This is an oral presentation assignment. Again, there are three regular problems (#1-#3) and one programming problem (#4). 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 1-hour 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 problem is due **Monday Feb 10th**, 11:59pm, and should be submitted to autolab, similar to HW#1. You will not have to present anything orally for the programming problem. You can discuss the problem with your group-mates, but must write the program by yourself. Please do not copy.

1. **(25 pts) Dequeue**

   A dequeue is data structure that represents an ordered list of items. It allows access to both ends of the list. (The ends are called the *front* and *back* of the list.) Specifically it supports the following operations:

   - d isempty(): return true if the dequeue is empty
   - d push(x): add x to the front of the dequeue
   - d pop(): remove and return the element at the front of the dequeue
   - d inject(x): add x to the back of the dequeue
   - d eject(): remove and return the element at the back of the dequeue

   A stack is a similar data structure that just allows the s isempty() s push() and s pop() operations.

   Describe how to implement a dequeue using three stacks. (Your data structure can maintain additional bookkeeping information, such as the number of elements in the dequeue. But it should not make use of an array or other data structures to store the data in the dequeue.)

   Your algorithm should be constant amortized time per operation. To be concrete, measure the cost of an operation in terms of the number of stack pushes and pops that it does. Find small constants $c_1$ and $c_2$ such that the amortized costs of d push() and d inject() are at most $c_1$ and the amortized costs of d pop() and d eject() are at most $c_2$. Prove your result using a potential function.
2. (25 pts) **Two Problems on Splaying**

**Deep Splaying**

Here’s a way to modify splay trees to avoid restructuring the tree on every access. Keep track of \( n \), the number of nodes in the tree. When a node is accessed we compute its depth (distance from the root). If its depth is greater than \( 1 + 4 \log_2 n \), then we do the usual splay operation. Otherwise we leave the tree alone.

(a) Prove that the amortized cost per access using this algorithm is \( O(\log n) \).

(b) The number of rotations done by this algorithm is bounded by a function of the initial tree, which is not a function of the access sequence. So no matter how long the sequence is, the number of rotations done cannot exceed this bound. Find a function \( f(T) \) of the initial tree \( T \), and then prove that the number of rotations done on any access sequence is at most \( f(T) \). (We’re not asking that you find the smallest such function, just find any function that works.) It will be useful to make use of the Strong Access Lemma from the lecture notes.

**Splaying Near The Front**

Consider a splay tree of \( n \) nodes. The nodes are called \( 1, 2, \ldots, n \), from left to right in the tree (inorder traversal). Choose weights on the nodes and apply the access lemma to prove that:

(c) The amortized cost of splaying node \( i \) is \( O(1 + \log i) \) for all \( i \leq n \).

Hint: \[
\sum_{i=1}^{\infty} \frac{1}{i^2} = \frac{\pi^2}{6}
\]
3. (25 pts) **An Interesting Hash Family.**

We say that $H$ is $\ell$-universal over range $m$ (or $\ell$-wise independent) if for every fixed sequence of $\ell$ distinct keys $\langle x_1, x_2, \ldots, x_\ell \rangle$, if we choose a hash function $h$ at random from $H$, the sequence $\langle h(x_1), h(x_2), \ldots, h(x_\ell) \rangle$ is equally likely to be any of the $m^\ell$ sequences of length $\ell$ with elements drawn from $\{0, 1, \ldots, m - 1\}$. It’s easy to see that if $H$ is 2-universal then it is universal. (Check for yourself!)

Consider a universe $U$ of strings $s = s_1, s_2, \ldots, s_n$ of length $n$ from an alphabet of size $k$. (Each character is an integer in $\{0, 1, \ldots, k - 1\}$.) Hence $|U| = k^n$. Assume that $m = 2^b$.

An interesting universal family $\mathcal{G}$ (of functions from $U$ to $\{0, \ldots, m - 1\}$) can be obtained as follows. First, generate a 2-dimensional table $T$ of $b$-bit random numbers; recall that $b = \lg(m)$. The first index of $T_{i,j}$ is in the range $[1, n]$ and the second index is in the range $[0, k - 1]$. Now define the hash function $g_T()$ as follows:

$$g_T(s) = \bigoplus_{i=1}^{n} T_{i,s_i}$$

where “$\bigoplus$” represents the bitwise-xor function (recall, each $T_{i,j}$ is a $b$-bit string). The output of $g_T(s)$ is a $b$-bit string which is then interpreted as a number in $\{0, \ldots, m - 1\}$.

Note that since each choice of the table $T$ gives a hash function $g_T$, and $T$ is specified by $n \cdot k \cdot b$ bits, the family $\mathcal{G}$ consists of $2^{nkb}$ functions.

(a) (10 pts) Prove that $\mathcal{G}$ is not 4-universal.

**Hint:** To show that $\mathcal{G}$ is not 4-universal, you should exhibit 4 distinct keys $\langle x_1, x_2, x_3, x_4 \rangle$ such that if you were told the values of $g_T(x_1), g_T(x_2)$, and $g_T(x_3)$, you could infer the value of $g_T(x_4)$ uniquely (without knowing anything else about $T$). This will mean that not all 4-tuples of hash-values are equally likely, since the first 3 entries in the tuple $\langle g_T(x_1), g_T(x_2), g_T(x_3), g_T(x_4) \rangle$ determined the 4th entry. You can do this using $n = 2$ and $k = 2$.

(b) (15 pts) Prove that $\mathcal{G}$ is 3-universal. (You can get 7 of these points by proving the weaker statement that it is 2-universal.)
4. (25 pts) **Pancake Sorting**

Given a permutation of $0, 1, \ldots, n - 1$ it’s possible to sort it using a sequence of *prefix reversals*. This operation replaces a prefix of the permutation by its reversal. Your task is to write a program that computes a sequence of prefix lengths to reverse in order to sort a given permutation. The running time bound for your program is 10 seconds. Call your program *pancake.*

Here’s the grading scheme. If the sequence of prefix reversals doesn’t sort the permutation, the answer is wrong. Let $k$ be the number of prefix reversals in your solution. If $k > 4n$ your answer is considered wrong. If $k > 3n$ on any of your solutions, two points are deducted from your total score. If $k > 2n$ on any of your solutions one point is deducted from your total. So doing it in at most $2n$ is the gold standard for this problem.

**INPUT:** The first line contains $n$, with $2 \leq n \leq 5 \times 10^5$. The second line consists of the numbers $a_0, a_1, \ldots, a_{n-1}$ separated by blanks. These numbers will be a permutation of $0, 1, \ldots, n - 1$.

**OUTPUT:** The first line of output is $k$, the number of prefix reversals in your solution. This must be at most $2n$. The following $k$ lines each contain a number, which is the length of the prefix to reverse. These numbers should all be between 1 and $n$, inclusive.

Below are some input/output pairs on the left and right.

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>5 3 4 0 1</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 0 1</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>3 2 1 0</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>3 1 2</td>
<td>3</td>
</tr>
</tbody>
</table>