First Name: Zhicheng
Last Name: Yan
Student ID: 1002643142

First Name: Zhihong
Last Name: Wang
Student ID: 1002095207

First Name: Jialiang
Last Name: Yi
Student ID: 1002286929

We declare that this assignment is solely our own work, and is in accordance with the University of Toronto Code of Behavior on Academic Matters.

This submission has been prepared using LaTeX.

Writer: Zhicheng Yan

Reader: Zhihong Wang, Jialiang Yi

Question 1(20 marks)
Consider the following "Graph Dismantling Problem". Let $G = (V, E)$ be a connected undirected graph with $n \geq 4$ nodes $V = \{1, 2, ..., n\}$ and m edges. Let $e_1, e_2, ..., e_m$ be all the edges of G listed in some specific order. Suppose that we remove the edges from G one at a time, in this order. Initially the graph is connected, and at the end of this process the graph is disconnected. Therefore, there is an edge $e_i, 1 \leq i \leq m$, such that just before removing e_i the graph 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. Give an efficient algorithm that determines this edge e_i. Assume that G is given to the algorithm as a (plain) linked list of the edges appearing in the order $e_1, e_2, ..., e_m$.

The worst-case running time of your algorithm must be asymptotically better than $O(mn)$. More efficient solutions will get more marks.

Hint : See "An application of disjoint-set data structures" in pages 562-564 of CLRS (Chapter 21).

Solutions that are unclear, inefficient, or overly complex may not get any points.

Solution:
The algorithm can be developed base on the disjoint sets implemented via forest with path compression. First, after remove all the edges from G, we can initialize n single element sets, and the element of each set is each vertex in the graph G. Also, each set is assigned an attribute which is $number\_of\_elements$. Initially, $number\_of\_elements$ of all the sets are 1. Next, we add the edges back in the reverse order(i.e. in the order of $e_m, e_{m-1}, ..., e_1$) one at a time. When add back each edges, two of the vertices will be connected. And we union the two corresponding sets(with path compression) that contain those two vertices. And the size of the set after doing the union will be the sizes of original two sets adding together. After the union, we check whether the size of that set is over $\lfloor \frac{n}{4} \rfloor$. If the

size of that set is over $\lfloor \frac{n}{4} \rfloor$. The edge that most recently added will be returned. If the size of that set is not greater than $\lfloor \frac{n}{4} \rfloor$, we continue adding the vertex until at least one set has a size of over $\lfloor \frac{n}{4} \rfloor$.

The worst-case running time is asymptotically better than $O(mn)$. This algorithm have two main steps. The first step to make single element sets, which totally takes $O(n)$ times. It takes $O(1)$ to make each set, and totally n sets, so it is $O(n)$. The second step is union two corresponding sets and check whether the $number\_of\_elements$ of that set is over $\lfloor \frac{n}{4} \rfloor$ after the union. To union, $FINDSET$ operation is performed to find the representative of the set and union two sets together. Thus, when doing each union with the path compression and checking the $number\_of\_elements$ to determine the most recent added edge that to be returned, it takes $O(\alpha(n))$ in worst case. Also, the total number of times that doing all the unions and checks are bounded by m times. So, the total complexity of the second step is not worse than $O(m\alpha(n))$. Then, the total complexity of the two main steps is asymptotically better than $O(n + m\alpha(n))$. Since $m \geq n - 1$, so there exists a positive constant c_0, such that $n \leq c_0 m$. Also, according to CLRS ($Page$572) and lecture notes, $\alpha(n) \leq 4$, we can get that $m\alpha(n) \leq 4m$. Consequently, $n + m\alpha(n) \leq c_0 m + 4m \leq cm$, where c is a positive constant such that $c = c_0 + 4$. Then, $O(n + m\alpha(n))$ is not worse than $O(m)$. Therefore the overall complexity of the algorithm is asymptotically better than $O(mn)$.

Writer: Jialiang Yi

Reader: Zhicheng Yan, Zhihong Wang

Question 2(20 marks)
A manufacturing company producing Product X maintains a display showing how many Product X's they have manufactured so far. Each time a new Product X is manufactured, this display is updated. Interestingly, this company prefers base 3 notation; instead of using bits (0 and 1) as in base 2 notation, they use trits (0, 1 and 2) to represent this number.
The total cost of updating the display is $\$(c + dm)$, where c is a fixed cost to open up the display, d is the cost of changing each display digit, and m is the number of digits that must be changed. For example, when the display is changed from 12201222 to 12202000 (because 12201222 + 1 = 12202000 in base 3) the actual cost to the company is $\$(c + dm)$ because 4 digits must be changed. The manufacturing company wants to amortize the cost of maintaining the display over all of the Product X's that are manufactured, charging the same amount to each. In other words, we are interested in the amortized cost of incrementing the display by one.
Let $A(n)$ be the amortized cost per display change operation when we perform a sequence of n successive display changes, starting from the number 0. For example, $A(3) = c + \frac{4}{3}d$.
In this question, we want you to give an upper bound on $A(n)$. More precisely, consider the list:

$$c + \tfrac{4}{3}d, c + \tfrac{17}{12}d, c + \tfrac{3}{2}d, c + \tfrac{7}{4}d, c + 2d, c + \tfrac{5}{2}d$$

(note that the numbers in this list are sorted from smallest to largest). What is the smallest cost B in the above list such that $A(n) \leq B$ for all $n \geq 1$?

Prove the correctness of your answer in two ways: first analyse $A(n)$ using the aggregate method, and then analyse $A(n)$ using the accounting method with an appropriate charging scheme.

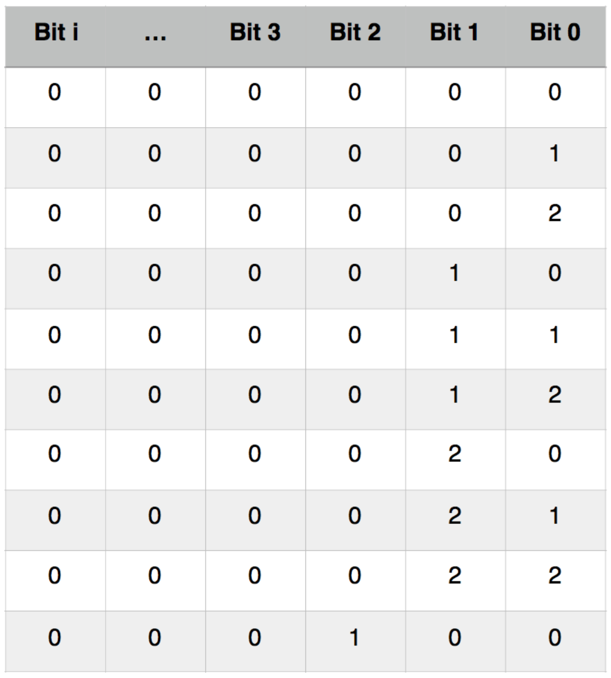
Make sure you justify why your answer is the smallest such B from the above list.

Solution:

$c + \frac{3}{2}$d is the smallest cost, for all $n \geq 1$

Aggregate Method:
Examine the question, the actual cost to the company is \$$(c + dm)$, wher c is fixed cost to open up the display, d is the cost of changing each display digit, and m is the number of digits that must be changed. Let $A(n)$ be the amortized cost per display change operation, by the definition of c, d and m, the actual cost and amortized cost is strongly related to m.
To explore the number of digits that must be changed per operation, a base 3 incremented chart is created to represent the company's display.

Base 3 notation increment chart



Bit i	...	Bit 3	Bit 2	Bit 1	Bit 0
0	0	0	0	0	0
0	0	0	0	0	1
0	0	0	0	0	2
0	0	0	0	1	0
0	0	0	0	1	1
0	0	0	0	1	2
0	0	0	0	2	0
0	0	0	0	2	1
0	0	0	0	2	2
0	0	0	1	0	0

Let n represent the number of time of incrementation(changing the display); let $T(n)$ be the worst-case(i.e max) cost of doing a sequence of n operations; let k represent the number of digits, and the sequence:

Bit 0 flips every single increment; total increment: n times;
Bit 1 flips every three increments; total increment: $\lfloor \frac{n}{3} \rfloor$ times;
Bit 2 flips every nine increments; total increment: $\lfloor \frac{n}{9} \rfloor$ times;
Bit i flips every 3^i increments; total increment: $\lfloor \frac{n}{3^i} \rfloor$ times.

Therefore, T(n) = $\sum_{i=0}^{\lfloor k-1 \rfloor} \lfloor \frac{n}{3^i} \rfloor$
$= \sum_{i=0}^{\lfloor log_3 n \rfloor} \lfloor \frac{n}{3^i} \rfloor$
$\leq n \sum_{i=0}^{\infty} (\frac{1}{3})^i$
$= n \frac{1}{1-\frac{1}{3}}$
$= \frac{3}{2} n$

By definition of amortized complexity analysis, $\frac{T(n)}{n}$ is the average cost of an operation in worst-case, in another word, the amortized cost of an operation. Thus

$A(n) = \frac{T(n)}{n} \leq$B$= c + \frac{3}{2}d$ for all $n \geq 1$

Justification:

Assume $\frac{3}{2}$ is not the smallest, one of the elements in the list $c + \frac{4}{3}d, c + \frac{17}{12}d, c + \frac{3}{2}d, c + \frac{7}{4}d, c + 2d, c + \frac{5}{2}d$ has the smaller amortized constant. As the numbers in the list are sorted from smallest to largest, start with $c + \frac{17}{12}d$, the number slightly smaller than $c + \frac{3}{2}d$

For $n = 9$, Actual Cost = T(9) = $\sum_{i=0}^{\lfloor log_3(9) \rfloor} \frac{9}{3^i} = 9 + 3 + 1 = 13$

$A(9) = \frac{T(9)}{9} = \frac{13}{9}$

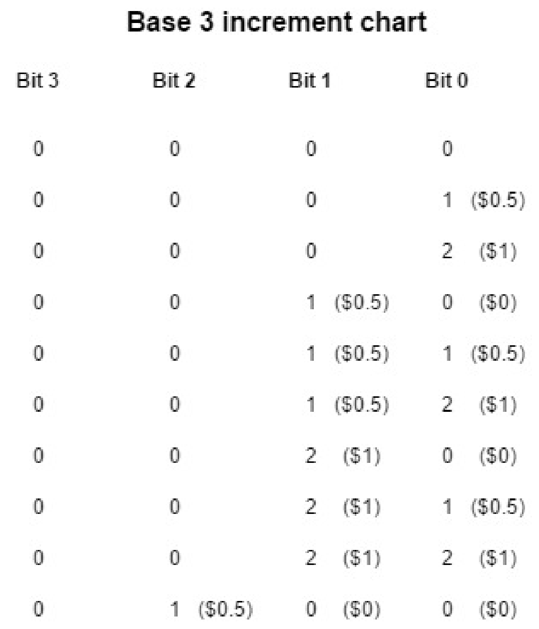
$\frac{17}{12} < \frac{13}{9} < \frac{3}{2}$, thus $\frac{17}{12}$ is not abundant enough to complete 9 incrementation. Consequently, $\frac{3}{2}$ is the smallest such B from the above list.

Accounting method:

Invariant:
$I(n)$: If n increments happened, then every digit of 0 have 0 token of credit; every digit of 1 have 0.5 token of credit; every digit of 2 have 1 token of credit.

c and d are fixed cost except m, which is the number of digits that must be changed. By definition of accounting method, we overcharge each operation by a small amount to save up for later use.

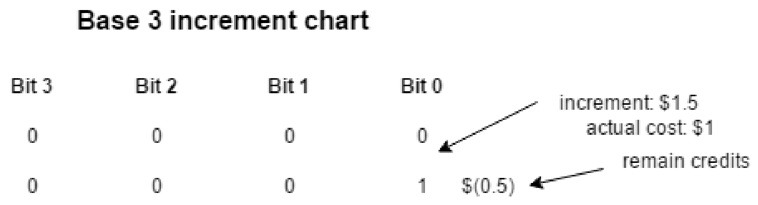
Assume the actual cost for every incrementation on the base 3 display is 1 token, but we overcharge each operation such that it costs 1.5 tokens ($\frac{3}{2}$) for every incrementation. To demonstrate each operation, a graph is created and is shown below.



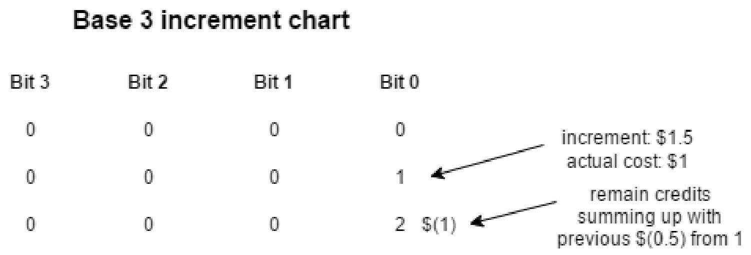
Base 3 increment chart

Bit 3	Bit 2	Bit 1	Bit 0
0	0	0	0
0	0	0	1 ($0.5)
0	0	0	2 ($1)
0	0	1 ($0.5)	0 ($0)
0	0	1 ($0.5)	1 ($0.5)
0	0	1 ($0.5)	2 ($1)
0	0	2 ($1)	0 ($0)
0	0	2 ($1)	1 ($0.5)
0	0	2 ($1)	2 ($1)
0	1 ($0.5)	0 ($0)	0 ($0)

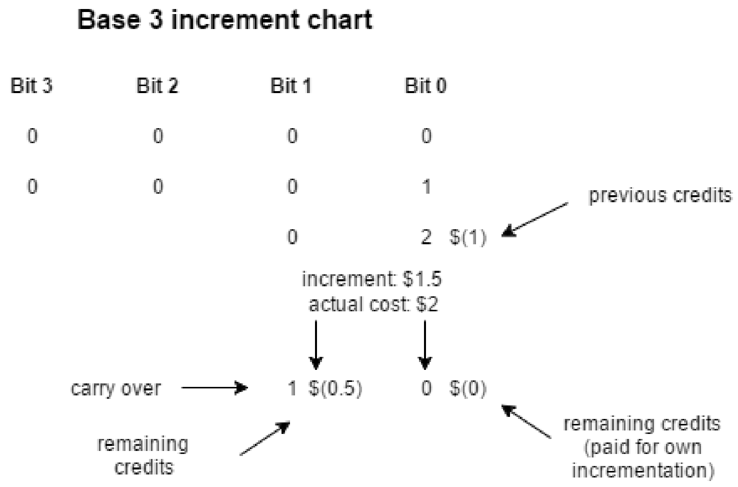
By assumption, the actual cost for every incrementation is 1 token, the extra charges will be covering the carry-over, which happens every three incrementation. Returning to the increment chart, starting at Bit 0, the digit increments from 0 to 1 and costs 1.5 tokens, but the actual cost is 1 token. Consequently, when the digit flips to 1, it remains 0.5 token as the bracket showed.



Next, the digit is incrementing from 1 to 0 and costs another 1.5 tokens. Extracting the actual cost, 1 token, another 0.5 token is remaining. Summing up with the previous 0.5 tokens, when the digit flips to 2, it remains 1 token as the bracket showed.



Now, the digit reaches 2, when next incrementation comes, it has to carry over to Bit 1 and reset itself back to 0. Two flips happened in this process, which costs 2 tokens in total, but only charging for 1.5 token. The digit at Bit 0 has 1 token credit remaining and will be used up to reset itself back to 0. The incremented bit at Bit 1 will be charged for 1 tokens. Since the digit at Bit 0 paid for itself using its remaining credit, this gives the digit at Bit 1 a remaining credit of 0.5 as the bracket indicated.



In conclusion, each incrementation actually costs 1 token. However, for each carry-over bit, which happens every three incrementation, one extra token is needed. Thus, charges each incrementation 1.5 tokens and save up the remaining credits for future use. This gives us the result of, for every digit that is 0, remaining credit is \$0; for every digit that is 1, remaining credit is \$0.5; for every digit that is 2, remaining credit is \$1 and when next incrementation comes, use previous remaining credit to set itself back to 0 and save the remaining 0.5 token from the overall 1.5 token charge to the carry-over bit. Result is identical to the Base 3 increment chart shown above.

Justification:

Assume $\frac{3}{2}$ is not the smallest, and there exist a constant $c < \frac{3}{2}$ such that $A(n) \leq B = c$ for all $n \geq 1$. That means, assume the actual cost for every incrementation on the base 3 display is 1 token, but we overcharge each operation such that it costs c tokens ($c < \frac{3}{2}$) for every incrementation.

By assumption, the actual cost for every incrementation is 1 token, the extra charges will be covering the carry-over, which happens every three incrementation. Starting from Bit 0, the digit is initially 0, remaining credit is 0 token. First, incremented to 1, remaining credit is $c - 1$, where $c - 1 < 0.5$. Next, the digit incremented to 2, remaining credit is $(c-1) + (c-1)$ which is $2c - 2$, where $2c - 2 < 1$. When the next incrementation happened, current digit has to carry over to sibling Bit and reset itself back to 0, which costs 2 tokens in total. Now, we are only charging c tokens, current bit doesn't have enough credit to reset it self back to 0 ($2c - 2 < 1$), so it has to use $-c + 2$ tokens ($c - (2c - 2)$) from the charge. The carry-over bit remains credit $c - (-c + 2) - 1$, which is $2c - 3$ tokens. However $2c - 3 < 2(\frac{3}{2}) - 3 = 0$. Therefore, contradiction; c is not abundant enough to complete n incrementation.

$A(n) = \frac{T(n)}{n} \leq$B$=\frac{3}{2}$ for all $n \geq 1$; $\frac{3}{2}$ is the smallest B.

Therefore, in the list, $c + \frac{4}{3}d, c + \frac{17}{12}d, c + \frac{3}{2}d, c + \frac{7}{4}d, c + 2d, c + \frac{5}{2}d$, the smallest is $c + \frac{3}{2}d$.

Writer: Zhihong Wang

Reader: Jialiang Yi, Zhicheng Yan

Question 3(10 marks)
A $multi - set$ is like a set where repeated elements matter. For example, $\{1, 8, 3\}$ and $\{3, 3, 8, 1, 8\}$ represent the same set but different $multi - sets$. Consider the abstract data type that consists of a multi-set of integers S and supports the following operations:

- $Insert(S, x)$: insert integer x into multi-set S;

- $Diminish(S)$: delete the $\lceil |S|/2 \rceil$ largest integers in S.

(For example, if $S = \{3, 3, 8, 1, 8\}$, then after $Diminish(S)$ is performed, $S = \{3, 1\}$.)

A simple data structure for this ADT, and the corresponding algorithms for each operation, are as follows:
The elements of S are stored in an unsorted linked list, and the size of S is stored in a variable n.

- $Insert(S, x)$: Append x at the end of the list; increment n.

- $Diminish(S)$:

i. $m \leftarrow Med(S)$ find the median of the elements in S

ii. loop over all elements in S: keep all those $< m$, delete all those $\geq m$

iii. add as many copies of m as required to have exactly $\lfloor n/2 \rfloor$ elements remaining

iv. $n \leftarrow \lfloor n/2 \rfloor$

The above uses an algorithm Med that returns the "median" of any list of integers. To simplify this question, we use a slightly non-standard definition of "median": Med returns the $\lceil n/2 \rceil$-th largest integer in the list (including duplicates); for example $Med([3, 2, 1, 1, 3, 3, 3])$ returns 3. Also for this question, assume that *Med* performs at most $5n$ pairwise comparisons between integers during its execution on any list of length n.

Prove that when we execute an arbitrary sequence of *Insert* and *Diminish* operations starting from an empty set S, the amortized cost of an operation, in terms of the number of pairwise comparisons between the elements of the set, is constant. To prove this, use the accounting method to analyse the amortized complexity of the *Insert* and *Diminish* algorithms: (a) state clearly what charge you assign to each operation, (b) state and prove an explicit credit invariant, and (c) use your credit invariant to justify that the amortized cost of an operation is constant.

Solution:

By lectures, tutorials and textbooks, assume all code lines except recursions or loops only require constant time.

Prove the amortized cost of an operation (number of pairwise comparisons between the elements of the set) is constant.

Note: Let the length of any list to be n. Assume we can get the tail from any list, so there is no comparisons in *Insert* and the cost of *Insert* is constant. *Med* performs at most $5n$ pairwise comparisons. Step *ii* of *Diminish* performs n pairwise comparisons because it will loop over all elements (n elements) in a list. Step *iii* of *Diminish* will call *Insert* to add node copies. These is no comparison in step *iii* and *iv* of *Diminish*. i.e. Only step *i* and *ii* of *Diminish* have pairwise comparisons that we need to consider in this question, and the max number of comparisons is $6n$.

(a). State what charge we assign to each operation:

12 credits for *Insert*,

0 credits for *Diminish*.

(b). State and prove an explicit credit invariant:

Define invariant:

$I(k)$: If k operations executed,
then every node still have at least 12 credits.

Prove $\forall k \in \mathbb{N}, I(k)$

prove by simple induction:

Base Case:

Before any operation executed, $k = 0$, then there is no node, so this case is trivially true.

then $I(0)$

Inductive Step: Let $k \in \mathbb{N}$

Inductive Hypothesis: Assume $I(k)$

Want To Prove: $I(k+1)$

Assume there are $k+1$ operations and the number of nodes is n before $(k+1)th$ operation executed. Each node hold at least 12 credits after k times executions (By I.H) and at least $12n$ credits totally. In $(k+1)th$ execution, there are 3 cases:

Case 1: $(k+1)th$ operation is $Insert$.

In this case, there is no comparison and the new node also holds 12 credits.

Therefore, $I(k+1)$ holds.

Case 2: $(k+1)th$ operation is $Diminish$ and n is even number.

In this case, the max number of comparisons will be $6n$ (By Note). Then the remaining number of nodes is $n/2$ (By *Diminish iii*) and there are at least $6n$ credits remaining too ($12n - 6n = 6n$). So $6n$ credits for $n/2$ nodes, each node will still holds at least 12 credits.

Therefore, $I(k+1)$ holds.

Case 3: $(k+1)th$ operation is *Diminish* and n is odd number.

In this case, the max number of comparisons will be $6n$ (By Note). Then the remaining number of nodes is $\lfloor n/2 \rfloor$ (By *Diminish iii*), and there are at least $6n$ credits remaining too ($12n - 6n = 6n$). So $6n$ credits for $\lfloor n/2 \rfloor$ nodes, each node will still holds at least 12 credits.

Therefore, $I(k+1)$ holds.

Therefore, by all cases, $I(k+1)$ holds.

(c). Use our credit invariant to justify that the amortized cost of an operation is constant:

By (a) and (b), if there are k operations, the worst cost will be $12k$. So on average, the amortized cost is smaller or equal to $12k/k = 12$, which is constant.

i.e. if we earn 12 credits on *Insert* operation, it is enough to pay all costs of operations. In other words, amortized cost per operations is smaller or equal to 12, which is constant.