Solutions for Homework Assignment #4

Answer to Question 1. Let $G = (V, E)$ be the given graph where $V = \{1, 2, \ldots, n\}$ and $e_1, e_2, \ldots, e_m$ is the list of all edges in E, in the order in which they are to be removed. We are looking for the first edge e_i in the sequence such that, if we start with G and then remove the edges in the order $e_1, e_2, e_3, \ldots$, then after the removal of e_{i-1} and before removing e_i the graph still has at least one connected component with more than $\lfloor n/4 \rfloor$ nodes, but after removing e_i every connected component of the graph has at most $\lfloor n/4 \rfloor$ nodes.

Our algorithm reverses this process, as follows. We start with the graph $G' = (V, \emptyset)$ (i.e., G' has all the n nodes of G, but no edges). We then *add* to G' edges in the *reverse* order, i.e., first we add e_m, and then e_{m-1}, and then e_{m-2}, etc. While doing so, we keep track of the size (i.e., number of nodes) of the largest connected component of the graph so far. It is clear that the *first* edge whose addition in this process causes the graph G' to have at least one component of size greater than $\lfloor n/4 \rfloor$, is also the first edge whose removal in the original process breaks the graph G into connected components of size $\lfloor n/4 \rfloor$ or less.

To implement the above algorithm, we use the forest of rooted trees disjoint sets data structure, with the weighted unions and path compression rules. To keep track of the size of the connected components of the graph, we augment the data structure with a procedure FIND-SIZE(u) that gives the size of the set represented by element u: more precisely, FIND-SIZE(u) takes a representative u of a set (implemented as a pointer to the root of the corresponding tree), and returns the size of the set. To implement this procedure, we store the size of each tree in an additional variable associated with the root of each tree. This variable can be accessed and its value returned by FIND-SIZE(u) in constant time.

Assume that UNION is implemented to take two pointers to the roots of two disjoint trees, and returns a pointer to the root of the newly created tree. When we perform a UNION(u, v) operation, we set the size variable for the newly created tree to the sum of the size variables of the two trees pointed to by u and v. This only incurs constant overhead over the usual time complexity of UNION.

We begin with n disjoint sets, each containing a node of G. We then consider the edges, one at a time, in the order $e_m, e_{m-1}, e_{m-2}, \ldots$. When considering edge e_k we determine whether the two vertices of the edge belong to different sets (using the FIND operation). If so, we merge the two sets (using the UNION operation), determine the size of the merged set, and update the variable that keeps track of the size of the largest set so far. If the size of the largest set is now greater than $\lfloor n/4 \rfloor$, the addition of e_k created a connected component with more than $\lfloor n/4 \rfloor$ nodes, and so e_k is the edge that we are looking for.

The pseudocode of the algorithm is given below.

```
1  for i := 1 to n do MakeSet(i) end for
2  MaxSize = 1
3  let L = e_m, e_{m-1}, ..., e_1        L is the given list of edges in reverse order
4  (u, v) := first edge of L
5  while true do
6      u' := Find(u)
7      v' := Find(v)
8      if u' ≠ v' then
9          u := Union(u', v')
10         u_size := Find-Size(u)
11         MaxSize := max{MaxSize, u_size}
12         if MaxSize > ⌊n/4⌋ then return (u, v) end if
13     end if
14     (u, v) := next edge of L
15 end while
```

Since we use the forest of rooted trees disjoint sets data structure, with the weighted unions and path compression rules, the time complexity of this algorithm is $O(m \log^* n)$, where $m = |E|$ and $n = |V|$. To

see this, first note that it takes $O(n)$ time to initialize the n singleton sets in line 1, and an additional $O(m)$ time to form the reverse list of edges L in line 3. Then the algorithm executes a sequence of $n-1$ UNIONs and at most $2m$ FINDs (i.e., at most 2 FINDs for each edge in L). Note that since the graph G is connected $m \geq n-1$. So executing this sequence of UNIONs and FINDs takes at most $O(m \log^* n)$ time. Thus, the total time for the algorithm is $O(n) + O(m) + O(m \log^* n) = O(m \log^* n)$.

Answer to Question 2.

(1) We first show that for some $n \geq 1$, we have: $A(n) > c + \frac{17}{12}d$.

To do so consider the case where $n = 9$. Doing a sequence of $n = 9$ increments starting from number 0 costs: $(c+d)+(c+d)+(c+2d)+(c+d)+(c+d)+(c+2d)+(c+d)+(c+d)+(c+3d)$, so $T(9) = 9c+13d$, and $A(9) = T(9)/9 = c + \frac{13}{9}d > c + \frac{17}{12}d$.

(2) We next show that, for all $n \geq 1$, we have: $A(n) \leq c + \frac{3}{2}d$.

We prove this in two different ways.

a. Using the Accounting Method:

Note that each increment does the following:

i. It first resets to 0 every trit of a (possibly empty) initial sequence of 2's, and

ii. then it changes one trit from 0 to 1 or from 1 to 2.

Each reset in (i) costs d, and the trit change in (ii) costs $c+d$, so that the total actual cost of an increment that changes m trits is $c + md$.

To amortized the cost of each increment, we charge each increment only for doing (ii) (i.e., changing one trit from 0 to 1, or from 1 to 2). The cost of doing the resets in (i) will be covered by previously accumulated credits in the 2's.

Specifically, we charge each increment operation a total of $c + \frac{3}{2}d$ as follows:

- $c + d$ is used to cover the cost of changing one trit from 0 to 1 or from 1 to 2, plus
- $\frac{d}{2}$ is used as an extra credit attached to that trit.

Note that before any trit becomes a 2, it will receive $2 \times \frac{d}{2}$ credits: $\frac{d}{2}$ when it changes from 0 to 1, and an additional $\frac{d}{2}$ when it changes from 1 to 2. So every trit 2 in the display has accumulated a credit of d (this is the charge invariant). Each increment operation can then use these credits to cover the cost of resetting 2's to 0's, i.e., to cover the cost of doing (i) above.

So the charge of $c + \frac{3}{2}d$ for each increment fully covers the cost of executing this increment, i.e., of doing *both* (i) and (ii) above. Thus $A(n) \leq c + \frac{3}{2}d$.

b. Using the Aggregate Method:

Consider a sequence of n increments, starting with number 0.

The total cost of these n increments is $T(n) = n \times c +$ (total number of trits changes) $\times\, d$.

We now evaluate the total number of trits changes caused by n increments.

In the following, bit 0 is the least significant (i.e. the rightmost) trit:

- Trit 0 changes every $3^0 = 1$ increment. So it changes n times.
- Trit 1 changes every $3^1 = 3$ increments. So it changes $\lfloor \frac{n}{3} \rfloor$ times.
- Trit 2 changes every $3^2 = 9$ increments. So it changes $\lfloor \frac{n}{3^2} \rfloor$ times.
- ...
- Trit i changes every 3^i increments. So it changes $\lfloor \frac{n}{3^i} \rfloor$ times.

- ...

So the total number of trit changes due to the n increments is:

$$\begin{aligned} &\leq \Sigma_{i=0}^{\infty} \lfloor \frac{n}{3^i} \rfloor \\ &\leq \Sigma_{i=0}^{\infty} \frac{n}{3^i} \\ &\leq n \Sigma_{i=0}^{\infty} \frac{1}{3^i} \\ &\leq n \times \frac{1}{1-\frac{1}{3}} \;=\; \frac{3}{2} n \end{aligned}$$

Thus, $T(n) \leq n \times c + \frac{3}{2} n \times d$.

So the amortized cost of an increment is $A(n) = \frac{T(n)}{n} \leq c + \frac{3}{2} d$.

Answer to Question 3.

Data structure: Store members of S in an unsorted linked list; also keep track of the size in integer n.

Operations:

- INSERT(S, x): Append x at the end of the list; increment n.
- DIMINISH(S):
  i. $m \leftarrow$ MED(S)
  ii. loop over all elements in S: keep all those $< m$, skip all those $\geq m$
  iii. add as many copies of m as required to have exactly $\lfloor n/2 \rfloor$ elements remaining
  iv. $n \leftarrow \lfloor n/2 \rfloor$

Analysis: First, we describe the cost and charges for each operation.

- INSERT:
  - cost = \$0 (no comparison between elements is performed);
  - charge = \$12 (we'll see why below).
- DIMINISH:
  - cost = \$6n (\$5n for comparisons during call to MED, \$$n$ to compare every element to m);
  - charge = \$0 (we'll see why below).

Next, prove the following credit invariant by induction on the number of operations performed, starting from an initially empty multiset.

> The credit associated with multiset S of size n is always at least \$12n.

Base Case: When $S = \varnothing$ and $n = 0$, the total credit is \$0.

Ind. Hyp.: Suppose that the total credit for multiset S of size n is \$12n.

Ind. Step: If INSERT is called, it costs \$0 and adds \$12 to the total credit. The resulting size is $n + 1$ and the resulting credit is $\$12n + \$12 = \$12(n + 1)$.

If DIMINISH is called, it costs \$6n and adds \$0 to the total credit. The resulting size is $\lfloor n/2 \rfloor$ and the resulting credit is $\$12n - \$6n = \$6n = \$12(n/2) \geq \$12\lfloor n/2 \rfloor$.

Hence, in every case, the credit invariant is maintained.

Finally, we use the credit invariant to conclude that, in every sequence of m operations starting from an empty multiset, the total cost is bounded above by the total charge, and the total charge is at most $\$12m$. Hence, the amortized cost per operation is at most \$12—a constant, as desired.

(Note: this proves only an upper bound on the amortized cost. That upper bound is constant, so it answers the question asked in the assignment. In practice, to show that \$12 is the best constant, we would have to also describe how to construct a sequence of m operations, for any m, so that the total cost of the sequence is close to $\$12m$. This is left as an "exercise to the reader.")