- See previous practice problem sets for instructions.
- I repeat one very important instruction: You must explicitly state all *non-trivial* assumptions that you make. When in doubt, over-communicate.
- Since the TAs are busy with their final exams, there will be no tutorial for this problem set. Please feel free to approach me if you have questions.

1. Recall the problem of implementing a k-bit binary counter $C[0 \ldots (k-1)]$ that we discussed in class. We had—naturally—assumed that flipping a bit costs 1 unit of time. In this question we look at a weighted version of this problem, where flipping bit $C[i]$ costs 2^i steps. We start with the counter being all zeroes, and increment it n times.
   (a) What is the worst-case cost of an increment operation? What is an upper bound on the worst-case cost of n increment operations?
   (b) Use aggregate analysis to show that the amortized cost of an increment operation is $\mathcal{O}(\log_2 n)$.

2. The array is a data structure which supports constant-time index-based insertion and retrieval. But this speed comes at a cost: each array has a fixed size that is decided when it is created, and the array cannot hold more than the pre-assigned maximum number of elements. In contrast, a linked list can grow to accommodate as many elements as needed but retrieval takes linear time in the worst-case (and also on average).

   It turns out that we can use arrays to implement a data structure that (i) can grow to hold as many elements as needed, and (ii) allows for constant-time index-based retrieval, and constant amortized cost of insertion.

   The idea is as follows. We use a variable $DynArray$ to hold the elements. We assume that array indexing starts with 0. At any point of time, $DynArray$ refers to an underlying array. Let $count$ denote the number of elements currently stored in $DynArray$, and let $size$ denote the maximum number of elements that can be stored in $DynArray$. Thus $DynArray[count-1]$ stores the last element in the data structure, and the data structure is full when $count = size$.

   We initialize $count$ and $size$ to 0, and $DynArray$ to NIL.

   The $Retrieve(i)$ function is implemented as you would expect: if $i \geq count$ then return NIL, indicating that the element $DynArray[i]$ doesn't exist. Otherwise, return $DynArray[i]$.

   The $Insert(x)$ function is implemented as follows:

- If $\mathtt{count} = 0$: Create an array A of size 1 and set $\mathtt{DynArray} = A$. Set $\mathtt{DynArray}[0] = x$, $\mathtt{count} = 1$, $size = 1$.
- Else:
  - If $\mathtt{count} = size$:
    - Create an array A of size $2 \times size$
    - Copy over the contents of $\mathtt{DynArray}$ to $A[0] \dots A[size - 1]$
    - Set $\mathtt{DynArray} = A$, $size = 2 \times size$
  - Set $\mathtt{DynArray}[\mathtt{count}] = x$, $\mathtt{count} = \mathtt{count} + 1$

That is: when we run of space in the current array we create a new array with twice the current size, copy over the existing array to the beginning of the new array, and carry on with the new array.

Note that $Retrieve()$ runs in constant worst-case time, since it consists of an indexed array lookup and a couple of constant-time checks. But what is the cost of $\mathtt{Insert}()$? Suppose we start with an empty data structure, and call $\mathtt{Insert}()$ n times. What would be the amortized cost of an $\mathtt{Insert}()$ operation in this sequence?

Assume that creating an array takes time linear in the size of the array, and accessing/writing an array element takes constant time. Ignore the other costs (such as the cost for comparing two numbers) in the following analysis, since it is the array operations that contribute most to the cost.

(a) What is the worst-case cost of a call to $\mathtt{Insert}()$? What is an upper bound on the worst-case cost of n calls to $\mathtt{Insert}()$?

(b) Use aggregate analysis to show that a call to $\mathtt{Insert}()$ has *constant* amortized cost.

(c) Use the *accounting* method to show that a call to $\mathtt{Insert}()$ has constant amortized cost.

(d) Use the *potential* method to show that a call to $\mathtt{Insert}()$ has constant amortized cost.

3. Let (G, s, t, c) be a flow network with integral capacities, let f be a feasible flow in this network, and let G_f be the corresponding residual network. Prove that there exists a feasible flow g in (G, s, t, c) with $val(g) > val(f)$ if and only if there exists a flow h in G_f such that $val(h) = (val(g) - val(f))$. Clearly state any properties of networks or flows that we proved in class, if you use them in your proof.