedule that minimizes the average completion-time in this new situation. Analyze its running time and prove that it is correct.

(3) **(Deleting the larger half)** (20 points) Design a data structure that supports the following two operations on a set S of integers:

- `Insert`(x, S), which inserts element x into set S, where x is not currently in S, and
- `DeleteLargerHalf`(S), which deletes the largest $\lceil |S|/2 \rceil$ elements from S.

Show how to implement this data structure so both operations take $O(1)$ amortized time.

(Note: You may use the accounting method or the potential method for your analysis.)

(4) **(Binary search with insertions)** (30 points) Storing a set of n key-item pairs (k_i, x_i) in an array sorted by keys allows us to efficiently *find* an item x_i by its key k_i using binary search in $O(\log n)$ time. To *insert* a new element into a sorted array is very inefficient, however, and takes $\Theta(n)$ time in the worst case. Using multiple sorted arrays, we can achieve a much better balance between the time for finding and inserting elements.

Suppose we store the n elements in a data structure D that uses $\ell = \lceil \lg(n+1) \rceil$ different arrays. We refer to these arrays as $A_1, A_2, \ldots, A_\ell$. Array A_i has length 2^{i-1}, and is *full* if bit i is 1 in the *binary representation* of n; otherwise, if bit i is 0, array A_i is *empty.* Each array A_i is *sorted* by its keys, but we do not maintain any ordering relationship between the keys in different arrays.

We wish to support two operations on this data structure D:

- `Find`(k, D), which returns the item x associated with key k in D, and
- `Insert`(k, x, D), which inserts the pair (k, x) into D.

(a) (10 points) Show how to implement `Find` so it takes $O(\log^2 n)$ worst-case time.

(b) (20 points) Show how to implement `Insert` so it takes $O(\log n)$ *amortized* time, even though it can take $\Theta(n)$ worst-case time.

Show how to implement `Find` so it takes $O(\log^2 n)$ worst-case time, and `Insert` so it takes $O(\log n)$ *amortized time* (even though an insertion can take $\Theta(n)$ worst-case time).

(Hint: Review the material in Section 17.1 of the text on incrementing a binary counter. The aggregate method may be convenient for your amortized analysis in Part (b).)

Note that Problem (2)(b) is a *bonus* question. It is *not* required, and its points are not added to regular points.