This is an oral presentation assignment. Again, there are three regular problems (#1-#3) and two programming problem (#4a and #4b). You should work in groups of three. The sign-up sheet will be online soon (details on Piazza) and your group should sign up for a slot. Each person in the group must be able to present every regular problem. The TA/Professor will select who presents which regular problem. You are not required to hand anything in at your presentation, but you may if you choose.

The programming problems will be submitted to autolab, similar to HW#1. In order to make sure that you start on this assignment in a timely fashion, it is broken into two parts. The first part is worth only 5 points and is due at 11:59pm on Sunday September 16th. The second part is worth the remaining 20 points and is due at 11:59pm on Friday September 21st. You will not have to present anything orally for the programming problem. Please include a comment in your program explaining the algorithm you used. You can discuss the problem with your group-mates, but must write the program by yourself. Please do not copy.

1. (Balls in Bins) There are $n$ balls and an infinite number of bins. A bin can have 0 or more balls in it. A move consists taking all the balls of some bin and putting them into distinct bins. The cost of a move is the number of balls moved.

   a) Define the potential of a state of this system as the sum of the potentials of all the bins. The potential of a bin with $k$ balls in it is:

   $$\Phi(k) = \max(0, k - z)$$

   Where for convenience $z = \lfloor\sqrt{n}\rfloor$.

   Prove that the amortized cost of a move is at most $2z$.

   b) Suppose $n = k(k+1)/2$ for some integer $k$. Give an initial arrangement of the $n$ balls in the bins, and an infinite sequence of moves, such that every move costs $k$.

   c) Letter of Recommendation Problem (i.e. advanced optional problem): Prove a tighter upper bound than the one in part (a), that matches the lower bound in part (b), assuming that $n = k(k+1)/2$ for some integer $k$. 
2. **Potentials are good for Lower Bounds too**

Consider the problem of finding both the maximum element and the minimum element of an arbitrary set of \( n \) distinct numbers, where \( n \) is even.

It turns out that \( \frac{3}{2}n - 2 \) comparisons are required in the worst case to do this. That is, \( \frac{3}{2}n - 2 \) is a lower bound on the number of comparisons needed. There’s also an algorithm that achieves this bound. That is \( \frac{3}{2}n - 2 \) is an upper bound on the number of comparisons needed. Since these bounds are equal, they are optimal.

(a) Prove the following theorem:

**Theorem:** Consider a deterministic comparison based-algorithm \( A \) which does the following: Given a set \( S \) of \( n \) numbers as input, \( A \) returns the largest and the smallest element of \( S \). Prove that there is an input on which \( A \) must perform at least \( \frac{3}{2}n - 2 \) comparisons.

**Hints:** Call an element *top* if it has been involved in at least one comparison and it has never lost a comparison. Call an element *bottom* if it has been involved in at least one comparison and never won a comparison. Call an element *free* if it has been involved in zero comparisons. Call an element *middle* if it has won at least one comparison and lost at least one comparison.

Every element falls into exactly one of these categories, and this classification of elements evolves over time. Initially all elements are free. You know that at the end of a run of any correct algorithm, there are _______free ones, _______middle ones, _______top ones and _______bottom ones.

Define a potential function \( \Phi() \) that is a linear function in the number of each type of element. You, the adversary, have to decide the outcome of each comparison to give to algorithm \( A \) when it asks you to compare two elements. By making the “right” choice of the outcome of a comparison, you should ensure that the potential decreases by at most 1 for each comparison. Using this fact, along with the initial value and final value of the potential, prove that algorithm \( A \) must do at least \( \frac{3}{2}n - 2 \) comparisons.

(b) The proof above leads to a optimal deterministic algorithm (in terms of the number of comparisons). Describe one such algorithm.
To Form a More Perfect Hash

In this problem, we design another scheme for creating a perfect hash function. We have a set \( S \) of \( n \) elements from a universe \( U \), and we let \( M = [0, \ldots, m - 1] \). The perfect hash function \( PH() \) must have the following properties:

- \( PH : U \rightarrow M \).
- If \( x \) and \( y \) are distinct elements of \( S \) then \( PH(x) \neq PH(y) \).
- The size of the range \( M \) is \( O(n) \).
- Preprocessing the set \( S \) to create \( PH() \) must be \( O(n) \) time.
- The space required to store \( PH() \) must be \( O(n) \) bits.
- Computing \( PH(x) \) can be done “efficiently”.

Let \( m \) be even. Let \( h_0 \) and \( h_1 \) be hash functions, where \( h_0 : U \rightarrow [0, \ldots, m/2 - 1] \) and \( h_1 : U \rightarrow [m/2, \ldots, m - 1] \). Define a bipartite graph \( G = (E, V) \) where \( V = \{0, \ldots, m - 1\} \) and there is an edge \( e(x) \) for each element \( x \) in \( S \): \( e(x) = \{h_0(x), h_1(x)\} \).

One can show (but it’s beyond the scope of this problem) that if \( h_0 \) and \( h_1 \) are chosen from a universal family, and \( m = 2.5n \) then \( G \) is acyclic with probability at least \( 3/5 \).

The setup for this perfect hash function is as follows. So we’re given the set \( S \) of size \( n \). We’re going to preprocess \( S \) to construct the perfect hash function as follows: Pick \( m \) to be even and about \( 2.5n \). Randomly pick \( h_0() \) and \( h_1() \) from the universal family. Construct \( G \) and test if it’s acyclic. Keep trying different \( h_0() \) and \( h_1() \) until \( G \) is acyclic.

There will be additional information computed when the perfect hash function is being created. The information is stored at each node of \( G \). Let the information stored at node \( k \) be denoted \( I[k] \).

Now the perfect hash function we’ll design, \( PH(x) \), works as follows. Given \( x \in U \) compute the pair \( (k, \ell) = (h_0(x), h_1(x)) \). Making use of \( k, \ell, I[k], \) and \( I[\ell] \) the algorithm will choose which of \( k \) or \( \ell \) is to be the hash value of \( x, PH(x) \).

It remains for you to fill in these details: (1) What is the information \( I[k] \) that is stored at node \( k \) of \( G \)? (2) How do you compute it in time \( O(n) \) for all nodes? And (3) how is it used to choose between \( k \) or \( \ell \)?

Describe a scheme for (1), (2) and (3) where \( I[k] \) is only one bit (for each \( k \). [Hint: you might first try to allow \( I[k] \) up to \( O(\log m) \) bits. Then try to get it down to two bits. And finally get it down to one bit. Partial credit is available.]
(5 pts) 4a. **Splay Test**

In this part you will implement the bottom-up splaying algorithm described in lecture. For purposes of submission, your program will be called `splaytest.*`. It’s due Sunday September 16th at 11:59pm.

The input consists of two lines. The first line has two integers \( n \) and \( m \) with \( 1 \leq n, m \leq 10^6 \). The second line contains a sequence of \( m \) space-separated integers, \( k_i \) with \( 0 \leq k_i < n \).

Your program will build a splay tree with keys \( 0, 1, 2, \ldots n-1 \) that is a long left path with \( n-1 \) at the root. For each \( k_i \) your program will output the depth of \( k_i \) in the current tree, and then splay key \( k_i \). For our purposes the depth of the root is 0. The time limit is 10 seconds.

First sample input:

```
5 6
2 1 0 4 4 3
```

output:

```
2 1 1 4 0 2
```

Second sample input:

```
1000000 10
0 1 2 3 4 5 6 7 8 9
```

output:

```
999999 500000 250001 125000 62503 31251 15627 7813 3909 1954
```

(20 pts) 4b. **Block Sorting Revisited**

This part is due September 21st at 11:59pm. For purposes of submitting your code the program will be called `blocksort2`.

This problem is almost identical to the programming problem on homework 1. The only difference is in the bounds. This time you must complete the sorting in at most \( 2n \) block swaps. Also, \( n \) can be up to a million, and the time bound is 20 seconds.