COMP 3711H Design and Analysis of Algorithms
2015 Fall Semester
Homework 2
Handed out: Oct 3
Due: Oct 16

Assignments should be submitted to the collection box located outside room 4210A in the lab area.

Problem 1. Let's consider a long, quiet country road with houses scattered very sparsely along it. (We can picture the road as a long line segment, with an eastern endpoint and a western endpoint.) Further, let's suppose that despite the bucolic setting, the residents of all these houses are avid cell phone users. You want to place cell phone base stations at certain points along the road, so that every house is within four miles of one of the base stations.

Design an efficient algorithm that achieves this goal, using as few base stations as possible. Justify the running time and correctness of your algorithm.

Problem 2. The wildly popular Spanish-language search engine El Goog needs to do a serious amount of computation every time it recompiles its index. Fortunately, the company has at its disposal a large supercomputer, together with an essentially unlimited supply of high-end PCs.

They've broken the overall computation into n distinct jobs, labeled $J_1, J_2, \ldots, J_n$, whoich can be performed completely independently of one another. Each job consists of two stages: first it needs to be *preprocessed* on the supercomputer, and then it needs to be *finished* on one of the PCs. Let's say that job J_i needs p_i seconds of time on the supercomputer, followed by f_i seconds of time on a PC.

Since there are at least n PCs available on the premises, the finishing of the jobs can be performed fully in parallel—all the jobs can be processed at the same time. However, the supercomputer can only work on a single job at a time, so the system managers need to work out an order in which to feed the jobs to the supercomputer. As soon as the first job in order is done on the supercomputer, it can be handed off to a PC for finishing: at that point in time a second job can be fed to the supercomputer; when the second job is done on the supercomputer, it can proceed to a PC regardless of whether or not the first job is done (since the PCs work in parallel); and so on.

Let's say that a *schedule* is an ordering of the jobs for the supercomputer, and the *completion time* of the schedule is the earliest time at which all jobs will have finished processing on the PCs. This is an important quantity to minimize, since it determines how rapidly El Goog can generate a new index.

Design a polynomial-time algorithm that finds a schedule with as small a completion time as possible. Justify the correctness of your algorithm.

Problem 3. A small business—say, a photocopying service with a single large machine—faces the following scheduling problem. Each morning they get a set of jobs from customers. They want to do the jobs on their single machine in an order that keeps their customers happiest. Customer i's job will take t_i time to complete. Given a schedule (i.e., an ordering of the jobs), let C_i denote the finishing time of job i. For example, if job j is the first to be done, we would have $C_j = t_j$; and if job j is done right after job i, we would have $C_j = C_i + t_j$. Each customer i also has a given weight w_i that represents his or her importance to the business. The happiness of customer i is expected to be dependent on the finishing time of i's job. So the company decides that they want to order the jobs to minimize the weighted sum of the completion times, $\sum_{i=1}^{n} w_i C_i$.

Design an efficient algorithm to solve this problem. That is, you are given a set of n jobs with a processing time t_i and a weight w_i for each job. You want to order the jobs so as to minimize the weighted sum of the completion times, $\sum_{i=1}^{n} w_i C_i$. Justify the correctness of your algorithm.

Problem 4. You have just been hired as the quality-control engineer for a company that makes coins. The coins must have identical weight. You are given a set of n coins, and are told that *at most one* (possibly none) of the n coins is either too heavy or too light (but you do not know which). Your task is to develop an efficient test procedure to determine which of the n coins is defective, or report that none is defective. To do this test you have a scale. For each measurement you place some of the coins on the left side of the scale and some of the coins on the right side. The scale indicates either (1) the left side is heavier, (2) the right side is heavier, or (3) both subsets have the same weight. It does not indicate how much heavier or lighter.

(a) Prove that in the worst-case the minimum number of measurements using the scale is at least $\lceil \log_3(1 + 2n) \rceil$. (Hint: Use a decision tree argument.)

(b) Present a method to determine the defective coin using at most $(\lceil \log_3(1 + 2n) \rceil + c)$ scale measurements, where c is a constant (independent of n). Try to make c as small as possible. Explain your algorithm's correctness. (If you cannot succeed in this, then try to get at least $c \lceil \log_3(1 + 2n) \rceil$, for a small constant c.)

To simplify your arguments, you may assume, for example, that $1 + 2n$ is a power of 3 or that n is a power of 3.

Problem 5. An array $A[1 \ldots n]$ is said to have a *majority element* if more than half of its entries are the same. Given an array, the task is to design an efficient algorithm to tell whether the array has a majority element, and, if so, to find that element. The elements of the array are not necessarily from some ordered domain like the integers, and so there can be no comparisons of the form "is $A[i] > A[j]$?". (Think of the array elements as GIF files, say.) However, you *can* answer questions of the form: "is $A[i] = A[j]$?" in constant time.

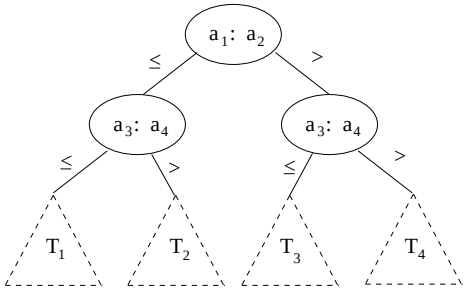
(a) Design an algorithm for solving this problem that runs in $O(n \log n)$ time, based on the following hint: First split the array A into two arrays A_1 and A_2 of half the size. Does knowing the majority elements of A_1 and A_2 help you figure out the majority element of A? If so, you can use a divide-and-conquer approach.

(b) Design a faster algorithm for this problem that runs in linear time, based on the following hint:

- Pair up the elements of A arbitrarily, to get $n/2$ pairs
- Look at each pair: if the two elements are different, discard both of them; if they are the same, keep just one of them

Argue that after this procedure there are at most $n/2$ elements left, and that they have a majority element if A does. Use this idea to design your $O(n)$ time algorithm.

Problem 6. The figure shows part of the decision tree for *merge-sort* operating on a list of 4 numbers, a_1, a_2, a_3, a_4. For this question, you need to expand subtree T_2, i.e., show all the internal (comparison) nodes and leaves in subtree T_2.



Problem 7. Consider the problem of merging two sorted lists containing n elements each. Any algorithm that uses only comparisons to solve this problem can be viewed as a decision tree.

(a) Find a lower bound on the number of leaves in this decision tree. Justify your answer.

(b) From your answer in part (a), compute a lower bound on the number of comparisons necessary in the worst case to merge the two sorted lists.

(c) Can you think of a different argument to improve the lower bound to $2n - 1$?

(*Hint:* You may use the following approximation to C_n^{2n}, the number of ways of choosing n items out of $2n$ items: $C_n^{2n} \sim 4^n/\sqrt{\pi n}$.)

Problem 8. Show that there is no comparison sort whose running time is linear for at least half of the $n!$ inputs of length n. What about a fraction $1/n$ of the inputs of length n? What about a fraction $1/2^n$?