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 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 $1 \leq i, j \leq 2^k$.

Since $H_k H_k^\top = 2^k I_k$, finding the dot product between the ith and jth row is equivalent to finding the dot product between the ith and jth row is equivalent to finding the dot product between the ith and jth column of H_k. By the previous proof, we showed that $H_k H_k^\top = 2^k I_k$, thus the only time when dotting the ith row and jth column yields a non-zero entry is when $i = j$. More specifically dotting the ith row and jth column of H_k yields 2^k whenever $i = j$. In other words, the dot product between any two rows i and j is defined by the following summation:

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

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.

Suppose change is x and that $C_k \leq x < C_{k+1}$. Any optimal solution will take C_k. Otherwise there must be a sequence of $C_1 \dots C_{k-1}$ coins that add up to x. The table below indicates that if change is x, then there exists no possible way to make change from the set $C_1 \dots C_{k-1}$ without breaking the optimality condition. Thus, coin C_k must be taken and we can make change for $x - C_k$ by the same principle. The table sets the conditions for the optimal solution.

k	C_k	Optimal solutions must satisfy	Max value $(C_1 \dots C_{k-1})$
1	1	$C_1 \leq 1$	-
2	2	$C_1 + C_2 \leq 2$	1 $(1 \cdot C_1)$
3	5	$C_3 \leq 1$	4 $(2 \cdot C_2)$
4	10	$C_4 \leq 1$	9 $(C_3 + 2 \cdot C_2)$
5	20	$C_5 + C_4 \leq 2$	19 $(C_4 + C_3 + 2 \cdot C_2)$
6	50	$C_6 \leq 1$	49 $(2 \cdot C_5 + C_3 + 2 \cdot C_2)$
7	100	$C_7 \leq 1$	99 $(C_6 + 2 \cdot C_5 + C_4 + C_3 + 2 \cdot C_2)$
8	200	Unbounded	199 $(C_7 + C_6 + 2 \cdot C_5 + C_4 + C_3 + 2 \cdot C_2)$

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 $T(N) = O(\log N)$.

Problem 4: Suppose that you want to multiply the two polynomials $x+1$ and x^2+1 using the FFT.

(a) Choose an appropriate power of two (the FFT dimension), find the FFT of the two sequences, multiply the resulting sequence componentwise, and then compute the inverse FFT to get the coefficients of the product polynomial. Do the transforms by using matrix vector products as in problem 5 of the previous homework.

(b) Explicitly compute the product of your polynomial and check your result.

part a: Since the product of the two polynomials will result in an x^3 term, we want to find the fourth root of unity. In this case, our coefficient vectors will have size of 4×1 and our FFT matrix will be of size 4×4. Then our $\omega = e^{\frac{\pi i}{2}} = i$.

$1+x$ can be represented by the column vector $\begin{bmatrix}1\\1\\0\\0\end{bmatrix}$ and $1+x^2$ can be represented by $\begin{bmatrix}1\\0\\1\\0\end{bmatrix}$

Multiplying the coefficient vectors with the FFT matrix, we obtain

$$\begin{bmatrix}1 & 1 & 1 & 1\\ 1 & i & i^2 & i^3\\ 1 & i^2 & i^4 & i^6\\ 1 & i^3 & i^6 & i^9\end{bmatrix}\begin{bmatrix}1\\1\\0\\0\end{bmatrix} = \begin{bmatrix}2\\1+i\\1+i^2\\1+i^3\end{bmatrix} \text{ and } \begin{bmatrix}1 & 1 & 1 & 1\\ 1 & i & i^2 & i^3\\ 1 & i^2 & i^4 & i^6\\ 1 & i^3 & i^6 & i^9\end{bmatrix}\begin{bmatrix}1\\0\\1\\0\end{bmatrix} = \begin{bmatrix}2\\1+i^2\\1+i^4\\1+i^6\end{bmatrix}$$

Multiplying the two coefficient vectors component wise, we obtain

$$\begin{bmatrix}2\\1+i\\1+i^2\\1+i^3\end{bmatrix}\begin{bmatrix}2\\1+i^2\\1+i^4\\1+i^6\end{bmatrix} = \begin{bmatrix}4\\0\\0\\0\end{bmatrix}$$

We multiply this vector with the inverse FFT matrix to obtain the coefficient matrix for the product

$$\frac{1}{4}\begin{bmatrix}1 & 1 & 1 & 1\\ 1 & i & i^2 & i^3\\ 1 & i^2 & i^4 & i^6\\ 1 & i^3 & i^6 & i^9\end{bmatrix}\begin{bmatrix}4\\0\\0\\0\end{bmatrix} = \frac{1}{4}\begin{bmatrix}1 & 1 & 1 & 1\\ 1 & -i & -1 & i\\ 1 & -1 & 1 & -1\\ 1 & i & -1 & -i\end{bmatrix}\begin{bmatrix}4\\0\\0\\0\end{bmatrix} = \begin{bmatrix}1\\1\\1\\1\end{bmatrix}$$ Since we have the coefficient vector of the product, we can use it to find $C(x)$. In this case, $C(x) = 1+x+x^2+x^3$.

Part b: verifying correctness:

$A(x) = x^2 + 0 \cdot x + 1$

$B(x) = 0 \cdot x^2 + x + 1$

$$\begin{array}{rrrrrrr} & & x^2 & + & 0x & + & 1\\ \times & & 0x^2 & + & x & + & 1\\ \hline & & x^2 & + & 0x & + & 1\\ & x^3 & + & 0x^2 & + & x & \\ \hline & x^3 & + & x^2 & + & x & + & 1 \end{array}$$

Thus our FFT computation is accurate.

Problem 5: Given an array of positive integers $A[1, \ldots, n]$, and an integer $M > 0$, you want to partition the array into segments $A[1, \ldots, i_1],\ A[i_1 + 1, \ldots, i_2],\ \ldots,\ A[i_{k-1} + 1, \ldots, i_k]$, so that the sum of integers in every segment does not exceed M, while minimizing the number of segments k. You can assume that $M \geq A[i]$ for all i (which guarantees that such a partition exists). Design an $O(n)$ greedy algorithm for solving this problem and prove that it is optimal (i.e. that the obtained partition has the smallest possible number of segments).

Algorithm: This algorithm will return a set of containing pairs of indexes.

partition(A):
1) $sum = 0$
2) $begin = 1$
3) $partitions = \{\}$
4) for $i = 1$ to $length[A]$
5) if $sum + A[i] \leq M$
6) $sum \leftarrow sum + A[i]$
7) else
8) $partitions \leftarrow partitions \cup \{(begin,\ i - 1)\}$
9) $begin \leftarrow i$
10) $sum \leftarrow 0$
11) end for
13) $partitions \leftarrow partitions \cup \{(begin,\ length[A])\}$
14) return *partitions*

Explanation: Make one scan through the array. If the current array element added to the previous sum does not exceed M, add it to the sum and keep scanning. Otherwise, append the pair of the begin index and one less than the current index to the set of partitions, reset the sum and begin counter to 0 and keep scanning. At the end, return the set of partitions.

Proof solution is optimal: Assume towards a contradiction that our greedy algorithm splits the array into n partitions and there exists a more optimal solution that splits the array into fewer than n partitions. Since the greedy algorithm takes in the maximum possible sum without exceeding the limit M, for any arbitrary $(i, j) \in partitions_{greedy}$ created by the greedy algorithm, $\sum_{n=i}^{j} A[n] \leq M$, and $\sum_{n=i}^{j+1} A[n] > M$, where $1 \leq i, j < length[A]$. Since the more optimal solution partitions A into fewer than n partitions, there exists at least one $(l, k) \in partitions_{optimal}$ such that $A[l \ldots k]$ contain more elements than any partition generated by the greedy algorithm. Since such is the case $\sum_{n=l}^{k} A[n] > M$. However since the sum of the elements within this partition exceeds M, we have arrived at a contradiction, thus showing that our greedy algorithm indeed creates the most optimal number of partitions.