The goode workes that men don whil they ben in good lif al amortised by synne folwyng.

— Geoffrey Chaucer, "The Persones [Parson's] Tale" (c.1400)

I will gladly pay you Tuesday for a hamburger today.

— J. Wellington Wimpy, "Thimble Theatre" (1931)

I want my two dollars!

— Johnny Gasparini [Demian Slade], "Better Off Dead" (1985)

14 Amortized Analysis

14.1 Incrementing a Binary Counter

It is a straightforward exercise in induction, which often appears on Homework 0, to prove that any non-negative integer n can be represented as the sum of distinct powers of 2. Although some students correctly use induction on the number of bits—pulling off either the least significant bit or the most significant bit in the binary representation and letting the Recursion Fairy convert the remainder—the most commonly submitted proof uses induction on the value of the integer, as follows:

Proof: The base case $n = 0$ is trivial. For any $n > 0$, the inductive hypothesis implies that there is set of distinct powers of 2 whose sum is $n - 1$. If we add 2^0 to this set, we obtain a *multiset* of powers of two whose sum is n, which might contain two copies of 2^0. Then as long as there are two copies of any 2^i in the multiset, we remove them both and insert 2^{i+1} in their place. The sum of the elements of the multiset is unchanged by this replacement, because $2^{i+1} = 2^i + 2^i$. Each iteration decreases the size of the multiset by 1, so the replacement process must eventually terminate. When it does terminate, we have a *set* of distinct powers of 2 whose sum is n. □

This proof is describing an algorithm to increment a binary counter from $n - 1$ to n. Here's a more formal (and shorter!) description of the algorithm to add 1 to a binary counter. The input B is an (infinite) array of bits, where $B[i] = 1$ if and only if 2^i appears in the sum.

```
INCREMENT(B[0..∞]):
  i ← 0
  while B[i] = 1
      B[i] ← 0
      i ← i + 1
  B[i] ← 1
```

We've already argued that INCREMENT must terminate, but how quickly? Obviously, the running time depends on the array of bits passed as input. If the first k bits are all 1s, then INCREMENT takes $\Theta(k)$ time. The binary representation of any positive integer n is exactly $\lfloor \lg n \rfloor + 1$ bits long. Thus, if B represents an integer between 0 and n, INCREMENT takes $\Theta(\log n)$ time in the worst case.

14.2 Counting from 0 to n

Now suppose we want to use INCREMENT to count from 0 to n. If we only use the worst-case running time for each INCREMENT, we get an upper bound of $O(n \log n)$ on the total running time. Although this bound is correct, we can do better. There are several general methods for proving that the total running time is actually only $O(n)$. Many of these methods are logically equivalent, but different formulations are more natural for different problems.

14.2.1 The Summation (Aggregate) Method

The easiest way to get a tighter bound is to observe that we don't need to flip $\Theta(\log n)$ bits *every* time INCREMENT is called. The least significant bit $B[0]$ does flip every time, but $B[1]$ only flips every other time, $B[2]$ flips every 4th time, and in general, $B[i]$ flips every 2^ith time. If we start with an array full of 0's, a sequence of n INCREMENTs flips each bit $B[i]$ exactly $\lfloor n/2^i \rfloor$ times. Thus, the total number of bit-flips for the entire sequence is

$$\sum_{i=0}^{\lfloor \lg n \rfloor} \left\lfloor \frac{n}{2^i} \right\rfloor < \sum_{i=0}^{\infty} \frac{n}{2^i} = 2n.$$

Thus, *on average*, each call to INCREMENT flips only two bits, and so runs in constant time.

This 'on average' is quite different from the averaging we consider with randomized algorithms. There is no probability involved; we are averaging over a sequence of operations, not the possible running times of a single operation. This averaging idea is called *amortization*—the *amortized* time for each INCREMENT is $O(1)$. Amortization is a ~~sleazy~~ clever trick used by accountants to average large one-time costs over long periods of time; the most common example is calculating uniform payments for a loan, even though the borrower is paying interest on less and less capital over time. For this reason, 'amortized *cost*' is a common synonym for amortized running time.

Most textbooks call this technique for bounding amortized costs the *aggregate* method, or *aggregate analysis*, but this is just a fancy name for adding up the costs of the individual operations and dividing by the number of operations.

> **The Summation Method.** *Let $T(n)$ be the worst-case running time for a sequence of n operations. The amortized time for each operation is $T(n)/n$.*

14.2.2 The Taxation (Accounting) Method

A second method we can use to derive amortized bounds is called either the *accounting* method or the *taxation* method. Suppose it costs us a dollar to toggle a bit, so we can measure the running time of our algorithm in dollars. Time is money!

Instead of paying for each bit flip when it happens, the Increment Revenue Service charges a two-dollar *increment tax* whenever we want to set a bit from zero to one. One of those dollars is spent changing the bit from zero to one; the other is stored away as *credit* until we need to reset the same bit to zero. The key point here is that we always have enough credit saved up to pay for the next INCREMENT. The amortized cost of an INCREMENT is the total tax it incurs, which is exactly 2 dollars, since each INCREMENT changes just one bit from 0 to 1.

It is often useful to distribute the tax income to specific pieces of the data structure. For example, for each INCREMENT, we could store one of the two dollars on the single bit that is set for 0 to 1, so that *that* bit can pay to reset itself back to zero later on.

> **Taxation Method 1.** *Certain steps in the algorithm charge you taxes, so that the total cost incurred by the algorithm is never more than the total tax you pay. The amortized cost of an operation is the overall tax charged to you during that operation.*

A different way to schedule the taxes is for *every* bit to charge us a tax at *every* operation, regardless of whether the bit changes of not. Specifically, each bit $B[i]$ charges a tax of $1/2^i$ dollars for each INCREMENT. The total tax we are charged during each INCREMENT is $\sum_{i \ge 0} 2^{-i} = 2$ dollars. Every time a bit $B[i]$ actually needs to be flipped, it has collected exactly \$1, which is just enough for us to pay for the flip.

> **Taxation Method 2.** *Certain portions of the data structure charge you taxes at each operation, so that the total cost of maintaining the data structure is never more than the total taxes you pay. The amortized cost of an operation is the overall tax you pay during that operation.*

In both of the taxation methods, our task as algorithm analysts is to come up with an appropriate 'tax schedule'. Different 'schedules' can result in different amortized time bounds. The tightest bounds are obtained from tax schedules that *just barely* stay in the black.

14.2.3 The Charging Method

Another common method of amortized analysis involves *charging* the cost of some steps to some other, earlier steps. The method is similar to taxation, except that we focus on where each unit of tax is (or will be) spent, rather than where is it collected. By charging the cost of some operations to earlier operations, we are overestimating the total cost of any sequence of operations, since we pay for some charges from future operations that may never actually occur.

For example, in our binary counter, suppose we charge the cost of clearing a bit (changing its value from 1 to 0) to the previous operation that sets that bit (changing its value from 0 to 1). If we flip k bits during an INCREMENT, we charge $k-1$ of those bit-flips to earlier bit-flips. Conversely, the single operation that sets a bit receives at most one unit of charge from the next time that bit is cleared. So instead of paying for k bit-flips, we pay for at most two: one for actually setting a bit, plus at most one charge from the future for clearing that same bit. Thus, the total amortized cost of the INCREMENT is at most two bit-flips.

> **The Charging Method.** *Charge the cost of some steps of the algorithm to earlier steps, or to steps in some earlier operation. The amortized cost of the algorithm is its actual running time, minus its total charges* ***to*** *past operations, plus its total charge* ***from*** *future operations.*

14.2.4 The Potential Method

The most powerful method (and the hardest to use) builds on a physics metaphor of 'potential energy'. Instead of associating costs or taxes with particular operations or pieces of the data structure, we represent prepaid work as *potential* that can be spent on later operations. The potential is a function of the entire data structure.

Let D_i denote our data structure after i operations, and let Φ_i denote its potential. Let c_i denote the actual cost of the ith operation (which changes D_{i-1} into D_i). Then the *amortized* cost of the ith operation, denoted a_i, is defined to be the actual cost plus the increase in potential:

$$\boxed{a_i = c_i + \Phi_i - \Phi_{i-1}}$$

So the *total* amortized cost of n operations is the actual total cost plus the total increase in potential:

$$\sum_{i=1}^{n} a_i = \sum_{i=1}^{n} \left(c_i + \Phi_i - \Phi_{i-1}\right) = \sum_{i=1}^{n} c_i + \Phi_n - \Phi_0.$$

A potential function is *valid* if $\Phi_i - \Phi_0 \geq 0$ for all i. If the potential function is valid, then the total *actual* cost of any sequence of operations is always less than the total *amortized* cost:

$$\sum_{i=1}^{n} c_i = \sum_{i=1}^{n} a_i - \Phi_n \leq \sum_{i=1}^{n} a_i.$$

For our binary counter example, we can define the potential Φ_i after the ith INCREMENT to be the number of bits with value 1. Initially, all bits are equal to zero, so $\Phi_0 = 0$, and clearly $\Phi_i > 0$ for all $i > 0$, so this is a valid potential function. We can describe both the actual cost of an INCREMENT and the change in potential in terms of the number of bits set to 1 and reset to 0.

$$\begin{aligned} c_i &= \text{\#bits changed from 0 to 1} + \text{\#bits changed from 1 to 0} \\ \Phi_i - \Phi_{i-1} &= \text{\#bits changed from 0 to 1} - \text{\#bits changed from 1 to 0} \end{aligned}$$

Thus, the amortized cost of the ith INCREMENT is

$$a_i = c_i + \Phi_i - \Phi_{i-1} = 2 \times \text{\#bits changed from 0 to 1}$$

Since INCREMENT changes only *one* bit from 0 to 1, the amortized cost INCREMENT is 2.

> **The Potential Method.** *Define a potential function for the data structure that is initially equal to zero and is always nonnegative. The amortized cost of an operation is its actual cost plus the change in potential.*

For this particular example, the potential is precisely the total unspent taxes paid using the taxation method, so not too surprisingly, we obtain precisely the same amortized cost. In general, however, there may be no way of interpreting the change in potential as 'taxes'. Taxation and charging are useful when there is a convenient way to distribute costs to specific steps in the algorithm or components of the data structure; potential arguments allow us to argue more globally when a local distribution is difficult or impossible.

Different potential functions can lead to different amortized time bounds. The trick to using the potential method is to come up with the best possible potential function. A good potential function goes up a little during any cheap/fast operation, and goes down a lot during any expensive/slow operation. Unfortunately, there is no general technique for doing this other than playing around with the data structure and trying lots of different possibilities.

14.3 Incrementing and Decrementing

Now suppose we wanted a binary counter that we can both increment and decrement efficiently. A standard binary counter won't work, even in an amortized sense; if we alternate between 2^k and $2^k - 1$, *every* operation costs $\Theta(k)$ time.

A nice alternative is represent a number as a pair of bit strings (P, N), where for any bit position i, at most one of the bits $P[i]$ and $N[i]$ is equal to 1. The actual value of the counter is $P - N$. Here are algorithms to increment and decrement our double binary counter.

```
INCREMENT(P, N):
  i ← 0
  while P[i] = 1
      P[i] ← 0
      i ← i + 1
  if N[i] = 1
      N[i] ← 0
  else
      P[i] ← 1
```

```
DECREMENT(P, N):
  i ← 0
  while N[i] = 1
      N[i] ← 0
      i ← i + 1
  if P[i] = 1
      P[i] ← 0
  else
      N[i] ← 1
```

Here's an example of these algorithms in action. Notice that any number other than zero can be represented in multiple (in fact, infinitely many) ways.

$$\begin{array}{c} P=10001 \\ N=01100 \\ P-N=5 \end{array} \xrightarrow{++} \begin{array}{c} P=10010 \\ N=01100 \\ P-N=6 \end{array} \xrightarrow{++} \begin{array}{c} P=10011 \\ N=01100 \\ P-N=7 \end{array} \xrightarrow{++} \begin{array}{c} P=10000 \\ N=01000 \\ P-N=8 \end{array} \xrightarrow{--} \begin{array}{c} P=10000 \\ N=01001 \\ P-N=7 \end{array} \xrightarrow{--} \begin{array}{c} P=10000 \\ N=01010 \\ P-N=6 \end{array} \xrightarrow{++} \begin{array}{c} P=10001 \\ N=01010 \\ P-N=7 \end{array}$$

Incrementing and decrementing a double-binary counter.

Now suppose we start from $(0,0)$ and apply a sequence of n INCREMENTS and DECREMENTS. In the worst case, operation takes $\Theta(\log n)$ time, but what is the amortized cost? We can't use the aggregate method here, since we don't know what the sequence of operations looks like.

What about the taxation method? It's not hard to prove (by induction, of course) that after either $P[i]$ or $N[i]$ is set to 1, there must be at least 2^i operations, either INCREMENTS or DECREMENTS, before that bit is reset to 0. So if each bit $P[i]$ and $N[i]$ pays a tax of 2^{-i} at each operation, we will always have enough money to pay for the next operation. Thus, the amortized cost of each operation is at most $\sum_{i\geq 0} 2 \cdot 2^{-i} = 4$.

We can get even better bounds using the potential method. Define the potential Φ_i to be the number of 1-bits in both P and N after i operations. Just as before, we have

$$\begin{aligned} c_i &= \text{\#bits changed from 0 to 1} + \text{\#bits changed from 1 to 0} \\ \Phi_i - \Phi_{i-1} &= \text{\#bits changed from 0 to 1} - \text{\#bits changed from 1 to 0} \\ \Longrightarrow \qquad a_i &= 2 \times \text{\#bits changed from 0 to 1} \end{aligned}$$

Since each operation changes *at most* one bit to 1, the ith operation has amortized cost $a_i \leq 2$.

*14.4 Gray Codes

An attractive alternate solution to the increment/decrement problem was independently suggested by several students. *Gray codes* (named after Frank Gray, who discovered them in the 1950s) are methods for representing numbers as bit strings so that successive numbers differ by only one bit. For example, here is the four-bit *binary reflected* Gray code for the integers 0 through 15:

$$0000, 0001, 0011, 0010, 0110, 0111, 0101, 0100, 1100, 1101, 1111, 1110, 1010, 1011, 1001, 1000$$

The general rule for incrementing a binary reflected Gray code is to invert the bit that would be set from 0 to 1 by a normal binary counter. In terms of bit-flips, this is the perfect solution; each increment of decrement *by definition* changes only one bit. Unfortunately, the naïve algorithm to *find* the single bit to flip still requires $\Theta(\log n)$ time in the worst case. Thus, so the total cost of maintaining a Gray code, using the obvious algorithm, is the same as that of maintaining a normal binary counter.

Fortunately, this is only true of the naïve algorithm. The following algorithm, discovered by Gideon Ehrlich[1] in 1973, maintains a Gray code counter in constant *worst-case* time per increment! The algorithm uses a separate 'focus' array $F[0..n]$ in addition to a Gray-code bit array $G[0..n-1]$.

```
EhrlichGrayInit(n):
  for i ← 0 to n − 1
      G[i] ← 0
  for i ← 0 to n
      F[i] ← i
```

```
EhrlichGrayIncrement(n):
  j ← F[0]
  F[0] ← 0
  if j = n
      G[n − 1] ← 1 − G[n − 1]
  else
      G[j] = 1 − G[j]
      F[j] ← F[j + 1]
      F[j + 1] ← j + 1
```

[1]Gideon Ehrlich. Loopless algorithms for generating permutations, combinations, and other combinatorial configurations. *J. Assoc. Comput. Mach.* 20:500–513, 1973.

The EHRLICHGRAYINCREMENT algorithm obviously runs in $O(1)$ time, even in the worst case. Here's the algorithm in action with $n = 4$. The first line is the Gray bit-vector G, and the second line shows the focus vector F, both in reverse order:

$$G: 0000, 0001, 0011, 0010, 0110, 0111, 0101, 0100, 1100, 1101, 1111, 1110, 1010, 1011, 1001, 1000$$
$$F: 3210, 3211, 3220, 3212, 3310, 3311, 3230, 3213, 4210, 4211, 4220, 4212, 3410, 3411, 3240, 3214$$

Voodoo! I won't explain in detail how Ehrlich's algorithm works, except to point out the following invariant. Let $B[i]$ denote the ith bit in the *standard* binary representation of the current number. ***If $B[j] = 0$ and $B[j-1] = 1$, then $F[j]$ is the smallest integer $k > j$ such that $B[k] = 1$; otherwise, $F[j] = j$.*** Got that?

But wait — this algorithm only handles increments; what if we also want to decrement? Sorry, I don't have a clue. Extra credit, anyone?

14.5 Generalities and Warnings

Although computer scientists usually apply amortized analysis to understand the efficiency of maintaining and querying data structures, you should remember that amortization can be applied to *any* sequence of numbers. Banks have been using amortization to calculate fixed payments for interest-bearing loans for centuries. The IRS allows taxpayers to amortize business expenses or gambling losses across several years for purposes of computing income taxes. Some cell phone contracts let you to apply amortization to calling time, by rolling unused minutes from one month into the next month.

It's also important to remember that ***amortized time bounds are not unique***. For a data structure that supports multiple operations, different amortization schemes can assign different costs to *exactly the same* algorithms. For example, consider a generic data structure that can be modified by three algorithms: FOLD, SPINDLE, and MUTILATE. One amortization scheme might imply that FOLD and SPINDLE each run in $O(\log n)$ amortized time, while MUTILATE runs in $O(n)$ amortized time. Another scheme might imply that FOLD runs in $O(\sqrt{n})$ amortized time, while SPINDLE and MUTILATE each run in $O(1)$ amortized time. These two results are not necessarily inconsistent! Moreover, there is no general reason to prefer one of these sets of amortized time bounds over the other; our preference may depend on the context in which the data structure is used.

Exercises

1. Suppose we are maintaining a data structure under a series of n operations. Let $f(k)$ denote the actual running time of the kth operation. For each of the following functions f, determine the resulting amortized cost of a single operation. (For practice, try *all* of the methods described in this note.)

   (a) $f(k)$ is the largest integer i such that 2^i divides k.

   (b) $f(k)$ is the largest power of 2 that divides k.

   (c) $f(k) = n$ if k is a power of 2, and $f(k) = 1$ otherwise.

   (d) $f(k) = n^2$ if k is a power of 2, and $f(k) = 1$ otherwise.

   (e) $f(k)$ is the index of the largest Fibonacci number that divides k.

   (f) $f(k)$ is the largest Fibonacci number that divides k.

   (g) $f(k) = k$ if k is a Fibonacci number, and $f(k) = 1$ otherwise.

   (h) $f(k) = k^2$ if k is a Fibonacci number, and $f(k) = 1$ otherwise.

(i) $f(k)$ is the largest integer whose square divides k.

(j) $f(k)$ is the largest perfect square that divides k.

(k) $f(k) = k$ if k is a perfect square, and $f(k) = 1$ otherwise.

(l) $f(k) = k^2$ if k is a perfect square, and $f(k) = 1$ otherwise.

(m) Let T be a *complete* binary search tree, storing the integer keys 1 through n. $f(k)$ is the number of ancestors of node k.

(n) Let T be a *complete* binary search tree, storing the integer keys 1 through n. $f(k)$ is the number of descendants of node k.

(o) Let T be a *complete* binary search tree, storing the integer keys 1 through n. $f(k)$ is the *square* of the number of ancestors of node k.

(p) Let T be a *complete* binary search tree, storing the integer keys 1 through n. $f(k) = \mathit{size}(k) \lg \mathit{size}(k)$, where $\mathit{size}(k)$ is the number of descendants of node k.

(q) Let T be an *arbitrary* binary search tree, storing the integer keys 0 through n. $f(k)$ is the length of the path in T from node $k-1$ to node k.

(r) Let T be an *arbitrary* binary search tree, storing the integer keys 0 through n. $f(k)$ is the *square* of the length of the path in T from node $k-1$ to node k.

(s) Let T be a *complete* binary search tree, storing the integer keys 0 through n. $f(k)$ is the *square* of the length of the path in T from node $k-1$ to node k.

2. Consider the following modification of the standard algorithm for incrementing a binary counter.

```
INCREMENT(B[0..∞]):
  i ← 0
  while B[i] = 1
      B[i] ← 0
      i ← i + 1
  B[i] ← 1
  SOMETHINGELSE(i)
```

The only difference from the standard algorithm is the function call at the end, to a black-box subroutine called SomethingElse.

Suppose we call Increment n times, starting with a counter with value 0. The amortized time of each Increment clearly depends on the running time of SomethingElse. Let $T(i)$ denote the worst-case running time of SomethingElse(i). For example, we proved in class that Increment algorithm runs in $O(1)$ amortized time when $T(i) = 0$.

(a) What is the amortized time per Increment if $T(i) = 42$?

(b) What is the amortized time per Increment if $T(i) = 2^i$?

(c) What is the amortized time per Increment if $T(i) = 4^i$?

(d) What is the amortized time per Increment if $T(i) = \sqrt{2}^i$?

(e) What is the amortized time per Increment if $T(i) = 2^i/(i+1)$?

3. An ***extendable array*** is a data structure that stores a sequence of items and supports the following operations.

- AddToFront(x) adds x to the *beginning* of the sequence.
- AddToEnd(x) adds x to the *end* of the sequence.
- Lookup(k) returns the kth item in the sequence, or Null if the current length of the sequence is less than k.

Describe a ***simple*** data structure that implements an extendable array. Your AddToFront and AddToBack algorithms should take $O(1)$ *amortized* time, and your Lookup algorithm should take $O(1)$ *worst-case* time. The data structure should use $O(n)$ space, where n is the ***current*** length of the sequence.

4. An ***ordered stack*** is a data structure that stores a sequence of items and supports the following operations.

   - OrderedPush(x) removes all items smaller than x from the beginning of the sequence and then adds x to the beginning of the sequence.
   - Pop deletes and returns the first item in the sequence (or Null if the sequence is empty).

   Suppose we implement an ordered stack with a simple linked list, using the obvious OrderedPush and Pop algorithms. Prove that if we start with an empty data structure, the amortized cost of each OrderedPush or Pop operation is $O(1)$.

5. A *multistack* consists of an infinite series of stacks $S_0, S_1, S_2, \ldots$, where the ith stack S_i can hold up to 3^i elements. The user always pushes and pops elements from the smallest stack S_0. However, before any element can be pushed onto any full stack S_i, we first pop all the elements off S_i and push them onto stack S_{i+1} to make room. (Thus, if S_{i+1} is already full, we first recursively move all its members to S_{i+2}.) Similarly, before any element can be popped from any empty stack S_i, we first pop 3^i elements from S_{i+1} and push them onto S_i to make room. (Thus, if S_{i+1} is already empty, we first recursively fill it by popping elements from S_{i+2}.) Moving a single element from one stack to another takes $O(1)$ time.

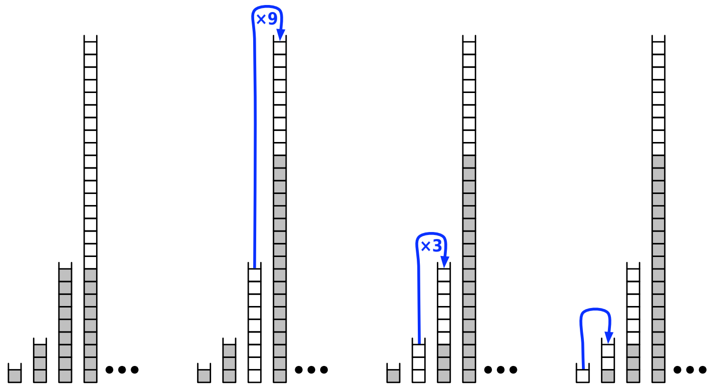
   Here is pseudocode for the multistack operations MSPush and MSPop. The internal stacks are managed with the subroutines Push and Pop.

```
MPush(x):
  i ← 0
  while S_i is full
      i ← i + 1
  while i > 0
      i ← i − 1
      for j ← 1 to 3^i
          Push(S_{i+1}, Pop(S_i))
  Push(S_0, x)
```

```
MPop(x):
  i ← 0
  while S_i is empty
      i ← i + 1
  while i > 0
      i ← i − 1
      for j ← 1 to 3^i
          Push(S_i, Pop(S_{i+1}))
  return Pop(S_0)
```

   (a) In the worst case, how long does it take to push one more element onto a multistack containing n elements?

   (b) Prove that if the user never pops anything from the multistack, the amortized cost of a push operation is $O(\log n)$, where n is the maximum number of elements in the multistack during its lifetime.



Making room in a multistack, just before pushing on a new element.

(c) Prove that in any intermixed sequence of pushes and pops, each push or pop operation takes $O(\log n)$ amortized time, where n is the maximum number of elements in the multistack during its lifetime.

6. Recall that a standard (FIFO) queue maintains a sequence of items subject to the following operations.

   - PUSH(x): Add item x to the end of the sequence.
   - PULL(): Remove and return the item at the beginning of the sequence.

   It is easy to implement a queue using a doubly-linked list and a counter, so that the entire data structure uses $O(n)$ space (where n is the number of items in the queue) and the worst-case time per operation is $O(1)$.

   (a) Now suppose we want to support the following operation instead of PULL:

   - MULTIPULL(k): Remove the first k items from the front of the queue, and return the kth item removed.

   Suppose we use the obvious algorithm to implement MULTIPULL:

```
MULTIPULL(k):
  for i ← 1 to k
      x ← PULL()
  return x
```

   Prove that in any intermixed sequence of PUSH and MULTIPULL operations, the amortized cost of each operation is $O(1)$

   (b) Now suppose we *also* want to support the following operation instead of PUSH:

   - MULTIPUSH(x, k): Insert k copies of x into the back of the queue.

   Suppose we use the obvious algorithm to implement MULTIPUUSH:

```
MULTIPUSH(k, x):
  for i ← 1 to k
      PUSH(x)
```

   Prove that for any integers ℓ and n, there is a sequence of ℓ MULTIPUSH and MULTIPULL operations that require $\Omega(n\ell)$ time, where n is the maximum number of items in the queue at any time. Such a sequence implies that the amortized cost of each operation is $\Omega(n)$.

(c) Describe a data structure that supports arbitrary intermixed sequences of MULTIPUSH and MULTIPULL operations in $O(1)$ amortized cost per operation. Like a standard queue, your data structure should use only $O(1)$ space per item.

7. Recall that a standard (FIFO) queue maintains a sequence of items subject to the following operations.

   - PUSH(x): Add item x to the end of the sequence.
   - PULL(): Remove and return the item at the beginning of the sequence.
   - SIZE(): Return the current number of items in the sequence.

   It is easy to implement a queue using a doubly-linked list, so that it uses $O(n)$ space (where n is the number of items in the queue) and the worst-case time for each of these operations is $O(1)$.

   Consider the following new operation, which removes every tenth element from the queue, starting at the beginning, in $\Theta(n)$ worst-case time.

```
DECIMATE():
  n ← SIZE()
  for i ← 0 to n − 1
      if i mod 10 = 0
          PULL()    ⟨⟨result discarded⟩⟩
      else
          PUSH(PULL())
```

   Prove that in any intermixed sequence of PUSH, PULL, and DECIMATE operations, the amortized cost of each operation is $O(1)$.

8. Chicago has many tall buildings, but only some of them have a clear view of Lake Michigan. Suppose we are given an array $A[1..n]$ that stores the height of n buildings on a city block, indexed from west to east. Building i has a good view of Lake Michigan if and only if every building to the east of i is shorter than i.

   Here is an algorithm that computes which buildings have a good view of Lake Michigan. What is the running time of this algorithm?

```
GOODVIEW(A[1..n]):
  initialize a stack S
  for i ← 1 to n
      while (S not empty and A[i] > A[TOP(S)])
          POP(S)
      PUSH(S, i)
  return S
```

9. Suppose we can insert or delete an element into a hash table in $O(1)$ time. In order to ensure that our hash table is always big enough, without wasting a lot of memory, we will use the following global rebuilding rules:

   - After an insertion, if the table is more than $3/4$ full, we allocate a new table twice as big as our current table, insert everything into the new table, and then free the old table.

- After a deletion, if the table is less than 1/4 full, we allocate a new table half as big as our current table, insert everything into the new table, and then free the old table.

Show that for any sequence of insertions and deletions, the amortized time per operation is still $O(1)$. *[Hint: Do not use potential functions.]*

10. Professor Pisano insists that the size of any hash table used in his class must always be a Fibonacci number. He insists on the following variant of the previous global rebuilding strategy. Suppose the current hash table has size F_k.

    - After an insertion, if the number of items in the table is F_{k-1}, we allocate a new hash table of size F_{k+1}, insert everything into the new table, and then free the old table.
    - After a deletion, if the number of items in the table is F_{k-3}, we allocate a new hash table of size F_{k-1}, insert everything into the new table, and then free the old table.

    Show that for any sequence of insertions and deletions, the amortized time per operation is still $O(1)$. *[Hint: Do not use potential functions.]*

11. Remember the difference between stacks and queues? Good.

    (a) Describe how to implement a queue using two stacks and $O(1)$ additional memory, so that the amortized time for any enqueue or dequeue operation is $O(1)$. The *only* access you have to the stacks is through the standard subroutines PUSH and POP.

    (b) A *quack* is a data structure combining properties of both stacks and queues. It can be viewed as a list of elements written left to right such that three operations are possible:

    - QUACKPUSH(x): add a new item x to the left end of the list;
    - QUACKPOP(): remove and return the item on the left end of the list;
    - QUACKPULL(): remove the item on the right end of the list.

    Implement a quack using *three* stacks and $O(1)$ additional memory, so that the amortized time for any QUACKPUSH, QUACKPOP, or QUACKPULL operation is $O(1)$. In particular, each element in the quack must be stored in *exactly one* of the three stacks. Again, you ***cannot*** access the component stacks except through the interface functions PUSH and POP.

12. Let's glom a whole bunch of earlier problems together. Yay! An ***random-access double-ended multi-queue*** or ***radmuque*** (pronounced "rad muck") stores a sequence of items and supports the following operations.

    - MULTIPUSH(x, k) adds k copies of item x to the *beginning* of the sequence.
    - MULTIPOKE(x, k) adds k copies of item x to the *end* of the sequence.
    - MULTIPOP(k) removes k items from the *beginning* of the sequence and retrns the last item removed. (If there are less than k items in the sequence, remove them all and return NULL.)
    - MULTIPULL(k) removes k items from the *end* of the sequence and retrns the last item removed. (If there are less than k items in the sequence, remove them all and return NULL.)
    - LOOKUP(k) returns the kth item in the sequence. (If there are less than k items in the sequence, return NULL.)

Describe and analyze a simple data structure that supports these operations using $O(n)$ space, where n is the current number of items in the sequence. LOOKUP should run in $O(1)$ *worst-case* time; all other operations should run in $O(1)$ *amortized* time.

13. Suppose you are faced with an infinite number of counters x_i, one for each integer i. Each counter stores an integer mod m, where m is a fixed global constant. All counters are initially zero. The following operation increments a single counter x_i; however, if x_i overflows (that is, wraps around from c to 0), the adjacent counters x_{i-1} and x_{i+1} are incremented recursively.

```
NUDGE_m(i):
  x_i ← x_i + 1
  while x_i ≥ m
      x_i ← x_i − m
      NUDGE_m(i − 1)
      NUDGE_m(i + 1)
```

   (a) Prove that NUDGE_3 runs in $O(1)$ amortized time. *[Hint: Prove that* NUDGE_3 *always halts!]*

   (b) What is the worst-case total time for n calls to NUDGE_2, if all counters are initially zero?

14. Now suppose you are faced with an infinite two-dimensional grid of modular counters, one counter $x_{i,j}$ for every pair of integers (i, j). Again, all counters are initially zero. The counters are modified by the following operation, where m is a fixed global constant:

```
2DNUDGE_m(i, j):
  x_{i,j} ← x_i + 1
  while x_{i,j} ≥ m
      x_{i,j} ← x_{i,j} − m
      2DNUDGE_m(i − 1, j)
      2DNUDGE_m(i, j + 1)
      2DNUDGE_m(i + 1, j)
      2DNUDGE_m(i, j − 1)
```

   (a) Prove that 2DNUDGE_5 runs in $O(1)$ amortized time.

   ★(b) Prove or disprove: 2DNUDGE_4 also runs in $O(1)$ amortized time.

   ★(c) Prove or disprove: 2DNUDGE_3 always halts.

*15. Suppose instead of powers of two, we represent integers as the sum of Fibonacci numbers. In other words, instead of an array of bits, we keep an array of *fits*, where the ith least significant fit indicates whether the sum includes the ith Fibonacci number F_i. For example, the fitstring 101110_F represents the number $F_6 + F_4 + F_3 + F_2 = 8 + 3 + 2 + 1 = 14$. Describe algorithms to increment and decrement a single fitstring in constant amortized time. *[Hint: Most numbers can be represented by more than one fitstring!]*

*16. A *doubly lazy binary counter* represents any number as a weighted sum of powers of two, where each weight is one of *four* values: −1, 0, 1, or 2. (For succinctness, I'll write $\bar{1}$ instead of −1.) Every integer—positive, negative, or zero—has an infinite number of doubly lazy binary representations. For example, the number 13 can be represented as 1101 (the standard binary representation), or

$2\cancel{1}01$ (because $2 \cdot 2^3 - 2^2 + 2^0 = 13$) or $10\cancel{1}1\cancel{1}$ (because $2^4 - 2^2 + 2^1 - 2^0 = 13$) or $\cancel{1}1200010\cancel{1}1\cancel{1}$ (because $-2^{10} + 2^9 + 2 \cdot 2^8 + 2^4 - 2^2 + 2^1 - 2^0 = 13$).

To increment a doubly lazy binary counter, we add 1 to the least significant bit, then carry the rightmost 2 (if any). To decrement, we subtract 1 from the lest significant bit, and then borrow the rightmost $\cancel{1}$ (if any).

```
LazyIncrement(B[0..n]):
  B[0] ← B[0] + 1
  for i ← 1 to n − 1
      if B[i] = 2
          B[i] ← 0
          B[i + 1] ← B[i + 1] + 1
          return
```

```
LazyDecrement(B[0..n]):
  B[0] ← B[0] − 1
  for i ← 1 to n − 1
      if B[i] = −1
          B[i] ← 1
          B[i + 1] ← B[i + 1] − 1
          return
```

For example, here is a doubly lazy binary count from zero up to twenty and then back down to zero. The bits are written with the least significant bit $B[0]$ on the *right*, omitting all leading 0's.

$$0 \xrightarrow{++} 1 \xrightarrow{++} 10 \xrightarrow{++} 11 \xrightarrow{++} 20 \xrightarrow{++} 101 \xrightarrow{++} 110 \xrightarrow{++} 111 \xrightarrow{++} 120 \xrightarrow{++} 201 \xrightarrow{++} 210$$

$$\xrightarrow{++} 1011 \xrightarrow{++} 1020 \xrightarrow{++} 1101 \xrightarrow{++} 1110 \xrightarrow{++} 1111 \xrightarrow{++} 1120 \xrightarrow{++} 1201 \xrightarrow{++} 1210 \xrightarrow{++} 2011 \xrightarrow{++} 2020$$

$$\xrightarrow{--} 2011 \xrightarrow{--} 2010 \xrightarrow{--} 2001 \xrightarrow{--} 2000 \xrightarrow{--} 20\cancel{1}1 \xrightarrow{--} 2\cancel{1}10 \xrightarrow{--} 2\cancel{1}01 \xrightarrow{--} 1100 \xrightarrow{--} 11\cancel{1}1 \xrightarrow{--} 1010$$

$$\xrightarrow{--} 1001 \xrightarrow{--} 1000 \xrightarrow{--} 10\cancel{1}1 \xrightarrow{--} 1\cancel{1}10 \xrightarrow{--} 1\cancel{1}01 \xrightarrow{--} 100 \xrightarrow{--} 1\cancel{1}1 \xrightarrow{--} 10 \xrightarrow{--} 1 \xrightarrow{--} 0$$

Prove that for any intermixed sequence of increments and decrements of a doubly lazy binary number, starting with 0, the amortized time for each operation is $O(1)$. Do *not* assume, as in the example above, that all the increments come before all the decrements.