Introduction to Algorithms: 6.006
Massachusetts Institute of Technology — October 5th, 2010
Professors Konstantinos Daskalakis and Patrick Jaillet — Handout 3

Problem Set 3

This problem set is divided into two parts: Part A problems are theory questions, and Part B problems are programming tasks.

Part A questions are due **Tuesday, October 19th** at **11:59PM**.

Part B questions are due **Thursday, October 21st** at **11:59PM**.

Solutions should be turned in through the course website. Your solution to Part A should be in PDF form using LaTeX or scanned handwritten solutions. Your solution to Part B should be a valid Python file, together with one PDF file containing your solutions to the two theoretical questions in Part B.

Templates for writing up solutions in LaTeX are available on the course website.

Remember, your goal is to communicate. Full credit will be given only to the correct solution which is described clearly. Convoluted and obtuse descriptions might receive low marks, even when they are correct. Also, aim for concise solutions, as it will save you time spent on write-ups, and also help you conceptualize the key idea of the problem.

Part A: Due Tuesday, October 19th

1. **(20 points)** Finding the Largest i Elements in Sorted Order

   Given an array of n numbers, we want to find the i largest elements in sorted order. That is, we want to produce a list containing, in order, the largest element of the array, followed by the 2nd largest, etc., until the ith largest. Assume that i is fixed beforehand, and all inputs have $n > i$.

   (a) **(5 points)** One idea is to mergesort the input array in descending order, and then list the first i elements of the array. Analyze the running time of this algorithm in terms of n and i.

   (b) **(10 points)** Describe an algorithm that achieves a faster asymptotic time bound than the one in Part (a). Analyze the running time of this algorithm in terms of n and i.

   (c) **(5 points)** Now suppose that the elements of the array are drawn, without replacement, from the set $\{1, 2, ..., 2n\}$. Give an algorithm that solves the problem, with this additional information, and analyze its running time in terms of n and i.

   For parts (b) and (c), full credit will be awarded only for an asymptotically optimal solution, with partial credit given for solutions with slower asymptotic running times.

2. **(15 points)** Dynamic Medians

   Marianne Midling needs a data structure "DM" for maintaining a set S of numbers, supporting the following operations:

- CREATE( ): Create an empty set S
- INSERT(x): Add a new given number x to S
- MEDIAN( ): Return a median of S. (Note that if S has even size, there are two medians; either may be returned. A median of a set of n distinct elements is larger than or equal to exactly $\lfloor (n+1)/2 \rfloor$ or $\lceil (n+1)/2 \rceil$ elements.)

(Assume no duplicates are added to S.)

Marianne proposes to implement this "dynamic median" data structure DM using a max-heap A and a min-heap B, such that the following two properties always hold:

1. Every element in A is less than every element in B, and
2. the size of A equals the size of B, or is one less.

To return a median of S, she proposes to return the minimum element of B.

(a) **(5 points)** Argue that this is correct (i.e., that a median is returned).

(b) **(10 points)** Explain how to implement INSERT(x), while maintaining the relevant properties. Analyze the running time of your INSERT algorithm in terms of n, the number of elements in S.

3. **(15 points)** The Optimality of the Binary Search Tree Construction

Consider a list of n numbers and a task of building a binary search tree that contains them. In Problem 2.1 (from Problem set 2) we outlined an algorithm that performs this task by starting with an empty tree and inserting to it the elements of the list one by one. As we mentioned, by employing appropriate rebalancing procedures, e.g. as in AVL-trees, the total running time of this procedure is $O(n \log n)$.

Prove that in the comparison-based model of computation this running time is asymptotically optimal. Namely, show that there is no algorithm in the comparison-based model that given a list of n numbers constructs a binary search trees containing them in $o(n \log n)$ time.

Hint: As a first step, show that given a binary search tree one can output a sorted list of all the numbers it contains in $O(n)$ time.

Part B: Due Thursday, October 21st

(50 points) Job Scheduling

Irene B. Matthews, a very smart MIT undergrad, constructed a large-scale quantum computer in her backyard. Although such a machine would allow her to break all the modern cryptography, she decided to become a billionaire through legal means. She founded a company that rents the machine time to the customers. It turned out, however, that the demand for such a service was far larger than Irene expected. In particular, after being flooded with a stream of hundreds of thousands of offers, she realized her machine could handle only a tiny fraction of these requests. Thus, overwhelmed by the situation, she hired you to help her out in dealing with the problem.

To formalize the task you are facing, let us divide time into one-hour-long steps. In each step there is a batch of offers (jobs) from clients arriving. Each offer consists of the reward (number of dollars) the client is willing to pay for having his/her job completed, and of its running time, being the number of one-hour-long steps needed to complete this job on Irene's machine. In our model, we assume that once the job is run on the machine it cannot be stopped until it finishes, and that the machine can process only one job at a time.

1. **(35 points)** Your first task is to implement `Scheduler`, a data structure that facilitates scheduling of the incoming jobs on the machine. This data structure should contain the currently processed job (together with amount of steps remaining till its completion), and the pool of yet not processed jobs. `Scheduler` is initialized by specifying a ranking function `rank` – this function takes a job as an argument and assigns score to it. Once initialized, it starts with an empty pool and is processing a job that has zero reward and running time of one. `Scheduler` should support the following operations:

   - `add(r,t)` – adds a job having reward r and running time t to the pool of yet not processed jobs;
   - `advance_time()` – reduces the number of steps needed to complete the currently processed job by one. If this causes the job to finish, a job in the pool that maximizes the value of `rank` function should begin being processed (and be removed from the pool);
   - `summary()` – returns the total reward accumulated for the jobs that were completed so far.

   Remember, since the `Scheduler` will need to deal with large amounts of jobs, your implementation should be as efficient as possible.

   Note: If your implementation of `Scheduler` *utilizes some data structure you have to implement this data structure yourself.*

2. **(10 points)** Although the jobs do not expire, i.e. they stay in the job pool unless we complete them, and the total available machine time of Irene's computer is relatively large, we might not be able complete to all of them. Therefore, one needs to choose carefully which jobs from the pool to process so as to maximize the total reward collected over time. In our setting, the choice of jobs is facilitated by specifying the `rank` function. Here is a list of possible ranking functions:

   - most valuable job first – the score assigned by `rank` to a job with reward r is equal to r;
   - shortest job first – the score assigned by `rank` to a job with running time t is equal to $-t$;
   - largest reward/running-time ratio first – the score assigned by `rank` to a job with reward r and running time t is equal to r/t;
   - largest running-time/reward ratio first – the score assigned by `rank` to a job with reward r and running time t is equal to t/r.

   Which one among these `rank` functions would you choose? Why?

3. **(5 points)** In our model we assumed that once the job is started on a machine it has to be processed till finish. Do you think that an ability to cancel already started jobs before their completion (without getting any reward for already processed part) might allow better performance? If so, construct an example of a job arrival pattern that yields larger total reward if canceling is allowed. Your example should correspond to at most four steps of total machine time of Irene's computer and involve at most four jobs.