Problem 1

Exercises for Units 18 and 19

1) Each Union operation takes $O(1)$ time $\Rightarrow$ $O(\ell)$ time for ℓ union operations.

Running time of m Find operations is $O(m + \#\text{times some node gets a new parent})$

Because we perform Find operations after Union operations each node can get a new parent only once, this gives us the total running time $O(\ell+m+n)$

If we look more carefully: initially all nodes are a root nodes and with each Union operation one root node becomes non-root node. So, after ℓ union operations, we still have $n-\ell$ root nodes and root nodes do not get a parent during Find operations. $\Rightarrow$ $\#$ times some node gets a new parent $\leq \ell$ and we have the total running time $O(m+\ell+\ell) = O(m+\ell)$

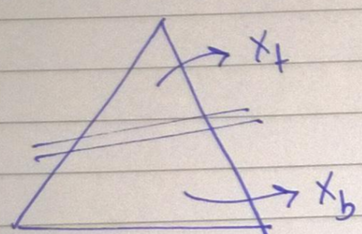
Problem 2

→ Exercise for unit 18 & 19

(2) To prove: The cost of a sequence of C compressions is $O((m + (k-1)n)\lceil \log_k n \rceil)$.

Proof: In this problem, instead of dividing the forest in equal parts, we make one part factor of k smaller.

i.e



$$|X_t| = \left\lceil \frac{n}{k} \right\rceil = n_t$$

$$|X_b| = n - \left\lceil \frac{n}{k} \right\rceil = n_b$$

Let C_t and C_b be the compression sequences for the top and bottom part.

→ $|C_t| = m_t$, $|C_b| = m_b$

We'll prove the bound by induction. For simplicity, consider the induction step first.

We know: $\text{cost}(C) \leq \text{cost}(C_b) + \text{cost}(C_t) + |X_b| + |C_t|$

By induction hypothesis:

$$\text{cost}(C_b) = (m_b + (k-1)n_b) \cdot \lceil \log_k n_b \rceil \quad \leq \lceil \log_k n \rceil$$

$$\text{cost}(C_t) = (m_t + (k-1) \cdot n_t) \cdot \lceil \log_k n_t \rceil \quad \leq \lceil \log_k n \rceil - 1$$

Adding the above two expressions

$$\text{cost}(C) \leq \left[(m_t + m_b) + (k-1)(n_t + n_b)\right] \cdot \lceil \log_k n \rceil$$

$$- \cancel{m_t} - (k-1)n_t + n_b + \cancel{m_t}$$

$$\leq (m + (k-1)n) \lceil \log_k n \rceil + n_b - (k-1)n_t$$

Observe that $n_b \leq (k-1)n_t$ and the claim follows.

We still need to prove the base case i.e when $\underline{n \leq k}$

→ By using the claimed bound,

$$\text{cost}(C) \leq (m + k^2) \cdot \lceil \log_k n \rceil^{\nearrow = 1} \leq m + k^2$$

We will argue that this is trivially true.

Using similar arguments as in exercise 1, we can argue that for any arbitrary sequence of union (·) and find (·), the cost of these operations is bounded as $O(n + m)$

∴ For base case, the bound holds trivially.

Problem 3

Exercises for Units 18 & 19

(3.) F forest, x node in F; $r(x)$ = height of subtree rooted at x

F is a rank forest iff $\forall$ node x, $\forall\, i$ s.t. $0 \le i < r(x)$ there is a child y_i of x with $r(y_i) = i$.

Dissection of a forest F with node set X:
partition of X into X_t and X_b s.t. $x \in X_t \Rightarrow$ every ancestor of x is also in X_t.

a) $(X_{\le s}, X_{>s})$ is a dissection

→ this is obviously a partition, we just have to check whether one of the sets is also upwards closed

$X_t := X_{>s}$, $X_b := X_{\le s}$

all nodes whose rank is $> s$ $\Rightarrow$ if $x \in X_{>s}$, then the subtree rooted at x has height $> s$

- but any any ancestor of x can only have a higher subtree

$\Rightarrow X_{>s}$ is upwards closed

b) $F(X_{\le s})$ is a rank forest with max. rank $\le s$

→ this is obvious – max. rank is $\le s$ by definition and it is a rank forest because F was a rank forest

c) $F(X_{>s})$ is a rank forest with max. rank $\le r-s-1$

only look at nodes whose rank in the original forest was $\ge s+1$

- nodes with rank $s+1$ in the original forest are now leaves
- all nodes in here lost children of rank $\le s$ in the original forest
- nodes who had max. rank $= r$ before now have children of rank

$0, 1, 2, \dots, r-s-2$ $\Rightarrow$ max rank is now $r-s-1$

rank in orig. forest: $s+1$, $s+2$, $s+3$, $s+(r-s-1) = r-1$

d) $|X_{>s}| \leq |X|/2^{s+1}$

- every node $x \in X_{>s}$ has at least one child of rank $0, 1, \dots, s$ and all these nodes and all their children are in $X_{\leq s}$

$x \in X_{>s} \Longrightarrow \geq 1 + 2 + 2^2 + \dots + 2^s = 2^{s+1} - 1$ nodes in $X_{\leq s}$

$x \in X_{>s}$ can have more than just one child of each rank

$$|X| = |X_{>s}| + |X_{\leq s}| \geq |X_{>s}| + (2^{s+1} - 1) \cdot |X_{>s}| = 2^{s+1} \cdot |X_{>s}|$$

$$\Rightarrow |X_{>s}| \leq \frac{|X|}{2^{s+1}}$$

e) # roots in $F(X_{\leq s}) \geq (s+1) \cdot |X_{>s}|$

- every root in $F(X_{\leq s})$ was either a root also in the original forest or was a child of some $x \in X_{>s}$

$x \in X_{>s} \longrightarrow \geq \underbrace{s+1}$ roots in $F(X_{\leq s})$

children of rank $0, 1, \dots, s$

can have more than 1 child of each rank

$\Rightarrow$ # roots in $F(X_{\leq s}) \geq (s+1) \cdot |X_{>s}|$

Problem 4

Point a)

$\texttt{extract} = [4, 3, 2, 6, 8, 1]$

Point b)

Let us first give a brief explanation of the algorithm. The main idea is that, instead of finding the number that is extracted each time there is such an operation, the algorithm finds, for each number, the operation that extracts it, if the number is ever extracted. The `extract_min` operation gets the smallest value that is already available. One other way to look at this is that the smallest number is extracted by the next `extract_min` operation, and, in general, each number is extracted by the next `extract_min` operation that does not extract a smaller number.

With this in mind, we will prove that the algorithm is correct by contradiction. We assume that there is some number that is not extracted correctly. Let $x = \texttt{extract}[j]$ be such a number, with minimal j, and let y be the number that should be in $\texttt{extract}[j]$.

Now, there are two possible cases: either $x < y$ or $x > y$. Let us start with $x < y$. We claim that, if x is in K_j when it is inserted into $\texttt{extract}[j]$, then, when the algorithm started, it must be that $x \in K_\ell$, for some $\ell \leq j$. This must be true, since the only operation that changes the sets K_j is on line 7, and it simply moves elements from a set K_j to a set K_ℓ, for some $\ell > j$. Therefore, if $x \in K_\ell$, $\ell > j$ originally, then it wouldn't be possible for it to be in K_j, at any point in the algorithm.

We conclude that, j-th `extract_min` could have extracted x instead of y, since the operation may extract any element that is already available (that is, from any K_ℓ, $\ell \leq j$), that hasn't been extracted up to that point. Since `extract` is correct until position $j-1$, by assumption, x was not extracted by any previous operation, and therefore it cannot happen that $x < y$.

Now, we prove that $x > y$ cannot happen as well, and therefore, reach a contradiction. It is clear from the algorithm that if a set is in some K_j when the algorithm starts, then it is always in some K_ℓ, that is, it is never removed from all the sets. If y was the correct value for $\texttt{extract}[j]$, then it must be that $y \in K_\ell$, for some $\ell \neq j$. Since `extract` is correct up to $j-1$, then it must be the case that $\ell > j$. Furthermore, since y can be extracted (and should) by the j-th operation, the it must have started at some set K_m, for some $m \leq j$. Therefore, since $\ell > j$, by some sequence of operations on line 7, y was moved from K_m to K_ℓ. This can only happen if K_j was removed, or y would be moved into K_j instead. But K_j cannot be removed, since K_j is removed only when $i = x$, which happens after $i = y$. We conclude that this case cannot happen, and, since there is no other possible case, we reach a contradiction on our assumption.

Point c)

We create n sets, one for each number, and then, using at most $n - m$ `union` operations, get m sets, one for each K_j. We also store, for each set (in the representative element), the index j, the representative elements of the next set (initially K_{j+1}) and the previous set (initially K_{j-1}). When computing the union of sets K_j, K_ℓ, and if we store them as K_ℓ (for $j < \ell$, as is the case in the algorithm), we simply keep the index and next element of K_ℓ, set the previous element to the previous of K_j, and set the next element of the previous element of K_j to be K_ℓ. To find the value of ℓ, on line 6, we use the index of the next representative element, which can be obtained in constant time.

Now, during the course of the algorithm, we run n `make_set` operations, at most n `union` operations (at most $n - m$ to obtain sets for each K_j, and then m more, one for each number that is extracted), and n `find_set` operations. All these operations take $O(n\alpha(n))$ time.