Question 1(Zhenjie Xiong):

//Verified by Xiaoxue Xing

The principle of my algorithm is going to be disjoint-set data structure. The question states that after removing a particular edge e_i every connected component will have $\leq \lfloor \frac{n}{4} \rfloor$ nodes. Instead of looking for that particular edge by removing edges, I add edges from e_m one by one until e_i when the first connected component with $> \lfloor \frac{n}{4} \rfloor$ nodes shows up.

So I'm looking for the threshold, here's how I would do it. We will create a set for every single node from vertices V = {1, 2, 3, … , n}. It takes constant time for creating one set. Thus, for n nodes, we create n sets which takes O(n) time.

Then we do union in an opposite order of the edges. For each union we do union by rank and path compression. Next to each edge, there are 2 vertices V_1 and V_2 whose node numbers are $|V_1|$ and $|V_2|$ respectively. That threshold edge e_i will have $|V_1|$, $|V_2|$ each $< \lfloor \frac{n}{4} \rfloor$, but the sum of the two will be barely $> \lfloor \frac{n}{4} \rfloor$. In the worst case, we have to union all the sets until we have 4 connected components, each of which has $\lfloor \frac{n}{4} \rfloor$ nodes. Comparing to a original connected graph G, we remove 4 edges as a result. Doing this and union by rank will take O($\alpha(n)$ m). $\alpha(n)$ is a slow growing load factor. Therefore the worst case running time is O(m - 4) which is O(m).

Now, n nodes implies at least n - 1 edges, that is m $\geq n-1$. So n $\leq$ m + 1. We combine our previous result should be getting O(m + n) for our algorithm in the worst case. Because n $\leq$ m + 1, we can reduce the algorithm's worst case to O(m+m+1) which is O(m).

Question 2(Xiaoxue Xing):

//Verified by Zhenjie Xiong

<u>Aggregation method</u>**:**

A(n) = c + 3d/2

It is a base three notation. So we have the following procedure.

```
Increment(B)
i = 0
while i<B.length and A[i] == 2:
        A[i] = o
        i = i +1
If i < B.length
        A[i]= 1
```

When each increment is called, B[o] changes. (0 to 1, or 1 to 2, or 2 to 3.) The next bit up, B[1], flips only once in every three times: a sequence of n increment operations on an initially zero counter causes B[1] to flip $\lfloor n/3 \rfloor$ times. Similarly, bit B[2] flips only every ninth time, or $\lfloor n/9 \rfloor$ times in a sequence of n increment operations. In general, for i = 0,1,2,...,k-1, bit B[i] flips $\lfloor n/3^i \rfloor$ in a sequence of n increment operations on an initially zero counter. For $i \geq k$, bit B[i] does not exist, and so it cannot flip. The total number of flips in the sequence is thus $\sum_{i=0}^{k-1} \lfloor n/3^i \rfloor < n \sum_{i=0}^{\infty} \lfloor 1/3^i \rfloor = 3n/2$. Since c is a fixed cost to open up the display, d is the cost of changing each display digit. The worst-case time for a sequence of n INCREMENT operations on an initially zero counter is therefore nc + 3nd/2. So, the amortized cost per operation is (nc + 3nd/)/n = c + 3d/2

Accounting method:

For the amortized analysis, firstly charge 3/2 dollars to set the bit to be 1 or 2. When a bit is set, we use 1 dollar (out of the 3/2 dollars charged) to pay for the actual setting of the bit, and we credit 1/2 dollar on the bit to use it later when we flip the bit back to 0. If the bit is 2, it must first set from 0 to 1, then from 1 to 2, so the credit is 1 dollar when the bit is 2. So if the bit is 2, there must 1 dollar credit on that bit. At any point in time, every bit is 2 in the counter has a dollar of credit on it, and thus we can charge nothing to reset a bit to 0. we just pay for the reset with the dollar bill on the bit.

We have the assumption that every bit equals to 1 has ½ credit and every bit equal to 2 has 1 credit. This can easily be proved by induction: initially, the counter is 0 and there is no credit, so the invariant is trivially true, credit is 1/2. When the counter is 1, it goes up to 2 and add ½ more credit and the credit goes up to 1.

Then, assuming that the invariant is true at a particular point, let's perform one INCREMENT operation: the cost of flipping bits from 2 to 0 is payed for by the credit

stored with each 2, the cost of flipping a single bit from 0 to 1 or 1 to 2 is payed for with 1 from the 3/2 charged to the operation, and we store the remaining 1/2 together with the bit that was just changed to 1 or 2.

None of the other bits are changed. Since if the bit want to be 2, it must first change from 0 to 1 and then 1 to 2. So, the bit with 1 has ½ dollar credit and bit for 2 has 1 dollar credit.Hence, the credit invariant is still true (every bit equal to 2 has a 1 credit).

This shows that the total charge for the sequence of operations is an upper bound on the total cost of the sequence, and since in this case the total charge is 2n, we get that the amortized cost per operation is no more than 2n/n = 2 (same as before).

Now we can determine the amortized cost of INCREMENT. The cost of resetting the bits within the while loop is paid for by the dollars on the bits that are reset. The INCREMENT procedure sets at most one bit to be 1 or 2, and therefore the amortized cost of an INCREMENT operation is at most 3/2 dollars. The number of 1 and 2 in the counter never becomes negative, and thus the amount of credit stays nonnegative at all times. Thus, for n INCREMENT operations, the total amortized cost is nc + 3nd/2, ans A(n) = c + 3d/2.

Question 3(Xiaoxue Xing):

//Verified by Zhenjie Xiong

Firstly, when we insert an number to the set, we charge 13. We use 1 for the cost of insert. The rest of 12 is the credit. After n insertion, it has 12n credit left.

Each time, we run the diminish. Firstly, it need to find the med, which cost at most 5n pairwise comparisons. Then, loop over the S and compare the element with the med. This cost at most n pairwise comparisons. Next, we add as many copies of m as required to have exactly ⌊n/2⌋ elements remaining and change n to n/2. This does

not need pairwise comparisons, so no cost needed. Therefore, each diminish takes about 6n.

After each diminish, the n change to ⌊n/2⌋. If we continue use the diminish, this time, we need 6*⌊n/2⌋cost. After we use diminish until n equals to 0. We need cost 6n + 6*⌊n/2⌋ + 6* ⌊⌊n/2⌋/2⌋ + …….$\leq 6 * 2n$. So the balance is always greater than 0.

The amortized time is 12n/n which is 12, so it is O(1). Therefore it is constant.