This homework is due Friday May 13th, 11pm 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). Textbook problem 2.28. This is a good algorithm to know as these matrices do show up and it’s important that you can multiply by them very quickly. Also the idea for this is similar to the idea for the Fast Fourier Transform.

Solution 1. For any column vector \( u \) of length \( n \), let \( u^{(1)} \) denote the column vector of length \( n/2 \) consisting of the first \( n/2 \) coordinates of \( u \). Similarly, define \( u^{(2)} \) to be the vector of the remaining coordinates.

Now, the multiplication of a matrix \( H_k \) and a column vector \( v \) is a column vector \( H_kv \). Note then that

\[
(H_kv)^{(1)} = H_{k-1}v^{(1)} + H_{k-1}v^{(2)} = H_{k-1}(v^{(1)} + v^{(2)})
\]

and

\[
(H_kv)^{(2)} = H_{k-1}v^{(1)} - H_{k-1}v^{(2)} = H_{k-1}(v^{(1)} - v^{(2)})
\]

This shows that we can find \( H_kv \) by calculating \( (v^{(1)} + v^{(2)}) \) and \( (v^{(1)} - v^{(2)}) \) and recursively computing \( H_{k-1}(v^{(1)} + v^{(2)}) \) and \( H_{k-1}(v^{(1)} - v^{(2)}) \).

Thus, we have divided the multiplication of a matrix of size \( (n \times n) \) and a vector of size \( n \) to two multiplications of a matrix of size \( (n/2 \times n/2) \) and a vector of size \( n/2 \).

Thus, the complexity analysis is:

\[
T(n) = 2T(n/2) + O(n)
\]

Which is \( T(n) = O(n \log(n)) \)

Question 2 (Online Caching, 30 points). In practice the furthest in the future algorithm is not usually used for making caching decisions, since while it’s optimal, it requires being able to look ahead and know future memory accesses in advance. In practice, one often needs an online algorithm, that is one that only knows about previous accesses when it makes the decision about what to eject at any given time. In practice, a slightly different greedy approach is often used. When a memory cell needs to be dropped from cache, you choose the cell that was used least recently. This is the least recently used (or LRU) algorithm.

(a) Show that there are some access sequences so that the LRU algorithm with a cache of size \( k \) has \( k \) times as many cache misses as the FITF algorithm. [10 points]

(b) Consider any sequence of accesses during which at most \( k \) distinct memory locations are addressed. Show that the LRU algorithm with cache size \( k \) makes at most \( k \) cache misses over such an interval. [10 points]

(c) Show that for any sequence of memory accesses that the number of cache misses obtained by the LRU algorithm with a cache of size \( 2k \) is at most twice the number of cache misses needed by the best offline algorithm using a cache of size \( k \). Hint: break the access sequence into time intervals in which exactly \( 2k \) distinct memory cells are accessed. [10 points]
Solution 2. (a) Let the cache pages be named by numbers. Let us assume that the cache of size $k$ already contains pages numbered $1, 2, \ldots, k$.

Now consider the sequence $k + 1, 1, 2, \ldots, k, k + 1, 2, \ldots, k, k + 1, \ldots$

The FIFT algorithm will evict the page numbered $k$ since it is the furthest. It would not encounter any page faults until the page numbered $k$ is requested at which point the page numbered $k - 1$ will be evicted. It is clear from the sequence that FIFT will encounter a page fault after every $k$ requests.

Now, the LRU algorithm would evict the page numbered $1$ since it is the least recently used page. The next page fault would occur at $1$ where $2$ would be evicted and so on. This algorithm would encounter a page fault after every request. Thus, LRU has $k$ times as many page faults as FIFT in this case.

(b) There are at most $k$ distinct memory locations which are addressed.

Claim: For each of the $k$ distinct memory locations, the total number of cache misses is at most one over any number of accesses.

Proof: Consider the first time the $i$th location is accessed. There will be a cache miss if the page to which the $i$th location belongs is not in the cache prior to the access. After the access, the $i$th location would be cached. Our claim is that the $i$th location page would never be evicted out of the cache in any sequence of subsequent accesses.

This can be proved by contradiction: Let the $i$th location be evicted from the cache for some access of the $j$th location. For this to happen, $i$th location is the least recently accessed location. Which means $k - 1$ distinct locations are in the cache and have been accessed after the $i$th location has been accessed. Note that the $j$th location is not in the cache yet since there has been a cache miss. This implies that there at least $k + 1$ distinct locations that are being accessed which is a contradiction.

(c) We us the hint: break the access sequence into time intervals in which exactly $2k$ distinct memory cells are accessed.

Claim: No matter what pages are in the cache prior to each time interval, for each time interval, the number of cache misses obtained by the LRU algorithm with a cache of size $2k$ is at most twice the number of cache misses needed by the best offline algorithm using a cache of size $k$.

Proof: For a time interval with $2k$ distinct memory cell accesses, the LRU algorithm makes at most $2k$ cache misses. This comes from Part (b). Now for a cache size of $k$, even the most optimal algorithm would have at least $k$ cache misses. This is because at most $k$ of the $2k$ distinct memory cells would have been in the cache at the beginning and the rest would have to loaded in the cache (with a cache miss) at some point in time.

Thus, with these arguments, it is clear that at each interval, the number of cache misses for LRU is at most twice. The same argument can be summed over all time intervals to say the same about the entire sequence.

Question 3 (Other Minimum Spanning Tree Algorithms, 45 points). Let $G$ be a graph whose edges are given distinct weights. For each of the following proposed greedy algorithms for computing a minimum spanning tree in $G$, either prove that the algorithm works, or provide an example of such a $G$ for which the algorithm obtains a wrong answer.

(a) Remove the heaviest edge whose removal does not disconnect $G$. Repeat until $G$ is a tree. [15 points]

(b) Pick to vertices $u, v$ in $G$. Add the shortest path between $u$ and $v$ to the tree, and glue all vertices on the path together. Repeat until $G$ has a single vertex. [15 points]

(c) For each vertex $v$, add the lightest edge incident to $v$ to the tree. Glue vertices along all such edges. Repeat until $G$ is a single vertex. [15 points]

Solution 3. (a) Claim: The algorithm produces a connected graph:

Proof: The algorithm only removes non-disconnecting edges
Claim: The algorithm produces a tree:
Proof: If at the end of the algorithm, if the connected graph is not a tree, then there has to be a cycle which means that removing one of the edges in that cycle will not disconnect the graph. Since the edge was not removed, we have a contradiction.

Claim: This algorithm produces a MST.
Proof:
Property 1: If $G$ has a cycle with a unique heaviest edge $e$, then $e$ cannot be part of any MST. To prove this, consider removing $e$ from the MST and adding another edge $e'$ belonging to the same cycle. Then we get a new tree with less total weight. To see why such an edge exists, if you remove the edge $e$ from the tree, you get a graph with two nonempty connected components $C_1$ and $C_2$. At least one of the other edges in the cycle must connect $C_1$ and $C_2$ (otherwise it wouldn't be a cycle).

Property 2: The edges removed by the above algorithm belong to at least one cycle where it is the heaviest edge. Proof: An edge is removed only when it does not disconnect $G$ which means it is a part of a cycle. Now let us assume that it is not the heaviest edge in the cycle. This means that a heavier edge belongs to the cycle. This is clearly a contradiction since the heavier edge would have been removed in the earlier iteration of the algorithm since it belongs to a cycle.

Since $G$ has unique edge weights, the above two properties together prove that only the edges which do not belong to any MSTs are being removed.

(b) This algorithm is incorrect. Consider the following graph: If we pick $A$ and $D$, the shortest path is through edge $A,D$. After combining $A$ and $D$, and repeating the algorithm, we get:

But the actual MST is:
(c) This algorithm will never produce cycles. For this, note that the heaviest edge in the cycle is not the lightest edge out of either vertex. Thus, a cycle will cause a contradiction.

The proof for why this algorithm produces a MST is similar to Prim’s algorithm. We use the cut property inductively. We prove that any instance of the algorithm, the set of edges $X$ that have been selected are a part of some spanning tree.

**Base Case:** $X = \emptyset$

**Induction Hypothesis:** Any instance of the algorithm where the set of edges $X$ has been selected. $X$ is a part of some MST

Now for each vertex in the graph, the lightest edge is selected. We can apply the cut property on each of these vertices to prove that all such edges $e$ are such that $X \cup \{e\}$ is a part of some MST. These edges have to be added one by one for the induction to work.