Exercises #4
Amortized Analysis

Theoretical Background

Different methods for proving amortized bounds:

- **Aggregate** method: examine/bound total cost and calculate the average
- **Accounting** method: impose extra charge on inexpensive operations, saving for future expensive operations
- **Potential** method: define a potential function on the data structure state and use it to bound the cost

Stack as an Array

Consider the **stack** model we have shown in the class, with two operations:

- `push(`x`)` - puts the item x on the top of the stack
- x `= pop()` - deletes and returns the top item of the stack (or `NULL` if the stack is empty).

You start with an array of size 1. When you reach its limit you need to allocate a new array, copying all the items to it. In the lecture we have used a model where allocating a new array had cost 0 and putting or removing an item from the array would cost 1 (so any push and pop without resizing would cost 1, and a resize would cost n, with n items being copied to the new array).

We have shown in the previous lecture that we could prove an **amortized cost of 3** (per `push`) if we **double the size of the array** any time we need to make it grow.

1. Consider now that **allocating memory for n positions costs** n (instead of 0). This means allocating a new array of size $2n$ and copying n items to it would now cost $3n$. Does the amortized cost of 3 from the previous example still hold? What would be the new value for the amortized cost?

2. Let's go back to the original memory model, but consider now that besides growing the array, you also want to **shrink it**, to save space when you have many `pop()` operations.

   (a) Imagine you use the same growing strategy for growing (doubling the array) and that you use the reverse shrinking strategy: **as soon as the stack has less items than half of its space, you halve the size of the array**. What is the worst possible sequence of operations you can now imagine? What would the amortized cost be in that case?

   (b) Consider now that you only **shrink to half when only 1/4 of the stack is occupied**. What is now the worst case? What is the amortized cost for any sequence of operations?

3. Consider now a stack implemented on an "infinite array", with no need to grow and shrink, and with `push` and `pop` operations, each one costing 1.
   Imagine you want to implement the following new operation:

   - `multipop(k)` - pop the top k items from the stack
     (or all elements if the size of the stack is less than k).

   One way you could implement this would be by doing successive `pop()` operations. Show that the amortized cost for any sequence of `push`, `pop` or `multipop` operations is still linear, that is, the amortized cost of a single operation is constant.

4. What if we now need to add another operation `multipush`, that pushes k elements to the stack. Would the amortized cost per operation still be constant?

Binary Counter

5. Consider a **binary counter** with the operation **increment** having a cost equal to the number of flipped bits. We have shown in the lecture that the amortized cost for one increment was (at most) **2**.

   Now, consider a version of this counter where **flipping the k-th bit costs** 2^k (with $k = 0$ being the less significant bit). Explain why now in a sequence of n increments a single increment could have a linear cost, but show that the amortized cost per increment is logarithmic.

Dictionary Structure

6. One of the most common classes of data structures are the **"dictionary" data structures** that support fast **insert** and **lookup** operations into a set of items.

   Suppose you implement a dictionary using a sorted array. This would be good for lookups (using binary search we can take up to $\mathcal{O}(\log n)$ time) but bad for insertions (can take linear time). If you used a linked list, than inserts are efficient ($\mathcal{O}(1)$) but the lookups are not (they can take linear time).

   Here is a simple array-like data structure that can provide better than (amortized) linear options for both insertions and lookups. The main idea is to use a collection of arrays, where array i has size 2^i . Each array is either empty or full, and each is in sorted order. There is no relationship between items in different arrays. To know which arrays should be full or empty, consider the binary representation of the number of items and use the arrays in which the bit is non-zero. For instance, 11 items ($11 = 1 + 2 + 8$) would use the arrays of sizes 1, 2 and 8, with the other arrays being empty. This would result in something like:

```
A[0] = {6}
A[1] = {2,13}
A[2] = empty
A[3] = {1,6,7,8,10,15,17,29}
```

   (a) Suppose that in order to do a lookup we do a binary search in each filled array. How much time will this take in the worst case?

   (b) And what about inserts? Suppose we start by creating a new array of size 1 with the new item. We then check to see if $A[0]$ if full. If it is empty, we are done (just put the new array in position 0). If not, then we merge the (sorted) contents of the new array with $A[0]$ and we now verify if $A[1]$ is full, and so on, stopping when we find an array that is not full. Experiment be "manually" inserting some numbers to see what happens. What is now the worst case for an insertion? And what is the amortized cost of an insertion? [hint: use the result of question 5]