CSE 101 Homework 4

Winter 2022

This homework is due on gradescope Friday February 18th at 11:59pm on gradescope. Remember to justify your work even if the problem does not explicitly say so. Writing your solutions in \LaTeX is recommend though not required.

**Question 1** (Hadamard Matrices, 25 points). The $n^{th}$ Hadamard matrix is a $2^n \times 2^n$ matrix of some importance. It can be defined recursively by

$$H_0 = [1], \quad H_{n+1} = \begin{bmatrix} H_n & H_n \\ H_n & -H_n \end{bmatrix}.$$

Give an algorithm that given a $2^n$-dimensional vector $v$ computes $H_n v$ in time $O(n^2)$. 

*Note: Given that $N = 2^n$ is the dimension of $H_n$ (and the size of the problem), this algorithm runs in nearly linear $O(N \log(N))$ time. Also, this is a reasonable analogue of the Fast Fourier Transform algorithm with some similar applications.*

**Solution 1.** We are given a matrix $H_n$ which is a $2^n \times 2^n$ matrix and a $2^n$ dimensional vector $v$. We are asked to compute the matrix-vector product $H_n \cdot v$. We solve this using a divide and conquer strategy. We can write the vector $v = (v_1, v_2)$, where $|v_1| = |v_2| = |v|/2$. Then,

$$H_n \cdot v = \begin{bmatrix} H_{n-1}v_1 + H_{n-1}v_2 \\ H_{n-1}v_1 - H_{n-1}v_2 \end{bmatrix}.$$

This implies that we just need to recursively compute $H_{n-1}v_1$ and $H_{n-1}v_2$ and then combining all these values by adding them up we can arrive at $H_n \cdot v$. The base case of this algorithm would be when $|v| = 1$, where we just return the $H_0 \cdot v$, which is $v$.

A concise pseudocode for the steps mentioned is given below:

```plaintext
Compute_Product(v):
  If $|v| = 1$, then return $v$
  Split $v = (v_1, v_2)$ where $|v_1| = |v_2| = |v|/2$
  $x_1 \leftarrow Compute_Product(H_{n-1}v_1)$
  $x_2 \leftarrow Compute_Product(H_{n-1}v_2)$
  Return the vector $(x_1 + x_2, x_1 - x_2)$
```

**Time Complexity:** It is given that $N = 2^n$. We can see that we’re reducing the problem into 2 subproblems of same size at each recursive call and the addition of the vectors $x_1$ and $x_2$ happens in $O(N)$. Thus, the recurrence relation for this algorithm is:

$$T(N) = 2T(N/2) + O(N)$$

Using the Master’s theorem where $a = 2, b = 2, d = 1$, we get the time complexity as $O(N \log(N))$. The algorithm can be proved using induction by inducting on $N$.

**Question 2** (Bird Keeping, 30 points). Aiden has $n$ birds and $n$ cages to keep them in. Each cage has a positive real numbered size and each bird has a positive real size requirement. Aiden wants to assign as many
of his birds as possible each to its own cage so that the size requirement of each bird is no larger than the size of the cage it is assigned to. Give an algorithm that given the lists of cage sizes and bird size requirements determines the maximum number of birds that can be successfully assigned.

For full credit, your algorithm should run in time \( O(n \log(n)) \) or better.

Solution 2. A successful assignment \((b, c)\) of a bird to a cage is valid if it is a one-to-one mapping and if for that bird \(b\), the size requirement is no larger than the size of the cage \(c\). We keep count of each of these assignments and return that count once we reach the end of the list.

We sort the cages in increasing order of size and birds in increasing order of size requirement. Then, for each bird from smallest to largest, we assign it the smallest possible cage, discarding all cages with smaller capacity. We keep count of each of these assignments and return that count once we reach the end of the list of birds or cages. This returned value is the maximum number of birds that can be successfully assigned. This takes \( O(n \log n) \) time where \( n \) is the number of birds and cages.

Proof of Correctness: We claim that our algorithm outputs the maximum number of birds that can be successfully assigned and that each assignment is valid since the algorithm only assigns a bird to a cage if it is not assigned to any other bird and the size of the cage is greater than or equal to the size requirement of the bird. Let us denote the size of cage, \( c \) by \( S(c) \) and size requirement of a bird, \( b \), by \( R(b) \).

We will now show that the algorithm outputs an optimal assignment, i.e. maximum number of birds are assigned. We will use an exchange argument to prove that the greedy solution is correct. Let \( G = \{(b_1, c_1), ..., (b_p, c_p)\} \) be our greedy assignment (\(|G| = p\)) and let \( G_i \) denote the partial assignment for the first \( i \) birds. Let us assume that there exists an optimal assignment \( A = \{(b_a, c_a), ...,(b_{a_q}, c_{a_q})\} \) and \(|A| = q\).

We assume that the solution \( A \) is better than our greedy assignment, i.e. \(|A| \geq |G| \) or \( q \geq p \).

Let us consider a base case when there is only one bird or cage. Then the optimal and greedy solution would be the same as they check the if the size requirement of the bird for the cage. This proves the base case.

Say, the optimal assignment \( A \) extends \( G_i \), i.e. \( A \) and \( G \) have the same assignments till the \( i^{th} \) index. We will now show that there exists another optimal assignment \( A' \) which does extend \( G_{i+1} \), i.e. \( A' \) which makes the greedy choices is as good as any solution. We do this by analyzing the following cases:

- **\( A_{i+1} \) does not use \( b_{g_{i+1}} \) (\( i + 1^{th} \) lightest bird):**
  - **\( A_{i+1} \) does not use \( c_{g_{i+1}} \):** It could be that \( A_{i+1} \) is only of size \( i \). So, we then add \( g_{i+1} \) to \( A_{i+1} \) which makes the new solution \( A' \) better than \( A \), i.e. \(|A'| > |A|\).
  - **If \( A_{i+1} \) does use \( c_{g_{i+1}} \):** We claim that we can swap the bird \( b_{g_{i+1}} \) with the bird paired with \( c_{g_{i+1}} \) in \( A \) and this allows us to say that \( A' \) is as good as \( A \). Let \( a_{i'} \) be the bird paired with \( c_{g_{i+1}} \) where \( i' > i \). The reason we are able to perform the exchange is because we know that based on our greedy algorithm, \( b_{g_{i+1}} \) is the \( i + 1^{th} \) lightest bird which means \( R(b_{g_{i+1}}) \leq R(b_{a_{i'}}) \). And as \( (b_{a_{i'}}, c_{g_{i+1}}) \) is a valid pairing, this implies that \( S(c_{g_{i+1}}) \geq R(b_{g_{i+1}}) \). Thus, the new assignment solution \( A' \) is as good as \( A \).

- **Both \( G \) and \( A \) use the \( b_{g_{i+1}} \) (\( i + 1^{th} \) lightest bird)**

  We can say that \( b_{g_{i+1}} = a_{i+1} \) (by reordering \( A' \)'s pairs if necessary).

  - **\( A_{i+1} \) does not use \( c_{a_{i+1}} \):** We claim here that we can swap \( c_{a_{i+1}} \) with \( c_{g_{i+1}} \). We know that \( R(b_{g_{i+1}}) \geq S(c_{a_{i+1}}) \) and \( R(b_{a_{i'}}) \leq S(c_{a_{i+1}}) \). The way our greedy algorithm works, we sort both the list of cages and birds. Thus, we know that \( S(c_{g_{i+1}}) \leq S(c_{a_{i+1}}) \). Thus, we can swap \( c_{a_{i+1}} \) with \( c_{g_{i+1}} \) and the new solution \( A' \) we get is as good as \( A \).
  
  - **If \( A \) does use \( c_{a_{i+1}} \):** Here, \( A \) uses \( c_{a_{i+1}} \), but it is paired with some bird \( a_{i'} \) where \( i' > i \). The pairings we have are \( (b_{g_{i+1}}, c_{a_{i+1}}) \) and \( (b_{a_{i'}}, c_{g_{i+1}}) \). We claim that we can swap \( c_{a_{i+1}} \) with \( c_{g_{i+1}} \). We know that \( R(b_{a_{i'}}) \geq S(c_{a_{i+1}}) \) and we also know because of how our greedy algorithm works that \( S(c_{g_{i+1}}) \leq S(c_{a_{i+1}}) \). We can also say that \( R(b_{a_{i'}}) \leq S(c_{a_{i+1}}) \). This means that we can swap \( c_{a_{i+1}} \) with \( c_{g_{i+1}} \). And thus, the modified solution \( A' \) is as good as \( A \).
Having proven that for each $i$, there is an optimal assignment extending $G_i$, we can say that there exists an optimal assignment $G_n$ which means there is an optimal solution $A'$ which has the same assignments as the greedy solution $G$. Thus, $G$ is an optimal assignment.

**Question 3** (Least Recently Used, 45 points). Although the Furthest In The Future protocol is guaranteed to produce the optimal answer in the caching problem, it has the issue that implementing it requires that one know the entire sequence of memory lookups ahead of time, which is often not the case. An often more practical protocol is the Least Recently Used (LRU) system, whereby when there is a cache miss, the new item replaces the item currently in cache that was used least recently.

(a) Unfortunately, LRU can often be far from the optimal protocol. Show that for any integer $k$ and real number $\epsilon > 0$, there is a sequence $S$ of memory accesses so that if the LRU protocol is run on the sequence $S$ with a cache of size $k$, the number of cache misses is more than $(k-\epsilon)$ times as large as the optimal schedule for $S$. [15 points]

(b) However, LRU does have some nice properties. For example, suppose that LRU is being run on some sequence of memory accesses with a cache of size $k$. Furthermore, suppose that there is some time period during which our memory accesses only requests $k$ different memory locations. Show that during this time period that LRU makes at most $k$ cache misses. [15 points]

(c) Although LRU may be far from optimal, there is a sense in which it is competitive. Show that for any sequence $S$ of memory accesses and any positive integer $k$ that LRU run on $S$ with a cache of size $2k$ makes at most twice as many cache misses as the best protocol for executing $S$ on a cache of size only $k$. [15 points]

**Solution 3.** (a) Let $S = [s_1, s_2, ..., s_n]$ where $n \gg k$. If we consider sequences in $S$ of size at least $k+1$, for e.g. $[s_1, ..., s_k, s_{k+1}]$ repeated $x$ times, we can see that LRU will miss $(k+1)x$ times since it literally misses every single turn. But if we consider the optimal solution (FITF), After missing the first $k+1$ entries, it then misses every $k^{th}$ memory access. Thus, the FITF protocol approximately misses $x(k+1)/k$. To prove that the number of cache misses on LRU protocol on $S$ is more than $k-\epsilon$ times as large as the FITF schedule for $S$, we say that we can find some $x$ for $\epsilon > 0$ using the following expression.

\[
(k + 1)x > x(k + 1)(k - \epsilon)/k
\]

\[
k(k + 1)x > (k - \epsilon)(k + 1)(x)
\]

\[
x(k + 1) > 0
\]

Using the above expression, we can say that for some $\epsilon > 0$ there exists an $x$ such that the LRU protocol misses $k - \epsilon$ times as large as the the optimal FITF protocol.

(b) We know that there is a cache size of $k$ and that we’re considering some time period during which our memory accesses only requests $k$ different memory locations. Let $S_t \in S$ denote the sequence during that time period, where $|S_t| = k$. We prove this by contradiction, let use assume that in $S_t$, LRU makes greater than $k$ cache misses, i.e. there are at least $k+1$ cache misses. This would mean that two of them would need to be accessing the same element. But because of how the LRU protocol works, it would not throw away any element that was loaded in $S_t$. Thus, there can be only $|S_t|$ cache misses, which means our initial assumption was wrong.

(c) We can split the sequence $S$ into blocks of $2k$ distinct/unique elements where for each block $B_i$ has $2k$ elements. Let us say $S = [s_1, s_2, ..., s_n]$ where $n \gg k$. Say, we start from the end, i.e. $s_n$ and work backwards till we observe distinct $2k$ elements, let us call this block $B_{end}$. Similarly, we keep working backwards and forming similar blocks such that we have $B = [B_1, B_2, ..., B_{end}]$. For $B_1$, which does not necessarily have $2k$ distinct elements, both LRU and the optimal protocol (FITF) miss however many distinct elements are in this block. For some other block, $B_i \in B, i > 1$, in the worst case, LRU protocol with a cache size of $2k$ makes at most $2k$ cache misses per block, i.e. at most one per distinct memory access requested. We are given that the FITF has a cache size of $k$. FITF misses at least $k$ times in each
block. This happens because there are at least \( k \) elements in each block that weren’t in FITF’s cache at the start of the block and all these elements must be loaded in.

We can thus conclude that for a sequence \( S \) of memory accesses, LRU with a cache size of \( 2k \) makes at most twice as many cache misses as the best protocol for executing \( S \) on a cache size of only \( k \).

**Question 4** (Extra credit, 1 point). *Approximately how much time did you spend working on this homework?*