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-conquering search of the space, which will reduce the run-time needed to process 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 starting from the first element and continuing until we find the one we're looking for, 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. The algorithm is given below and they are discussed in detail below that. :

Problem 3 continued on next page...

```
// A = unbounded, sorted array on integers starting at index 1
// x is the element we are searching A for
1.   findx(A, x) :
2.        ind = 1
3.        if x = A[ind] :  // special case, first item in array
4.             return true
5.        else if x < A[ind] :  // if its less than element #1 it can't exist
6.             return false
7.        while A[ind] <= x :  // while we haven't passed p
8.             ind = ind*2 // double the index value
9.        result = BinarySearch(A, x, ind/2, ind)
10.        if result == -1 :  // wasn't found in the sub-array
11.             return false
12.        else :
13.             return true // was found in the sub-array

// A = sorted array, x is the element we are searching A for
// l is left index of subarray to search, r is right index
// returns index of x if found, -1 otherwise
1.   BinarySearch(A, x, l, r) :
2.        if r == l :
3.             if x == A[r] :
4.                  return r
5.             else :
6.                  return -1
7.        ind = (r+l)/2
8.        if x == A[ind] :  // we found it
9.             return ind
10.        else if x < A[ind] :  // while we haven't passed p
11.             return BinarySearch(A, x, l, ind)
12.        else if x > A[ind] :  // while we haven't passed p
13.             return BinarySearch(A, x, ind, r)
```

I've broken up my implementation into two main parts. The first part scans exponentially from the beginning of the array until it finds an element bigger than x, the element we are looking for. At this point, it knows that if the element does exist, it must exist in between the current value which is bigger than x and the last value it looked at which was smaller than x. Because the algorithm now knows which two indices the element must exist between (if it exists at all), we can defer the rest of the work to a simple binary search over the sub-array. If the binary search returns an index value, we now know the element exists and so we return true. If BinarySearch returns -1, then we know it wasn't in the subarray and thus doesn't exist at all in the full array. Proofs of correctness and run-tme are given below.

Proof of Correctness :
The correctness of this algorithm relies on the fact that the elements in the array are sorted in ascending order. Normally, given a sorted array, we would just perform a binary search to locate an element. That of course does not work for this problem because the array is unbounded, but we can reduce this problem to one of binary search with some trickery. The trick is, if we can find two elements, one of which is greater than the item we are looking for and the other of which is smaller, we can reduce this problem to a problem of performing a binary search over the subarray between those two elements. This is enumerated below.

Problem 3 continued on next page...

Proof : The following properties are used in the proof.

1. Let's say we are looking for an element x in a sorted array A. If we examine an element of A and notice it is smaller than x, than x must either be further along in the array or it must not exist in the array at all. This is obvious from the properties of a sorted array.

2. Likewise, if we're looking for an element x in a sorted array A, and we examine an arbitrary element of A and notice it is bigger than x, this means that x must either be earlier in the array or not exist at all.

This algorithm is called on an array A and with a value x that we are looking for. It starts by examining the first element in A, if it is x then we go ahead and return true. If the element is greater than x, we know that x can't exist in the array so we go ahead and return false. Those are the two special cases of this algorithm, the general case of the algorithm in handled next. We start by doubling the index, from 1 to 2. If the $A[2]$ element is greater than x, we exit the loop. If it less than x, we double the index and repeat the loop. This process of querying and doubling the qqury index repeats until we find some element greater than x, which because this is an unbounded increasing array and x is finite must eventually happen. When this does happen we exit the loop.
When the loop is exited, we have the index of the first element we have come across that is bigger than x. By (2) above, this means that x must either occur earlier in the array or it doesn't exist at all. Likewise, because in the previous iteration we examined the element $A[ind/2]$ and found that element to be less than x, by (1) above we know that x must exist at an index after $ind/2$ or it doesn't exist at all. Combining these two facts together means that x must either be between the indices $ind/2$ and ind in A, or it doens't exist at all. Thus we know we have reduced the search space of A to a finite number of elements that, if x exists, it must be apart of. This sets us up perfectly for binary search on the subarray $A[ind/2, ind]$, which is called on line 9 of the $findx$ function given above. This binary search now looks for the element x, returning its index if it is found and -1 otherwise. If it returns -1, we return false because this means x doesn't exist in the subarray and thus does not exist in A at all, or it is found and its index is returned, in which case we return true. The correctness of this algorithm depends on the correctness of binary search, which I'm assuming I don't have to prove. ///

Proof of Run-time Complexity :
The algorithm is required to run in $O(log(p))$, where p is the number of elements in A that are strictly less than the item we are searching for. Because this algorithm consists of two components which are executed in sequence, I will prove that their individual run-times are individually $O(log(p))$, which means that when run back-to-back their total run-time will also be $O(log(p))$. The first algorithm to be proved is the one that finds an element greater than the one we are looking for by checking an exponentially increasing index value in the array. For this I will use a direct proof.
Let x be an element you are searching for in a sorted, unbounded array A. Let p denote the number of elements in A with values strictly less than x. If x exists in A, it must be at index $p+1$, given the definition of p. This menas our algorithm needs to query an element at an index greater than or equal to $p+1$ in order to find an element that is greater than x. The algorithm starts searching at index 1, and doubles the index value before each subsequent query into the array. The number of operations to find an element larger than x is given below :

$$2^k = p + 1$$

$$k = \lg(p + 1)$$

$$k = \lg(p + 1) \leq \lg(p) + 1$$

$$k \leq \lg(p) + 1$$

$$k = O(log(p))$$

Thus k, the number of array queries needed to get to index $p+1$ which contains an element greater than x is $O(log(p))$, as required for the proof.

After we have found an element which is greater than x, we perform a binary search over a sub-array of A. Given that the element larger than x is at index j, we now call binary search on the subarray $A[j/2, j]$, inclusive of the end elements. Since we doubled our search index every iteration, and we stopped on the first element greater than x that we found, we know that j is no larger than $2p$, otherwise we would have found an element larger than x during the last iteration. We also know that $j/2 \geq 1$, since all index values we use are positive. This means that BinarySearch will be called on a subarray of size at-most $2p-1$. BinarySearch is logarithmic in the number of elements, so the run-time is $O(log(2p-1))$, which is of course $O(log(p))$.

Because both the algorithm to find an element larger than x and the BinarySearch algorithm to actually find x in a sub-array both run in time $O(log(p))$, the two run in sequence will run in time $O(log(p)) + O(log(p))$, which is of course just $O(log(p))$. ///

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 $i, j \in \{1, 2, \ldots, n\}$.

(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)

Problem 5 continued on next page...

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?