15-451 Homework 3

Feb 18, 2008

March 6, 2008

Please hand in each problem on a separate sheet and put your **name** and **recitation** (time or letter) at the top of each page. You will be handing each problem into a separate box, and we will then give homeworks back in recitation.

Remember: written homeworks are to be done individually. You must try to solve the problems on your own. If you need to get help from others or from Google, this is acceptable provided that you acknowledge where the ideas came from and write the solution in your own words. You will not lose points for giving proper credit to where your ideas came from.

Direct questions to Bryant Lee (bryantl@cs.cmu.edu)

1 Question 1

Hashing. As discussed in class, the notion of universal hashing gives us guarantees that hold for arbitrary (i.e., worst-case) sets S, in expectation over our choice of hash function. In this problem, you will work out what some of these guarantees are.

1. Describe an explicit universal hash function family from $U = \{0, 1, 2, 3, 4, 5, 6, 7\}$ to $\{0, 1\}$. Hint: you can do this with a set of 4 functions.

2. Let H be a universal family of hash functions from some universe U into a table of size m. Let S subset of U be a set of m elements we wish to hash. Prove that if we choose h from H at random, the expected number of pairs (x, y) in S that collide is $<= (m-1)/2$.

3. Prove that with probability at least $3/4$, no bin gets more than $1+2\sqrt(m)$ elements. Hint: use part (2). To solve this question, you should use "Markov's inequality". Markov's inequality is a fancy name for a pretty obvious fact: if you have a non-negative random variable X with expectation $E[X]$, then for any $k > 0$, $Pr(X > kE[X]) <= 1/k$. For instance, the chance that X is more that 100 times its expectation is at most $1/100$. You can see that this has to be true just from the definition of expectation.

2 Question 2

In the carnival game of Whack a mole, there are n moles numbered $1, 2, ..., n$ all in a line. Neighboring moles are distance 1 apart. Each mole sits in a hole. Each second, one of the moles sticks its head out of its hole. You then have 1 second to whack the mole with a (cartoonish) mallet. If you are successful you get some points.

Let's say you know in advance the sequence of moles that will be popping up. Let m_t denote the mole that will pop up at time t, $1 <= t <= T$. If you successfully whack that mole you get p_t points. Also, the mallet is heavy, so that in 1 second you can only move the mallet a distance of 1. Thus the sequence of choices of what to whack must have the property that two whacks that are k apart must occur at times that differ by at least k. You are allowed to pick any desired starting point for the mallet.

For example, if the moles will stick out their heads in the order $< 2, 3, 1 >$, then you can get $Max\{p_2 + p_3, p_2 + p_1\}$ but cannot get $p_2 + p_3 + p_1$. Give an algorithm to efficiently compute the optimal sequence of moles to hit. The output should be a trajectory for the mallet - that is, an array indicating the position x_t of the mallet at time t. If $x_t = m_t$ then this means that we hit the mole at time t, and get p_t points. Otherwise we get 0 points at time t. Write-up your algorithm formally in pseudo-code. Try to make your algorithm as efficient as possible. Its running time should be polynomial in n and T.

3 Question 3

The number of combinations of n things taken m at a time, $\binom{n}{m}$, can be computed using the following recurrence:

$$\binom{n}{m} = \binom{n-1}{m} + \binom{n-1}{m-1} \text{ for } 0 < m < n$$

and

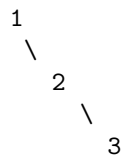
$$\binom{n}{0} = \binom{n}{n} = 1$$

1. Write a recursive algorithm to compute $\binom{n}{m}$ using the above recurrence.
2. Analyze the worst case running time of your algorithm as a function of m and n.
3. Produce a memoized version of your algorithm. Note: recall that a memoized algorithm is a recursive algorithm that stores information so that sub-problems do not have to be recomputed.
4. Give a dynamic programming algorithm to compute $\binom{n}{m}$.
5. Analyze the running time of your dynamic programming algorithm as a function of m and n.

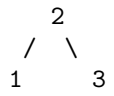
4 Question 4

BSTs and dynamic programming. Consider a binary search tree storing a set of keys $x_1 < x_2 < x_3 < ... < x_n$. Lets define the cost of handling a request for some key to be the number of comparisons made in searching for it (1 plus the distance of the node from the root of the tree). For example, if the root is requested, the cost is 1. Given a particular sequence of requests, one can calculate the cost that would be incurred on that sequence by different possible binary search trees. The tree that attains the minimum cost is called the optimal binary search tree for that sequence.

For example, for requests $< 1, 2, 1, 3, 1 >$ in this order, the tree



costs 8, whereas the tree



costs 9.

1. For a fixed tree, the cost of a given sequence of requests clearly only depends on the number of times each key is requested, not on their order. Suppose that $n = 4$ and that x_1 is accessed once, x_2 is accessed 9 times, x_3 is accessed 5 times, and x_4 is accessed 6 times. Find an optimal binary tree for this set of requests. (There is more than one possible answer.)

2. In general, suppose the optimal binary search tree for a given set of requests has x_i at the root, with L as its left subtree and R as its right subtree. Prove that L must be an optimal binary search tree for the requests to elements $x_1, ..., x_{i1}$ and R must be an optimal binary search tree for the requests to elements $x_{i+1}, ..., x_n$.

3. Give a general algorithm for constructing the optimal binary tree given a sequence of counts $c_1, c_2, ..., c_n$ (c_i is the number of times x_i is accessed). The running time of your algorithm should be $O(n^3)$. Hint: use dynamic programming.

Note #1: the notion of an optimal binary search tree is a lot like the notion of a Huffman tree, except that we also require the keys to be in search-tree order. This requirement is the reason that the greedy Huffman-tree algorithm doesnt work for finding optimal BSTs.

Note #2: its actually possible to improve the running time to $O(n^2)$ by a simple modification to this dynamic-programming solution. But proving correctness for this faster version is very tricky.