CMPS 102: Homework #3

Due on Tuesday, April 21st, 2015

John Allard 1437547

Problem 1

Hadamard matrices $H_0, H_1, H_2, \ldots$ are defined as follows:

- H_0 is the 1×1 matrix [1]
- For $k > 0, H_k$ is the $2^k \times 2^k$ matrix

$$H_k = \begin{bmatrix} H_{k-1} & H_{k-1} \\ H_{k-1} & -H_{k-1} \end{bmatrix}$$

Show by induction that the dot product between any two distinct rows of such a Hadamard matrix is zero.

Problem 2

Consider the Coin Changing problem with following set of coins:

$$\{1, k, k^2, ..., k^n\}.$$

Prove that the Cashier's Algorithm is optimal for all amounts $x \in \mathbb{N}$ given the above set of coins for any $k, n \in \mathbb{N}$ s.t. $k \geq 2$, $n \geq 0$.

I will start by showing the maximum number of times that each coin can appear in the optimal solution, then I will use this information to prove that the greedy algorithm will produce the optimal solution.

An optimal solution on $k \geq 2$, $n \geq 0$ requires that each coin (except the highest denomination, $c_n = \$k^n$) is present at most $k-1$ times. Proof : Take a coin c_m of value k^m for $m \geq 2$. If we were to have $k + \alpha$ for $\alpha \geq 0$ c_m coins in our solutions, these coins would account for $(k+\alpha)(k^m) = k^{m+1} + \alpha k$ units of currency. But, (assuming $m + 1 \leq n$), we could just replace the k c_m coins with a single c_{m+1} coin, because k c_m coins is $k * k^m = k^{m+1}$ units of currency. Because $k \geq 2$, replacing k coins with a single coin of higher-value will reduce the total number of coins by at least one. Thus any optimal solution, which by definition is the lowest number of coins which add up to the given total, cannot have more than $k-1$ of each coin denomination (except the highest) else we could remove one or more coins and create an even better solution.

Now that I have established that each (except the highest value) coin can only appear at most $k-1$ times in our final solution, I will attempt to prove that the cashiers greedy algorithm will find the optimal solution. To start, choose an $m \in \mathbb{N}$ arbitrarily, and let $k \geq 2$ also be chosen arbitrarily. We use a coin-space consisting of $m+1$ different coin values, $\{1, k, k^2, \ldots, k^m\}$, which will form the individual components of our solution. Pick a value $t : 1 \leq t \leq m$, and assume that $\forall y : 1 \leq y < k^t$, that the algorithm produces the optimal solution. Now choose a value of $x : k^t \leq x < k^{t+1}$. In this situation, the greedy algorithm will choose to insert the coin c_t with value k^t into our solution set, because it is the largest value coin that is less than or equal to the value x. I wish to show that the optimal solution will also contain this coin. The information revealed in the table below is used to construct this proof. It is inspired from the example in the slides.

p	k^p	restrictions	max val of $c_0, c_1, \ldots, c_{p-1}$
0	1	$c_o \leq k-1$	/
1	k	$c_1 \leq k-1$	$1 * (k-1)$
2	k^2	$c_2 \leq k-1$	$1 * (k-1) + k * (k-1) = k^2 - 1$
3	k^3	$c_3 \leq k-1$	$1 * (k-1) + k * (k-1) + k^2 * (k-1) = k^3 - 1$
		....	...
n	k^n	none	$(k-1) * \sum_{i=0}^{n-1} k^i$

Problem 2 continued on next page...

I will concentrate on the general term. For any given coin c_n, the maximum combined values of all coins less than that coin, M_n is defined as :

$$M_n = (k-1) * \sum_{i=0}^{n} k^i$$

This is of course a geometric series with common ratio k, which reduces to the following equation.

$$M_n = (k-1) * \frac{(1-k^n)}{1-k}$$

$$M_n = \frac{(k-1)}{(1-k)} * 1 - k^n$$

$$M_n = -1 * (1-k^n)$$

$$M_n = k^n - 1$$

Thus, we claimed that given a value $x : k^t \leq x < k^{t+1}$, the greedy algorithm will select the coin c_t with a value of k_t and that any optimal solution will also contain this coin. For this to be true, there must be no other way to creating a value of x with coins that are of a value less than k^t. The proof above showed that given a coin c_n, the max combined value (M_n) of all coins less than that coin is given by the equation $M_n = k^n - 1$. This is strictly less than k^n, which is the value of the coin c_n. Thus there is no valid way to construct the value x by selecting a coin other than c_t, which means the optimal solution will also contain c_t.

We now recurse on a problem of size $x - c_t$, if this value is still greater than c_t, the greedy algorithm will select another c_t coin, and the argument above again applies, confirming that the optimal solution will also contain another c_t coin. This process can repeat at most $k-1$ times (we can have at most that many of each coin), before we will be at a value $x < k^t$, so the inductive hypothesis applies and we infer that the optimal solution will be produced by our greedy algorithm.

Problem 3

You are given an unbounded array $A[1], A[2], A[3], \ldots]$ containing distinct integers sorted in ascending order. Describe an efficient algorithm that takes an integer k as input and finds out whether k is in the array in time $O(\log p)$ time where p is the number of integers in the array that are strictly less than k.

Since the required runtime is $O(log(p))$, I know I am going to have to shrink the search space by a constant factor for each iteration. This will result in an exponentially growing search of the space, which will reduce the run-time needed to process search n elements from linear to logarithmic. To do this, I will use a technique inspired by the exponential backoff algorithm. Instead of looking at each element in the array from the strart to the end, I will look at every 2^nth item for $n \in \ltimes$. Once I find an element larger than the desired element, I can simply perform a binary search in between the last two elements I have looked at. Since the entire array is ordered, binary search will work correctly and the element will be found. This is discussed in detail below :

Problem 4

We define an array A of n objects has a *dominant* object if at least $\lfloor n/2 \rfloor + 1$ entries of A are identical. Our goal is to design an efficient algorithm to tell whether the array has a dominant object, and, if so, to find that object. Our only access to A is by making a *query* asking whether $A[i] = A[j]$ for any two $i, j \in \{1, 2, \ldots, n\}$.

Problem 4 continued on next page...

(a) Design an algorithm to solve this problem with $O(n \log n)$ queries. (**Hint:** Split the array A into two arrays of half the size.)

(b) Design an algorithm to solve this problem with $O(n)$ queries. (**Hint:** Don't Split. Pair up the elements arbitrarily and get rid of as many as you can, repeatedly.)

Problem 5

KT, problem 17, p 197.)
Consider the following variation on the Interval Scheduling Problem.You have a processor that can operate 24 hours a day, every day. People submit requests to run *daily jobs* on the processor. Each such job comes with a *start time* and an *end time*; if the job is accepted to run on the processor, it must run continuously, every day, for the period between its start and end times. (Note that certain jobs can begin before midnight and end after midnight; this makes for a type of situation different from what we saw in the Interval Scheduling Problem.)

Given a list of n such jobs, your goal is to accept as many jobs as possible (regardless of their length), subject to the constraint that the processor can run at most one job at any given point in time. Provide an algorithm to do this with a running time that is polynomial in n. You may assume for simplicity that no two jobs have the same start or end times.

Example. Consider the following four jobs, specified by (*start-time, end- time*) pairs.

(6 PM, 6 AM), (9 PM, 4 AM), (3 AM, 2 PM), (1 PM, 7 PM)

The optimal solution would be to pick the two jobs (9 P.M., 4 A.M.) and (1 P.M., 7 P.M.), which can be scheduled without overlapping.

Problem 6

(KT, problem 4, p 190)
Sorry too long to retype. Use a greedy algorithm. Reason your time bound as well as correctness.

Problem 7

EC: We discussed an algorithm in class that finds the k-th largest element in an array for $n \geq k$ elements in worst case time $O(n)$. This algorithm starts by finding the medians of groups of 5 elements and then finds the medians of the roughly $n/5$ medians.
Why is this algorithm based on groups of size 5? Does it also work with groups of size 3? Why or why not?