CS 1510 Midterm 1
Fall 2018

Instructions

- If you are unsure how to proceed at some point, you will receive more partial credit for admitting this, than for proceeding in an incorrect direction.
- Algorithm descriptions that are predominantly pseudo-code, and that do not contain sufficient explanation, will receive no credit.
- Algorithm descriptions that use an array data structure must contain a statement of the intended meaning of the array entries. For example "The intended meaning of $A[i, j]$ is the length of the longest increasing subsequence of $x_i, \ldots, x_i$ that ends at x_j. " Algorithm descriptions that do not include such a statement, will receive no credit.

1. (40 points) We consider the following problem:

   INPUT: A collection of jobs $J_1, \ldots, J_n$, where the ith job is a 3-tuple (r_i, x_i, d_i) of non-negative integers.

   OUTPUT: A preemptive feasible schedule for these jobs on one processor, if such a schedule exists. A schedule is *feasible* if every job job J_i is run for x_i time units between its release time r_i and its deadline d_i. If there is no feasible schedule, the algorithm's output should be a statement of this.

   We consider greedy algorithms for solving this problem that schedule times in an online fashion, that is the algorithms are of the following form:

   $t = 0$;

   while there are jobs left not completely scheduled

   pick a job J_m to schedule at time t;

   increment t;

   One can get different greedy algorithms depending on how job J_m is selected. For each of the following methods of selecting J_m, prove or disprove the that resulting greedy algorithms produce feasible schedules, if they exist for the jobs being considered. Your proof of correctness must use an exchange argument.

   EDF Algorithm: Among those jobs J_i such that $r_i \leq t$, and that have not been scheduled for enough time units, pick J_m to be the job with minimal deadline. Ties may be broken arbitrarily.

   SRPT Algorithm: Among those jobs such that $r_i \leq t$, and that have not been scheduled for enough time units, pick J_m to be the job that has minimal remaining processing time. Ties may be broken arbitrarily. If a job have been executed for y time units before time t, then its *remaining processing time* is $x_i - y$. We call this algorithm SRPT.

   As an example of EDF and SRPT consider the following instance: $J_1 = (0, 100, 1000)$, $J_2 = (10, 10, 100)$, $J_3 = (10, 10, 101)$, and $J_3 = (115, 10, 130)$.

   EDF runs J_1 from time 0 through time 10. From time 10 until time 20, EDF runs J_2 because J_2's deadline, 100, is less than J_1's deadline, 1000, and less than J_3's deadline, 101. From time 20 until time 30, EDF runs J_3 because J_3's deadline, 101, is less than J_1's deadline, 1000, From time 30 to time 115, EDF runs J_1. From time 115 to time 125, EDF runs J_4 since J_4's deadline, 130, is less than J_1's deadline, 1000. EDF then runs J_1 from time 125 to time 130. Thus for this input, EDF finishes all the jobs by their deadline.

   SRPT runs J_1 from time 0 through time 10. From time 10 until time 30, SRPT runs J_2 and J_3 (either can go first) because through out this time period, J_2 and J_3's remaining processing times are always equal and less than J_1's remaining processing time, 90. From time 30 to time 115, SRPT runs J_1. From time 115 to time 120, SRPT runs J_1 since J_1's remaining processing time throughout this period is smaller than J_4's remaining processing time, 10. From time 120 to 130, SRPT runs J_4. Thus for this input, SRPT finishes all the jobs by their deadline.

   For each of EDF and SRPT, prove or disprove that the algorithm is correct. Proofs of correctness must use an exchange argument, and must explain why an exchange argument is sufficient to conclude that the algorithm is correct.

2. (20 points) Consider the problem where the input is n keys $K_1, \ldots, K_n$, where $K_1 < K_2 < \ldots < K_n$, and n associated probabilities $p_1, \ldots, p_n$, where $\sum_{i=1}^{n} p_i = 1$. Here p_i is the probability that key K_i is accessed. The output should be the minimum possible expected access time for binary search tree T on these keys. The expected access time for a tree T is the sum over the keys of the depth of the key in T times the probability that key is accessed, $\sum_{i=1}^{n}(\text{depth of } K_i \text{ in } T) * (p_i)$. Assume that the depth of the root is 1.

   (a) Let $A(i, j)$ be the minimum possible expected access time. Give pseudo-code for a recursive algorithm that computes $A(i, j)$.

   (b) **Formally** show that the running time of your algorithm is at least exponential.

   (c) Give pseudo-code for a polynomial-time iterative array-based algorithm that computes $A(i, j)$.

   (d) Assume that you have 4 keys with probabilities $p_1 = 0.3$, $p_2 = 0.1$, $p_3 = 0.1$ and $p_4 = 0.5$. Fill in the last entry in following table using your algorithm. Show your work.

	j=1	j=2	j=3	j=4
i=1	0.3	0.5	0.6	
i=2		0.1	.3	0.8
i=3			0.1	0.7
i=4				0.5

Table 1: Table for $A(i, j)$

3. (20 points) The input to this problem is a sequence of n points $p_1, \ldots p_n$ in the Euclidean plane. You are to find the shortest routes for two taxis to service these requests in order. Let us be more specific. The two taxis start at the origin. If a taxi visits a point p_i before p_j then it must be the case that $i < j$. Each point must be visited by at least one of the two taxis. The cost of a routing is just the total distance traveled by the first taxi plus the total distance traveled by the second taxi. Design an $O(n^2)$ time dynamic programming algorithm to find the minimum cost of a routing. Note that you do not need to compute the actual routes of the taxis, just the cost.

4. (20 points) Assume that you are given a collection $g_1, \ldots, g_n$ of gems. Each gem's type is either ruby or emerald. You know the type of each gem. You know that the value of each gem is an integer between 1 and L, inclusive. However, you do not know the specific value of any gem. You also do not know the specific value of L.

   You are provided with an assayer. A assayer functions in the following manner. You can give the assayer any two disjoint sub-collections, say S_1 and S_2, of the gems. Let $v(S_1)$ and $v(S_2)$ be the cumulative values of the gems in S_1 and S_2, respectively. The assayer then determines for you whether $v(S_1) < v(S_2)$, $v(S_1) = v(S_2)$, or $v(S_1) > v(S_2)$.

   The problem is to determine if one can partition the gems into two disjoint sub-collections A and B such that:

   - $v(A) = v(B)$,
   - the number of rubies in A is equal to the number of rubies in B, and
   - the number of emeralds in A is equal to the number of rubies in B.

   Note that every gem has to be in exactly one of A or B.

   Give an algorithm for this problem that makes at most $O(n^4\ L)$ uses of the assayer. For partial credit, give an algorithm where the number of uses of the assayer is polynomial in n and L.