David Sun
CMPS 102
Homework 3

Problem 1:

1a) Show by induction that $(H_k)^2 = 2^k I_k$, where I_k is the identity matrix of dimension 2^k.

Base Case: $k = 0$. Then $H_0^2 = [1][1] = [1] = 2^0 \cdot I_0$.

Induction Step: Let $n \geq 0$ and $0 \leq n < k$. Suppose that $(H_n)^2 = 2^n I_n$, where I_n is the identity matrix of dimension 2^n.

Then $(H_k)^2 = \begin{bmatrix} H_{k-1} & H_{k-1} \\ H_{k-1} & -H_{k-1} \end{bmatrix} \begin{bmatrix} H_{k-1} & H_{k-1} \\ H_{k-1} & -H_{k-1} \end{bmatrix} = \begin{bmatrix} 2 \cdot H_{k-1}^2 & H_{k-1}^2 - H_{k-1}^2 \\ H_{k-1}^2 - H_{k-1}^2 & 2 \cdot H_{k-1}^2 \end{bmatrix}$

Let I_{k-1} denote the identity matrix of dimension 2^{k-1} and 0_{k-1} denote the 0 matrix of dimension 2^{k-1}. By the induction hypothesis, $2(H_{k-1})^2$ can be simplified as $2 \cdot 2^{k-1} I_{k-1}$. $H_{k-1}^2 - H_{k-1}^2$ can be simplified down to 0_{k-1} since the difference of any matrix with itself is the 0 matrix. Thus $(H_k)^2$ can be simplified as

$$\begin{bmatrix} 2 \cdot 2^{k-1} I_{k-1} & 0_{k-1} \\ 0_{k-1} & 2 \cdot 2^{k-1} I_{k-1} \end{bmatrix} = \begin{bmatrix} 2^k I_{k-1} & 0_{k-1} \\ 0_{k-1} & 2^k I_{k-1} \end{bmatrix} = 2^k \begin{bmatrix} I_{k-1} & 0_{k-1} \\ 0_{k-1} & I_{k-1} \end{bmatrix}$$

Notice that two identity matrices of dimensions $k-1 \times k-1$ lie within the main diagonal, and long the antidiagonal are two 0 matrices of dimensions $k-1 \times k-1$. This means that the matrix itself the matrix itself is the identity matrix with dimensions $2^k \times 2^k$.

Thus, $2^k \begin{bmatrix} I_{k-1} & 0_{k-1} \\ 0_{k-1} & I_{k-1} \end{bmatrix} = 2^k I_k$ as required.

1b) Note that Hadamard matrices are symmetric, i.e. $H_k = H_k^\top$. Thus by the above, $H_k H_k^\top = 2^k I_k$ as well. Use this fact for deriving a formula for the dot product between the i-th and j-th row of H_k, for $1 \leq i, j \leq 2^k$.

The dot product between the i-th and j-th is 0 whenever $i \neq j$ and the sum of the squares of all the matrix entries whenever $i = j$. In a Hadamard Matrix, the sum of the squares of all the entries for a $2^k \times 2^k$ would be 2^k sums of 1s. In other words, the dot product between any two rows i and j is defined by the following summation:

$$\text{Dot product between the } i\text{th and } j\text{th row of } H = \begin{cases} i \neq j & 0 \\ i = j & \sum_{j=1}^{2^k} H_{ij}^2 = \sum_{j=1}^{2^k} 1 = 2^k \end{cases}$$

Problem 2: Consider the Coin Changing problem with the European coin set:

$$\{1, 2, 5, 10, 20, 50, 100, 200\}.$$

Prove that the Cashier's Algorithm is optimal given the above set of coins. Use the same proof method that was used for the American coin set in class.

Problem 3: Given a sorted array of distinct integers $A[1, ..., n]$, you want to find out whether there is an index i for which $A[i] = i$. Give a divide-and-conquer algorithm that runs in time $O(\log n)$.

Algorithm: Our algorithm will be called **findIndex**$(A, low, high)$. It will be initially called in the following manner **findIndex**$(A, 1, N)$ where A is the array, and N is $length[A]$. The first step of the algorithm is to check if $low > high$. If such is the case, return -1 since there exists no i for which $A[i] = i$. Otherwise, let $mid = \lfloor \frac{low+high}{2} \rfloor$ and check if $A[mid] == mid$. If the condition holds, return mid. Otherwise, if $A[mid] > mid$ check the left sub-array for i by calling **findIndex**$(A, low, mid - 1)$. If $A[mid] < mid$ check the right sub-array for i by calling **findIndex**$(A, mid + 1, high)$.

Proof of correctness: If the low bound is greater than the higher bound, then there is no index in either the left sub-array or right sub-array in which $A[i] = i$, therefore it is appropriate to return -1 to indicate no possible index exists. However if the middle element is equal to the index, then we have found a satisfactory index and we return it. However, if the middle element is greater than the middle index, then there can exist no such index within the right sub-array since every number in the right sub-array must be at least $A[mid] - mid$ greater than its index. Therefore a potential index must lie within the left sub-array, so the algorithm recurs on the left. If the middle element is less than the middle index, then there can exist no such i in the left sub-array since is at least $mid - A[mid]$ less than the index. Therefore an appropriate index must lie within the right sub-array, so the algorithm recurs on the right.

Proof of runtime: By recurring on the left sub-array or the right sub-array, half the array is discarded. The algorithm also has a constant number of array element comparisons at each level, and this is done in constant time. Thus the recurrence can be written as $T(N) = 2T(N/2) + O(1)$. Using the master theorem, we can show that $T(N) = O(\log N)$