Q1

Let n = # of items in array, m = length of the array

Let $\varphi(h) = 3n - 2m$

We know $\lceil 1.5n \rceil \geq 1.5n \geq m$ at all times so: $3n \geq 2m,\ 3n - 2m \geq 0$, thus $\varphi(h) \geq 0$

At n, m = 0 (initial value), $\varphi(h) = 0$

Case: n < m

$c_i = 1$

$\varphi(h_{i+1}) - \varphi(h_i) = (3(n+1) - 2m) - (3n - 2m)$

$= 3$

$T_A(append) = c_i + (\varphi(h_{i+1}) - \varphi(h_i)) = 1 + 3 = 4$

Thus $O(1)$ time

Case: n = m

$c_i = n + 1$

$\varphi(h_{i+1}) - \varphi(h_i) = (3(n+1) - 2\lceil 1.5n \rceil)) - (3n - 2n)$

$= 3 - 2\lceil 1.5n \rceil + 2n$

$c_i + (\varphi(h_{i+1}) - \varphi(h_i)) = n + 1 + 3 - 2\lceil 1.5n \rceil + 2n$

$= 3n - 2\lceil 1.5n \rceil + 4$

$\leq 3n - 2(1.5n) + 4$

$= 4$

Thus $O(1)$ time

Q2

a) Use a regular queue for FIFO behaviour and a doubly linked queue for getting min value.
b) Enqueue inserts the element into the regular queue as normal, for the doubly linked queue, if there are elements in the back that are greater than the element being inserted, pop them out of the back until the double queue is empty or the back element is less than the current one. Dequeue will remove from the regular queue as normal, and if the front element in the normal queue is the same as the front element in the

double queue, remove the front element from the double queue as well. Getting the min element is as simple as getting the front element in the double queue.

c) **Worst case cost**

Dequeue is 2=O(1) as normal, finding min-element in the list will be 1=O(1) because you can access the front element in the double queue in 1 step. Enqueue will be O(# items greater than the inserted element), which is at worst O(n) because of the case that all the items in the double queue must be popped (this happens when we insert an element that is smaller than all elements in the queue).

Amortized case cost

Let m = number of items in the queue, n = number of items in the double queue, p = number of items in the double queue greater than the element being inserted

Note that $m \geq n$ at all times, which leads us to the potential function:

Let $\varphi(h) = m - n + p^2$

<u>Enqueue:</u>

Potential method

Note that $0 \leq p \leq n$

$T_A(enqueue) = c_i + (\varphi(h_{i+1}) - \varphi(h_i)) = p + ((m+1) - ((n-p)+1)) - (m - n + p^2)$

$= p + p - p^2 = 0$

Thus O(1)

Aggregate method

After m insert operations, you can have at most m-1 pops from the double queue, which means is it O(2m-1)=O(m)

$Amortized\ complexity = \frac{O(m)}{m} = O(1)$

<u>Dequeue:</u>

Potential method

$T_A(dequeue) = c_i + (\varphi(h_{i+1}) - \varphi(h_i)) \leq 2 + ((m-1) - (n-1) + p^2) - (m - n + p^2) = 2$

Thus O(1)

Aggregate method

Actual cost: O(1)

$Amortized\ complexity = \frac{mO(1)}{m} = O(1)$

<u>Get min:</u>

Potential method

$T_A(get\_min) = c_i + (\varphi(h_{i+1}) - \varphi(h_i)) = m - n + p^2 - (m - n + p^2) = 0$

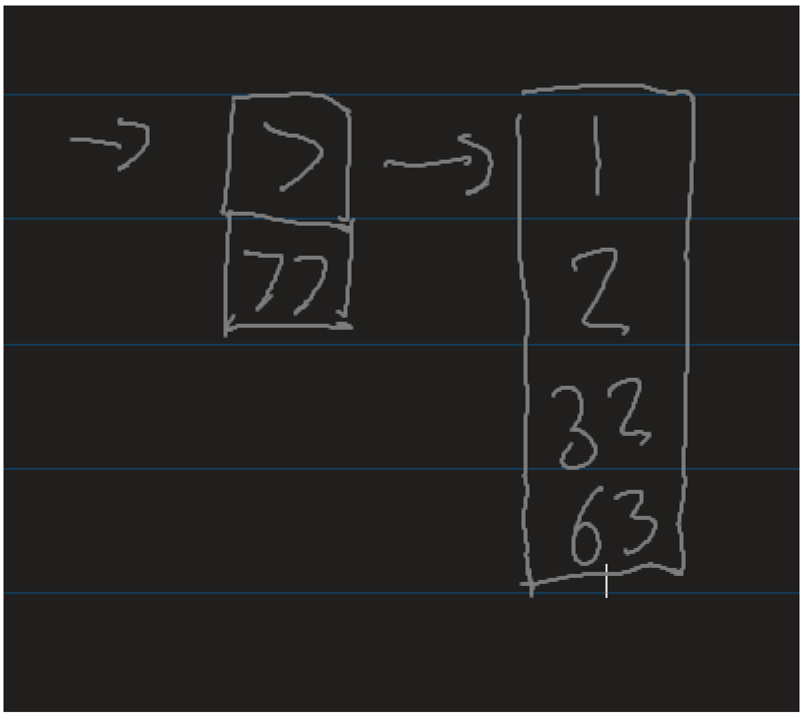
Aggregate method

Actual cost: O(1)

$Amortized\ complexity = \frac{mO(1)}{m} = O(1)$

<u>Q3</u>

a)



b) Let I_n denote the set of bit positions in the binary representation of n that contain the value 1. Position indices start at 0 with the least significant bit.

The maximum sum of elements in I_n is lg(n)

Using binary search on each array takes

$\Sigma_{i=0}^{lg(n)}(lg(2^i)) = lg(\Pi_{i=0}^{lg(n)} 2^i) = lg(2^{lg(n)+1}) = lg(n) + 1$

Thus the worst-case complexity is $O(log(n))$

c) Let I_n denote the set of bit positions in the binary representation of n that contain the value 1. Position indices start at 0 with the least significant bit.

The most array merges possible is $\lfloor lg(n) \rfloor$ if there exists an array of all sizes up to $2^{\lfloor lg(n) \rfloor}$

Merging arrays of size 2^i will cost 2^{i+1} because you need to sort the array, thus needing to compare each array's smallest items before inserting the smaller one, meaning 2^{i+1} comparisons at most.

So the max amount of comparisons will be:

$$\Sigma_{i=0}^{lg(n)}(2^{i+1}) = 2^{lg(n)} - 2 = n - 2$$

Thus the worst-case complexity will be $O(n)$

d) Actual cost: $O(n)$

Let n = number of inserts

For n inserts:

The max possible merges of arrays of length 2^i is $[log_{2^i}(n) - 1]$ for $i <= lg(n)$

Summing up the costs for all the inserts we get

$(lg(n))(worst\ complexity\ of\ insert) = lg(n) \times n = O(nlog(n))$

Using aggregate method:

$$Amortized\ cost = \frac{O(nlog(n))}{n} = O(log(n))$$

e) Actual cost: $O(n)$

<u>Pre-analysis</u>

The cost of inserting if there is no merge is 1

Let the cost of insert be the max possible number of arrays (lg(n)) and ignore the cost of merging arrays. When merging arrays, take the cost from the accumulated charges from initially inserting. There should be no more than $nlg(n)$ charges with n representing the cost of charge for merges of a certain length, and lg(n) representing the number of possible array lengths.

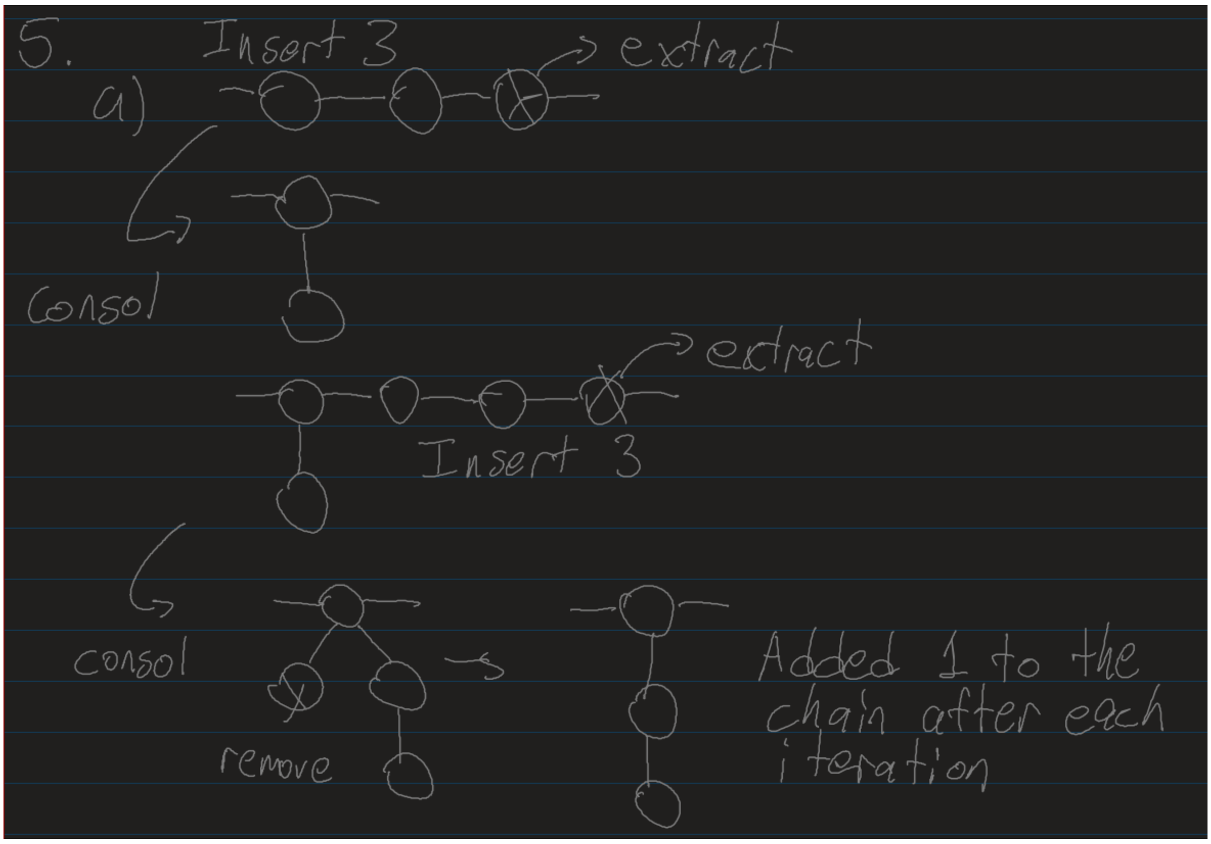
Thus $\frac{sum\ of\ charges}{inserts} = \frac{nlg(n)}{n} = lg(n) = O(logn)$

Q4

Iteration	Connection Components
0	{l}, {m}, {n}, {o}, {p}, {q}, {r}, {s}, {t}, {u}, {v}
1	{o, t}, {l}, {m}, {n}, {p}, {q}, {r}, {s}, {u}, {v}
2	{o, t}, {q, v}, {l}, {m}, {n}, {p}, {r}, {s}, {u}
3	{o, t, r}, {q, v}, {l}, {m}, {n}, {p}, {s}, {u}
4	{o, t, r, m}, {q, v}, {l}, {n}, {p}, {s}, {u}
5	{o, t, r, m}, {q, v}, {l, s}, {n}, {p}, {u}
6	{o, t, r, m, u}, {q, v}, {l, s}, {n}, {p}
7	{o, t, r, m, u, q, v}, {l, s}, {n}, {p}
8	{o, t, r, m, u, q, v}, {l, s}, {n}, {p}
9	{o, t, r, m, u, q, v}, {l, s}, {n}, {p}
10	{o, t, r, m, u, q, v}, {l, s}, {n}, {p}
11	{o, t, r, m, u, q, v}, {l, s, p}, {n}

Q5

a)



First, insert 3 and extract min to get a chain of length 2.

Repeat the process of inserting 3 nodes (one of which should be the lowest possible value), extract-min to consolidate, then remove the node attached to the root node of the chain that isn't part of the chain. Repeating this process always adds 1 node to the chain, thus if you repeat n-2 times the chain will be length n.

b)

```
ChainN():
        H = Make_Heap()
        for i in range(3):
                Insert(H, 0)
        Extract_min(H)
        // now we have a chain of length 2
```

```
i = 0
h_val = -1
while(i < n-2):
    Insert(H, -infinity)
    Insert(H, h_val)
    Insert(H, h_val - 1)
    Extract_min(H)
    Delete(H, i)
    i++
    h_val += 2
```

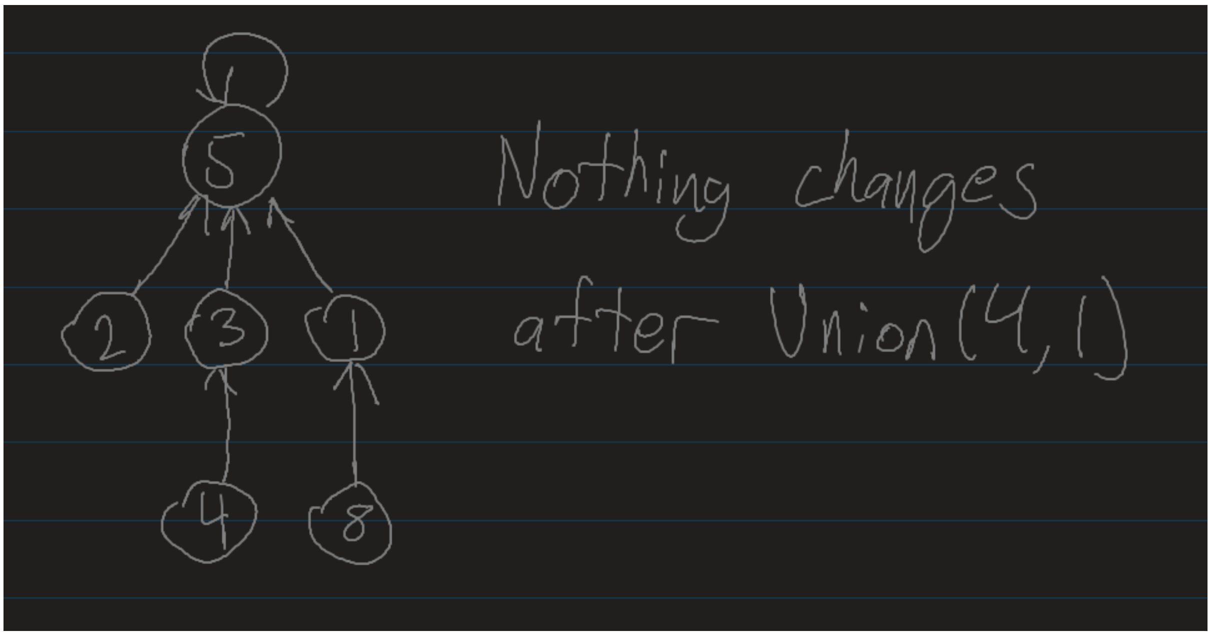
<u>Q6</u>

Use a list of children pointers along with the parent pointer.
This shouldn't affect the running time of union as you only need to update the parent's (i.e. y.children.add(x) or vice versa), this can be done during the normal union runtime: whenever you make some tree a child a another one, you would normally update the parent pointer, now you just need to also add the child pointer as well.

Now using the fact that all parents know their children, use the root pointer from the x of getset(x), we can perform anyu regular tree traversal and get all elements in the tree in O(n) time, with n being the number of elements in the set.

Q7

a)



b) **(and c)** Both implementations of makeset only access themselves thus both access value is 1

Findset is similar with path compression and head pointers: Li, Ti = 1

Thus the only differences in the number of nodes accessed could be in the union operation.

First, both of them access their representative to check weight/rank, but the difference lies in the item being attached to the other. In a list, the list with the smaller (or equal) amount of nodes will always be added to the other, requiring that all the head pointers in the smaller list be updated. However, in a tree, having equal or even lower rank does not mean having an equal amount of nodes. A tree of height 1 can have 2 or 3 nodes and the only thing deciding which tree it is attached to is the order in which they are inputted into the function. So it is entirely possible to have to attach a tree with more nodes to a tree with fewer, meaning TL < TT is entirely possible. But TT < TL is not, as the entire rest of the algorithm is identical in nodes accessed and the list implementation will always choose the lowest amount of nodes to append in a union operation.

<u>Q8</u>

We can use a different quicksort implementation with 3 partitions instead of 2. Each partition matches either less, greater than, or equal to the pivot.
Worst and average case would be the same as regular QS, but since the different number of credit cards (let’s call this number **k**), can logically be much less than the number of items (k << n potentially), the number of partitions we need can be greatly reduced because of the “equal to” one. If the number of different cards is O(k) this quicksort will be O(kn).

<u>Q9</u>

For any value c >= 0, t increases by the number of tries it takes for random(1, 2^c) to equal 1. This has a probability of 1/2^c, and remember c increases by 1 every time this is successful, meaning the probability will lower.
Using the expected value formula:

$E(t_c\ (increase\ in\ t\ given\ specific\ c)) = 2^c$

$E(t) = \Sigma_{i=1}^{n} 2^i = 2^{n+1} - 2$

Q10

a, b)

Consider the perfect binary tree with the values 1-7 has explained in the question:

Value	Algo a) comparisons	Algo b) comparisons
1	5	6
2	3	4
3	5	5
4	1	2
5	5	5
6	3	3
7	5	4

Taking the average over all the values gives the expected value of the algo:

$E(a) = \frac{27}{7}$ $E(b) = \frac{29}{7}$

c)

Search costs:

Algo a:

Left: 2

Right: 2

Self: 1

Algo b:

Left: 2

Right: 1

Self: 2

Since every search for a node has to end with itself, this automatically adds n comparisons to algo b.

We know b saves 1 comparison whenever you need to traverse right in the tree.

Lets calculate the number of times the tree traverses right to search for an element. We know that since this is a perfect binary tree, exactly half of the searches will traverse right.

Total searches: $\Sigma_{i=1}^{log(n)} i2^i = 2 + 2^{log(n)+1}(log(n) - 1) = 2 + 2n(log(n) - 1)$

Dividing that by 2, we get the total amount of right searches:

$n(logn - 1) + 1 = nlogn - n + 1$

Thus the increase of comparisons from algo a to b is:

$n - (nlogn - n + 1) = 2n - nlogn - 1$

A graph can show us that starting at a height of 4, algo b has less comparisons than algo a because that is the point where $2n - nlogn - 1 < 0$

Q11

a)

Sequence	Probability
Prepend, prepend	p^2
Prepend, access	$p(1-p)$
Access, prepend	$p(1-p)$
Access, access	$(1-p)^2$

b) Let X be a RV such that X=1 if the operation is prepend, and X=0 if the operation is access.

$P(X = 1) = p$

$P(X = 0) = 1 - p$

$E(X_i) = p$

$E(\Sigma_{i=1}^{k} X_i) = \Sigma_{i=1}^{k} E(X_i) = kp$

c) Let $Z_i = Uniform(1,\ i)$, Y_i be a RV such that $Y_i = 1$ if the operation is prepend, and $Y_i = Z_i$ if the operation is access.

$E(Y_k) = p + (1-p)E(Z_k) = p + (1-p)\frac{kp}{2}$ (because kp is the expected number of elements at operation k)

d) Using information from all the past parts:

$E(\Sigma_{i=1}^{n} Y_i) = \Sigma_{i=1}^{k} E(Y_i) = \Sigma_{i=1}^{k}(p + (1-p)\frac{ip}{2}) = np + (1-p)\frac{p}{2}(\Sigma_{i=1}^{k} i)$

$= np + \frac{1}{4}np(1-p)(n+1) = np(1 + \frac{1}{4}(1-p)(n+1))$

Q12

a)

```
Smallest(k, S):
    if S.size == 1:
        return S[0]
    Initialize Sets: Less, Equal, Greater
    int pivot = random(0, S.size)
    for element in S:
        if element < S[pivot]:
            Less.append(element)
        elif element == S[pivot]:
            Equal.append(element)
        else:
            Greater.append(element)

    if k <= Less.size:   // k less than the pivot
        return Smallest(k, Less)
    elif k <= (Less.size + Equal.size):      // k is the pivot
        return S[pivot]
    else:  // k is greater than the pivot
        return Smallest(k-(Less.size + Equal.size), Greater)
```

b) Like quicksort, the worse case complexity is $O(n^2)$ in the case that you always pick a pivot that is either the greatest or smallest element, causing very uneven partitions. In the case for this algo, the worst case will happen if you are looking for the largest element and always pick the lowest pivot and vice versa.

c) Like quicksort, the average case is assuming a more reasonable pivot to be chosen each time, let's say ¼-¾ split and assume we always search through the ¾ partition. In this case the number of elements we search through over the iterations is represented by the series:

   $\frac{3n}{4} + \frac{9n}{16} + ... = \Sigma_{i=0}^{n} \frac{3^i n}{4^i} = 4n - \frac{3}{4}^n(3n) < 4n - 3n = n$

   Thus O(n) average case.

Q13

a) Using quicksort with 3 partitions (<, =, >) and random pivot selection.

   If all partitions are less than n/2 size then there is no majority element. Else choose the partition with more than n/2 elements (there can only be 1 for obvious reasons), and check if all the elements are the same. If they are, then that is the majority element, if not then recurse the algorithm on this partition.

b) Similar to question 12, unless we choose the worst partition each time, we will likely get a reasonable split with random pivots (lets use ¼-¾ again)

   The # of comparisons is: $n + 2(\frac{3n}{4}) + 2(\frac{9n}{16}) + 2(\frac{27n}{64}) = \frac{143n}{32}$

   The series ends when the partition can no longer be > n/2.

   Thus again the average case is O(n).