Problem 1, CS38 Set 3, Matt Lim

Our sequence of operations will consist of $2^k + 1$ **insert** operations and one **extract-min** operation. We will prove that this creates a Fibonacci heap of rank k. We will do this proof by inducting on k.

So, let our base case be $k = 1$. Then we have that we perform three **insert** operations and one **extract-min** operation. After three **insert** operations, our Fibonacci heap will consist of three root nodes. Then, after doing the single **extract-min** operation, we will have two root nodes that we must consolidate. To do this is simple; both have rank zero, so we just combine them to make one tree of rank one. So we have that $rank(H) = 1 = k = \log_2(2^1) = \log_2(2^k)$, where $rank(H)$ is as defined in the lecture slides. Thus our base case is satisfied.

Now onto our inductive assumption. We will assume that for all $1 < n < k$, a sequence of operations consisting of $2^n + 1$ **insert** operations and one **extract-min** operation creates a Fibonacci heap of rank n. Now we must show that for $n = k$, a sequence of operations consisting of $2^n + 1$ **insert** operations and one **extract-min** operation still creates a Fibonacci heap of rank n. So, let us begin. Due to our inductive assumption, we have that our sequence of operations works for $n = k - 1$. We also have that $2^k = 2 \cdot 2^{k-1}$. Let us consider when $n = k - 1$. Then we have that consolidating 2^{k-1} nodes results in a Fibonacci heap of rank $k - 1$. Now let us consider when $n = k$. We must show that consolidating 2^k root nodes (2^k because **extract-min** takes one away) results in a Fibonacci heap of rank k. Consider the fact that consolidating 2^k root nodes is the same as consolidating the first half of them, then the second half of them, then consolidating the results of the previous two consolidations. This is when consolidating as shown in the lecture slides, left to right. Half of 2^k is 2^{k-1}. So when we consolidate the first half of the 2^k root nodes, due to our inductive assumption, we get a Fibonacci heap of rank $k - 1$. We also get a Fibonacci heap of rank $k - 1$ when consolidating the second half. Then, we are left with two heaps/trees of the same rank. So we will combine them, which results in a single Fibonacci heap of rank k, since combining trees of the same rank adds one to $rank(H)$. Thus our final consolidated Fibonacci heap has rank k. Then, by induction, we can conclude that our sequence of operations works for all $n \geq 1$.

Problem 2, CS38 Set 3, Matt Lim

Here is our algorithm.

```
MERGESORT(A, p, r)
  1. if p < r
      2. q = ⌊(p + r)/2)⌋
      3. MERGESORT(A, p, q)
      4. MERGESORT(A, q + 1, r)
      5. MERGE(A, p, q, r)
```

MERGE is as seen on page 31 on CLRS. However, we will make one change that will count the number of inversions. Let us consider the merging of two sorted lists of integers, $A = a_1, \cdots, a_n$ and $B = b_1, \cdots, b_n$, where the A is the left split (lower indices) and the B is the right split (high indices). If, when we are merging and considering the elements a_i and b_j, we have that $a_i \leq b_j$ (line 13 on page 31), the merge proceeds as normal. If, however, $b_j < a_i$ (line 16 on page 31), then we have an inversion. In fact, we have $(n - i + 1)$ inversions (assuming i starts at 1). So we will increase the count of inversions by $(n - i + 1)$. This is the only change we will make (besides adding a counter variable starting at zero to count inversions).

Now to show that our algorithm is correct. Let us consider our algorithm when it is merging two sorted lists of integers, $A = a_1, \cdots, a_n$ and $B = b_1, \cdots, b_n$, where the A is the left split (lower indices) and the B is the right split (high indices). If were are considering the elements a_i and b_j and $a_i \leq b_j$, then we do not increase our count of inversions. This is because $i < j$ and $a_i \leq b_j$. We have that $i < j$ because of the way we split the lists. If we are merging two lists, one the left split, the other the right split, all the indices in the left split will be lower than all the indices in the right split. Now, if we have that $a_i > b_j$, then we increase our count of inversions by $(n - i + 1)$. We already established that all indices of the elements of A are less than all the indices of the elements of B. So if $a_i > b_j$, then we have an inversion for the pair (a_i, b_j). But we also have an inversion for the pairs $(a_{i+1}, b_j), \cdots, (a_n, b_j)$ as well, since if $a_i > b_j$ then $a_k > b_j$ for $k \geq i$, since our lists are sorted. So we can see that this algorithm correctly counts the number of inversions.

Now to prove time correctness. The only thing we added to the merge sort algorithm was increasing the number of inversions under a certain condition, which is constant time. Alternatively, we have that the time is

$$T(n) = 2 \cdot T(\frac{n}{2}) + O(n)$$

since we break the problem up into two subproblems of half the size, and merging takes linear time. Both viewpoints give us that $T(n) = O(n\log n)$. Thus our proof is complete.

Problem 3, CS38 Set 3, Matt Lim

We start with M_N is an $N \times N$ matrix, where $N = 2^n$. Our algorithm for multiplying M_N be a vector is as follows. We will divide our matrix M_N into four even quadrants (TL = top left, TR = top right, BL = bottom left, BR = bottom right). Notice that the TL, TR, and BL quadrants are all the same. We can see this by the following argument. We have that (a, b) entry equals $(-1)^{\sum_{i=1}^{n} a_i b_i}$. Then, when looking at the differences between TL and TR, the only one is that the first b_i's are changing. For example, if we have $00, 01, 10, 11$, we can see that the difference between the first and third, second and fourth is that their first bit is flipped. But since we multiply the first b_i by zero in the top quadrants, this difference doesn't matter, so the TL and TR are the same. We can see that the TL and BL quadrants are the same, since here we are only changing the first a_i's, which we multiply by zero. Now we must consider the BR quadrant. For this quadrant, both the first a_i and the first b_i flip. For example, if we are indexing the rows and columns by $00, 01, 10, 11$, we can see that the difference between the first and third, second and fourth is that their first bit is flipped (and this time the difference applies to the row indices and the column indices). This changes $\sum_{i=1}^{n} a_i b_i$ for every (a, b) by one. So the sign will be flipped. Now, notice that the TL quadrant is simply the $M_{N/2}$ matrix. So the TL, TR, and BL quadrants are all the $M_{N/2}$ matrix. Then we can see that the BR quadrant is simply the $-M_{N/2}$ matrix. Now we will consider the vector we are multiplying M_N by. Let us call it V. We will call the top half of V V_1, and the bottom half V_2. Then we have that multiplying M_N by V gives the same results as stacking the results of matrix multiplying $M_{N/2}$ by $V_1 + V_2$ and $M_{N/2}$ by $V_1 - V_2$. That is, the result is the following:

$$\begin{array}{|c|}\hline M_{N/2} \cdot (V_1 + V_2) \\ \hline M_{N/2} \cdot (V_1 - V_2) \\ \hline \end{array}$$

We will now show that this algorithm uses $O(N \log N)$ operations. We have that $T(N) = 2 \cdot T(N/2) + O(N) \implies T(N) = O(N \log N)$. The first part, $2 \cdot T(N/2)$, is because we are dividing the problem into two subproblems that are half the size of the original (we multiply two matrices of half the original size). The second part, $O(N)$, is the time to add two vectors and subtract two vectors of size order N. Then, by the lecture slides, this implies that $T(N) = O(N \log N)$. So we have proved the time bounds.

Note that for the following sections, we will index our matrix rows/columns starting at 1, not 0. Now we will show that this algorithm is correct. Let us call the result matrix for $M_N \cdot V$ A, the result matrix for $M_{N/2} \cdot (V_1 + V_2)$ B, and the result matrix for $M_{N/2} \cdot (V_1 - V_2)$ C. Let $M_N = (m_{ij}), M_{N/2} = (l_{ij}), A = (a_i), B = (b_i), C = (c_i), V = (v_i), (V_1 + V_2) = (\upsilon_i), (V_1 - V_2) = (\nu_i)$ (note the difference between normal v (v) and upsilon (υ)). Multiplying M_N and V using normal matrix multiplication, we get the following:

$$a_i = (mv)_i = \sum_{k=1}^{N} m_{ik} \cdot v_k$$

Now let us consider the matrix multiplication of $M_{N/2}$ with $V_1 + V_2$. Let $i \leq \frac{N}{2}$. Note $(l_{ik}) = (m_{ik})$ for the indices we are considering. We get the following:

$$b_i = (l\upsilon)_i = \sum_{k=1}^{N/2} l_{ik} \cdot (v_k + v_{k+\frac{N}{2}})$$

$$= \sum_{k=1}^{N/2} m_{ik} \cdot (v_k + v_{k+\frac{N}{2}})$$

$$= \sum_{k=1}^{N/2} m_{ik} \cdot v_k + m_{ik} \cdot v_{k+\frac{N}{2}}$$

$$= \sum_{k=1}^{N/2} m_{ik} \cdot v_k + \sum_{k=1}^{N/2} m_{ik} \cdot v_{k+\frac{N}{2}}$$

We have that, for $i \leq \frac{N}{2}$, where i denotes the rows in M_N, $m_{ik} = m_{i(k+\frac{N}{2})}$. Then it is clear that, continuing the previous continuities,

$$b_i = \sum_{k=1}^{N} m_{ik} \cdot v_k$$

for $i \leq \frac{N}{2}$.

Now we will consider the matrix multiplication of $M_{N/2}$ with $V_1 - V_2$. Let $i \leq \frac{N}{2}$ for now. We get the following:

$$c_i = (l\nu)_i = \sum_{k=1}^{N/2} l_{ik} \cdot (v_k - v_{k+\frac{N}{2}})$$

Then we can see that this equals the following when $i > \frac{N}{2}$ (we have $i > \frac{N}{2}$ only in the summation; the i in c_i must still be $\leq \frac{N}{2}$), since the TL and BL quadrants are equal.

$$= \sum_{k=1}^{N/2} m_{ik} \cdot (v_k - v_{k+\frac{N}{2}})$$

$$= \sum_{k=1}^{N/2} m_{ik} \cdot v_k - m_{ik} \cdot v_{k+\frac{N}{2}}$$

$$= \sum_{k=1}^{N/2} m_{ik} \cdot v_k - \sum_{k=1}^{N/2} m_{ik} \cdot v_{k+\frac{N}{2}}$$

We have that, for $i > \frac{N}{2}$, where i denotes the rows in M_N, $m_{ik} = -m_{i(k+\frac{N}{2})}$, for $k \leq \frac{N}{2}$. Then it is clear that, continuing the previous continuities,

$$c_i = \sum_{k=1}^{N} m_{ik} \cdot v_k$$

for $i > \frac{N}{2}$ in the summation and $i \leq \frac{N}{2}$ in c_i.

Then we have that the matrix

$$\begin{array}{|c|}\hline b_i = M_{N/2} \cdot (V_1 + V_2) \\ \hline c_i = M_{N/2} \cdot (V_1 - V_2) \\ \hline \end{array}$$

correctly computes that matrix multiplication of M_N with V.

Problem 4, CS38 Set 3, Matt Lim

Let $A = (a_{ij})$ and $B = (b_{ij})$ be $n \times n$ Toeplitz matrices, and let $C = (c_{ij})$ be the result of matrix multiplying A with B, $A \cdot B$. Then we will define a polynomial called P_A, which represents A. We will define it as $P_A(x) = \sum_{i=0}^{2n-2} a_i x^i$, where a_i is the ith value of the list of values that represents A. Then we will define polynomials P_{Bi} (i being the column of B), where $P_{Bi}(x) = \sum_{k=0}^{n-1} b_{ki} x^{n-1-k}$. So the coefficients of these polynomials are the values of the column vectors of B. Although the column vectors are not specifically given to us, we can see that the first column is the first n elements of the list (in reverse order), the second column is the n elements starting from the second element (in reverse order), and so on. So getting the column vectors from the list is $O(n^2)$. Now let us multiply $P_A(x) \cdot P_{Bi}(x)$. Let us call the result $P_R(x)$. Then we have that the middle n coefficients of this polynomial, with them ordered by descending power of x (e.g. $x^7 + x^6 + \ldots$), are exactly the values of the ith column of C. To clarify, of the middle n coefficients, the one with the highest power is the top value of the ith column of C, the one with the next highest power is the value below the top value of the ith column of C, and so on. We take the middle n values because all the other ones are not given by the actual matrix multiplication.

We will no show that this algorithm runs in time $O(n^2 \log n)$. This is fairly simple. For each column, we do polynomial multiplication, which takes $O(n \log n)$ time. And we need to generate n columns, so we do this n times. Thus this gives us time $O(n^2 \log n)$ (we can ignore the $O(n^2)$ from getting the column vectors b/c it is less).

Problem 5, CS38 Set 3, Matt Lim

(a) We will define

$$A_{i,j}(x) = \Pi_{k=i}^{j}(x - a_k)$$

We will then build a recursive tree. This tree will represent what we mod $f(x)$ by. The top of our tree will be $A_{0,n-1}$. Its left child will be $A_{0,\lfloor \frac{n-1}{2} \rfloor}$ and its right child will be $A_{\lfloor \frac{n-1}{2} \rfloor +1,n-1}$. Then this will continue until the bottom layer, which will consist of leaves $A_{0,0}, A_{1,1}, \dots, A_{n-1,n-1}$. So we start with $f(x)$, and mod it by the polynomials along the paths to the leaves. So, for example, following the leftmost path, we would do $f(x)$ mod $A_{0,n-1}$ mod $A_{0,\lfloor \frac{n-1}{2} \rfloor}$ mod $\dots$ mod $A_{0,0}$. By property (2) as given in the problem, this series of mods reduces down to $f(x)$ mod $A_{0,0} = f(x)$ mod $(x - a_0)$. Then by property (1), we have that $f(x)$ mod $(x - a_0) = f(a_0)$. So if we do this along all paths to all the leaves, we end up with $f(a_0), f(a_1), \dots, f(a_{n-1})$. The time for this is $T(n) = 2 \cdot T(\frac{n}{2}) + O(n\log n) = O(n\log^2 n)$. This is because each level we divide our problem into 2 subproblems of half the size. And multiplying/dividing/modding polynomials is time $O(n\log n)$. Note that for us to get the desired time bounds, when we are modding by polynomials, they must be fully expanded. So to fully expand the polynomials, we will do the following.

To fully expand the polynomials, we will use divide and conquer. This time, we will start at the bottom and build up. So at the bottom we will have all our $(x - a_k)$ factors as leaves. Then we will just build our tree up, multiplying the $(x - a_k)$ factors in pairs. This builds a binary tree with $\Pi_{k=0}^{n-1}(x - a_k)$ (but expanded) at the very top. The time to build this tree is $T(n) = 2 \cdot T(\frac{n}{2}) + O(n\log n) = O(n\log^2 n)$. We can see this by looking at the tree top down. Each level we divide our problem into 2 subproblems of half the size. And multiplying/dividing polynomials is time $O(n\log n)$. Then we can use this tree to assist us in the overall algorithm. So basically, in the above paragraph, when we are modding by polynomials, we will use the expanded versions contained in this tree (there is a direct correspondence with the values in the first tree you need to mod by and the values contained in the second tree).

Thus our overall time bound for this algorithm is $O(n\log^2 n)$ and our proof is complete.

(b) Our algorithm for this is the following. We will compute the polynomial

$$\Pi_{i=0}^{\sqrt{M}-1}(X - i)$$

at $\sqrt{M}$ positions. These positions are $\sqrt{M}, 2\sqrt{M}, \dots, \sqrt{M}\sqrt{M}$. Then we will multiply all the values we got which will result in $M!$ mod N.

We will now show that this algorithm uses $O(\sqrt{M}\log^2 M)$ operations. We can do this by simply using part (a), which tells us that evaluation a polynomial at n positions takes time $O(n\log^2 n)$. Here $n = \sqrt{M}$. And since the log kills the $\sqrt{M}$ (can just be brought out as $\frac{1}{2}$, a constant) we have that this algorithm is $O(\sqrt{M}\log^2 M)$.

Now we will show why this algorithm is correct. We have that for $X = \sqrt{M}$, the maximum value of the $(X - i)$ multiple is $\sqrt{M}$ and the minimum value is 1. Then we can see that values in between are just $2, 3, \dots, \sqrt{M} - 1$, since the values change by 1 for each new $(X - i)$. So this first product is $1 \cdot 2 \cdot \dots \cdot \sqrt{M}$. Then for $X = \sqrt{M}$, the values are incremented starting at $\sqrt{M} + 1$ and ending at $2\sqrt{M}$. So this product is $(\sqrt{M} + 1) \cdot (\sqrt{M} + 2) \cdot \dots \cdot (2\sqrt{M})$. This continues so on, and in the end we get that the overall product is just $1 \cdot 2 \cdot \dots \cdot \sqrt{M}\sqrt{M}$. But this is exactly $M!$. And since the problem states we are doing all computations modulo N, it is actually $M!$ mod N. Thus our algorithm works as desired.

(c) Our algorithm is as follows. We will do $GCD(M!, N) = GCD(M!$ mod $N, N)$. If the result is not one, then there is a nontrivial factor of N that is at most M.

We will now show that this algorithm uses $O(N^{1/4}\log^2 N)$ operations. We have that the number of bits of an integer N is $O(\log N)$. So the time the finding the GCD takes is $O(\log^2 N)$. The time it takes to do $M!$ mod N is given by part (b) to be $O(N^{1/4}\log^2 N)$, since we have that $M \leq \sqrt{N}$. So the overall time is $O(N^{1/4}\log^2 N)$.

Now we will explain why our algorithm works. This is very simple. Taking $GCD(M!$ mod $N, N)$ and seeing if it is not equal to one is the same thing as seeing if one of $GCD(2$ mod $N, N)$, $GCD(3$ mod $N, N), \dots, GCD(M$ mod $N, N) \equiv GCD(2, N), GCD(3, N), \dots, GCD(M, N)$ is not equal to one. In other words, it is the same as determining whether there is a nontrivial factor of N that is at most M. But this is exactly what we want it to do. So we have that our algorithm is correct.