Analysis Clinic

November 2015

1 Useful References

1.1 Notes

The most important part of your analysis is not your final answer, but your reasoning! Even with the correct runtime complexity, you need to show *why* it is correct. Remember to use functional notation and to define what your variables mean; for example: $O([b \rightarrow b^2])$ where b is the length of the input list.

We've included some links about Latex, which is a markup language to help you format mathematical expressions. ShareLatex is an online text editor that lets you write documents in Latex (which you may find helpful when writing up your solutions to Amortized). If you take the time to learn it now, you'll find your life much easier in future classes!

1.2 Readings and Resources

Chapter on "Halloween Analysis":
`http://papl.cs.brown.edu/2015/amortized-analysis.html`

"Amortized Analysis Explained":
`https://www.cs.princeton.edu/~fiebrink/423/AmortizedAnalysisExplained_Fiebrink.pdf`

1.3 Latex

ShareLatex: `https://www.sharelatex.com/`

Essential Latex: `http://for.mat.bham.ac.uk/R.W.Kaye/latex/el2e.pdf`

Latex Symbols:
`http://cs.brown.edu/about/system/managed/latex/doc/symbols.pdf`

2 Analysis Warm-up: Reverse

In class, we discussed that we could use reverse for our queue structure. We agreed that we could take linear time to reverse a list, but not all implementations of reverse take linear time! Determine the worst-case runtime complexities of the following two implementations of reverse. Once you're done, can you write a new version of `reverse` that performs better (i.e. its runtime decreases less rapidly with an increase in the length of the list)? Justify your solution with an analysis.

2.1 First Implementation of Reverse

```
fun reverse<A>(lst :: List<A>) -> List<A>:
  doc: "Reverses a list, lst"
  cases(List) lst:
    | empty => empty
    | link(f, r) => reverse(r) + [list: f]
  end
where:
  reverse(empty) is empty
  reverse([list: 1, 2, 3]) is [list: 3, 2, 1]
  reverse([list: 1, 2, 3, 4, 5]) is [list: 5, 4, 3, 2, 1]
end
```

2.2 Second Implementation of Reverse

```
fun reverse<A>(lst :: List<A>) -> List<A>:
  doc: "Reverses a list, lst"
  cases(List) lst:
    | empty => empty
    | link(_,_) =>
      reverse-help(lst, lst.length() - 1)
  end
where:
  reverse(empty) is empty
  reverse([list: 1, 2, 3]) is [list: 3, 2, 1]
  reverse([list: 1, 2, 3, 4, 5]) is [list: 5, 4, 3, 2, 1]
end

fun reverse-help<A>(lst :: List<A>, n :: Number) -> List<A>:
  doc: ```
       Creates a list of all the elements in non-empty
       list lst before position n in reverse order
       ```
```
```

```
    if n == 0:
      link(lists.get(lst, n), empty)
    else:
      link(lists.get(l, n), reverse-help(l, n - 1))
    end
where:
  reverse-help([list: 1, 2, 3, 4, 5], 4) is [list: 5, 4, 3, 2, 1]
  reverse-help([list: 1, 2, 3, 4, 5], 2) is [list: 3, 2, 1]
  reverse-help([list: 1, 2, 3, 4, 5], 0) is [list: 1]
end
```

3 This Is Halloween (Analysis)

After you've collected your haul from trick-or-treating, you'd like to analyze the candy you collected from around the neighborhood. You noticed that there was a linear relationship between house addresses and the amounts of candy people gave out. In particular, house k gave k candies. We can say that the function mapping house number to candy is in the family $O([k \to k])$, where k is the house number.

Let h be the total number of houses you visited, and assume that the house numbers were $1, 2, 3, \ldots h$. What was the amortized candy-per-house?

Suppose instead that the relationship between house address and amount of candy was quadratic. In particular, house k gave k^2 candies. What was the amortized candy-per-house?

What do you notice about the relationship between the mapping (address $\to$ candy) and the amortized complexity?

4 Worked Example of Amortized Analysis

Because you want to see some real world examples of amortized analysis, you realize that the best way to manage your Halloween candy is to store it as a queue. Remember that a queue is a first-in-first-out (FIFO) data structure, so the first element you enqueue is the first to be dequeued. One way to implement a queue is with two lists, a head and a tail. Lists are last-in-first-out (LIFO) structures. You decide that the tail will simply keep track of the candies as they are enqueued (in LIFO order), but the head will be reversed to mimic the queue's FIFO order. On a list, you have the operations first, rest, link, and is-empty. Assume that each of these has a cost of 1.

Specifically, you implement the operations for your queue as:
`enqueue(candy)`: link the candy to the tail

`dequeue()`: if the head is empty, then for each element in the tail, take the first candy from the tail and link it with the head. This has the effect of reversing the candies in the tail and storing them in the head. Finally, return the first candy of the head.

For a more detailed (and Pyret-specific) description of this strategy, refer to the "Halloween Analysis" chapter. It is a helpful exercise to understand why this implementation results in the correct functionality of a queue!

If you were doing a conventional runtime analysis, you might argue that dequeuing takes time in the family $O([k \rightarrow k])$, where k is the number of candies in the queue. In the worst case, any given dequeue operation might require moving all of the candies from the tail to the head. Therefore, the worst-case time of any operation is the time it takes to reverse, which is linear in the length of the list, or the number of candies in the queue. In practice, however, you only need to perform the transfer from the tail to the head if the head happens to be empty. You get a more interesting picture if you consider the amortized cost per operation.

Problem: If you perform k enqueues followed by k dequeues, what is the amortized complexity per operation?

Solution: Each of the enqueues takes one step, linking a candy to the tail. On the first dequeue, the head is empty, so transferring candies from the tail to the head takes steps proportional to the k elements in the tail. For the rest of the $k-1$ dequeues, all k candies have already been moved to the head in the proper order. Thus, the rest of the $k-1$ dequeues each take two steps, checking if the head is empty and then returning the first candy in the head.

The total number of steps is $1 + 1 + \ldots + 1$ (k times) for the enqueues, k for the first dequeue, and $1+1+\ldots+1$ ($k-1$ times) for the last $k-1$ dequeues. Hence, the total is $k+k+(k-1) = 3k-1$ steps. Because these $3k-1$ steps are spread out over $2k$ operations (k enqueues and k dequeues), the amortized cost per operation is $\frac{3k-1}{2k}$, which is in the family of constant functions. Therefore, the amortized complexity per operation is in the family $O([k \rightarrow 1])$, where k is the number of enqueues and dequeues.

5 Trick or Treat

5.1

Suppose we define a new operation, `trick-or-treat`, as follows. Note that `trick-or-treat` requires there to be x candies in the queue before it is called.

```
trick-or-treat(x):
dequeue $x$ candies from the queue
enqueue 1 candy into the queue
```

If you perform a sequence of m `trick-or-treats`, what is the amortized cost per `trick-or-treat`? What is the total cost of the sequence of m operations?

6 Counting Candy

6.1

Now you'd like to count the Halloween candy you collected, and you decide it'll be a great idea to do so using a binary counter. You'll store this counter in an ordered collection S of stone tablets with a "1" on one side and "0" on the other, so that each element in S can be thought of as a digit of the binary number (either 0 or 1). All of the elements on the tablets initially start at 0, and at each step, you just need to increment the counter by 1. Stone tablets are heavy, and flipping one costs a fixed unit of work.

For example, incrementing $(0 \to 1)$ costs 1, $(01 \to 10)$ costs 2, $(10 \to 11)$ costs 1, $(011 \to 100)$ costs 3, $(100 \to 101)$ costs 1, $(101 \to 110)$ costs 2, and so on. Note that whenever the number of digits changes, the leading digits are all initially 0, so you never need to go out and collect more stone tablets.

What is the worst-case cost per increment? What is the amortized cost per increment?

6.2

Define S_i to be the ith tablet in your collection (which is zero-indexed). Imagine a version of the counter we just discussed in which it costs 2^k to flip the bit S_k. For example, incrementing $(0 \to 1)$ costs 1, $(01 \to 10)$ costs 3 (1 for S_0 and 2 for S_1), $(10 \to 11)$ costs 1, $(011 \to 100)$ costs 7, $(100 \to 101)$ costs 1, $(101 \to 110)$ costs 3, and so on.

What is the worst-case cost per increment? What is the amortized cost per increment? Remember that $\log_2 2^k = k$.