1. Consider a “superstack” data structure which supports four operations: create, push, pop, and superpop. The four operations are implemented using an underlying standard stack $S$ as shown below.

```python
def create():
    S = Stack.create()

def push(x):
    S.push(x)

def pop():
    return S.pop()

def superpop(k,A): // k is an integer, A is an array with size >= k
    i = 0
    while i < k
        A[i] = S.pop()
        i = i + 1
```

Show that each of these operations uses a constant amortized number of stack operations. In your solution you should:

- Define your potential function $\Phi$.
- State, for each operation, its actual time, the change in potential, and the amortized time.

2. Suppose we add a superpush operation to the superstack from the previous problem, defined as follows:

```python
def superpush(k,A): // k is an integer, A is an array with size >= k
    i = 0
    while i < k
        S.push(A[i])
        i = i + 1
```

Is it still true that each of the superstack operations uses a constant amortized number of stack operations? Answer YES or NO. If your answer is YES, give an amortized analysis as in the previous problem. (If you need to use a different potential function that is fine, just be sure to define it.) If your answer is NO, explain why.

3. The dynamic arrays described in class (and as implemented in Python) cannot efficiently remove an item $A[i]$ from the middle of an array $A$. Instead, performing this operation while preserving the order of the remaining items requires time proportional to the length of the array to move each element in a position greater than $i$ one position earlier in the array. But suppose that we do not care whether the remaining items stay in the same order, as long as the array contains the same remaining items in some order. In this case, describe how to remove $A[i]$ from an array, using only a constant number of length, get, set, increment, or decrement operations.

4. Suppose that we maintain a binary counter as described in class, but with a decrement operation as well as the increment operation that was described. The decrement operation should be the opposite of the increment operation, in that it subtracts one from the binary value stored in this counter.

(a) Describe how to perform the decrement operation in time proportional to the amount of information within the counter that gets changed by the operation.

(b) Does the data structure with both increment and decrement operations still take constant amortized time per operation? If so, provide a potential function for which the amortized time is constant, and explain why it is constant. If not, explain why there does not exist a suitable potential function.
5. Compute the amortized time for the Decrement operation on dynamic arrays for the case where the underlying array \( B \) is re-allocated. (This was left as an exercise at the end of the notes on amortized time.)

6. Consider the following problem:

We are given a collection of \( n \) rectangles with sides parallel to the \( x \)- and \( y \)-axes. (Such rectangles are sometimes called isothetic. We want to determine whether any two rectangles overlap.

To be precise, two rectangles overlap if they intersect and their intersection has positive area. If they touch at a point or a line but are otherwise disjoint, they do not overlap. For example, in the collection of rectangles shown below on the left, the answer is yes because the rectangles labeled D and F overlap, so for this example the answer is yes. Notice that E and G share a boundary, but they do not overlap.

![Diagram of rectangles]

Assume that each rectangle is represented as a 4-tuple \((x_L, x_R, y_B, y_T)\) where

- \( x_L \) is the \( x \)-coordinate of the left edge of the rectangle,
- \( x_R \) is the \( x \)-coordinate of the right edge of the rectangle,
- \( y_B \) is the \( y \)-coordinate of the bottom edge of the rectangle,
- \( y_T \) is the \( y \)-coordinate of the top edge of the rectangle.

The input for the collection of rectangles shown above on the left is given in the table shown above on the right.

<table>
<thead>
<tr>
<th></th>
<th>( x_L )</th>
<th>( x_R )</th>
<th>( y_B )</th>
<th>( y_T )</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>2</td>
<td>10</td>
<td>2</td>
<td>14</td>
</tr>
<tr>
<td>B</td>
<td>4</td>
<td>8</td>
<td>26</td>
<td>32</td>
</tr>
<tr>
<td>C</td>
<td>6</td>
<td>12</td>
<td>18</td>
<td>22</td>
</tr>
<tr>
<td>D</td>
<td>15</td>
<td>21</td>
<td>17</td>
<td>23</td>
</tr>
<tr>
<td>E</td>
<td>18</td>
<td>26</td>
<td>7</td>
<td>13</td>
</tr>
<tr>
<td>F</td>
<td>19</td>
<td>25</td>
<td>20</td>
<td>30</td>
</tr>
<tr>
<td>G</td>
<td>24</td>
<td>36</td>
<td>0</td>
<td>7</td>
</tr>
<tr>
<td>H</td>
<td>30</td>
<td>32</td>
<td>16</td>
<td>20</td>
</tr>
<tr>
<td>I</td>
<td>34</td>
<td>40</td>
<td>28</td>
<td>34</td>
</tr>
</tbody>
</table>

Suppose we solve this by sweeping a vertical line from left to right, as we did for the problem of detecting whether a collection of segments has an intersecting pair. Describe a data structure that could be used to solve this problem. That is, describe what type of data the data structure contains, and what operations it needs to support. Give high-level pseudocode that uses the data structure to solve the problem.

**Note:** The problem is only asking for the API: what objects need to be stored and retrieved, and the update and query operations that the data structure need to be provided to support to solve the problem specified above. It is NOT asking how to implement the data structure. Later in the quarter we will discuss an elegant implementation of this data structure.

**Hint:** Think about what the intersection of a vertical line and a collection of isothetic rectangles looks like, and how it changes when the vertical sweep line hits a vertical boundary of a rectangle.