Algorithm Design and Analysis

Amortized Analysis (The Potential Function Method)

Roadmap for today

- Learn about (or review) **amortized analysis**
- See the ***method of potential functions*** for amortized analysis
- Practice amortized analysis on dynamic arrays / lists

Arrays and Lists (aka *Dynamic arrays*)

- **Array**: A *fixed-size* container of items with constant-time access
- **List**: A container supporting constant-time access *and append*
  - ***Initialize****(): Creates an empty list*
  - ***Append****(x): Insert x at the end of the list*
  - ***Get****(i): Return the* i^{th} *element of the list*

Naïve list algorithm: Append creates a new array of length $n + 1$ and copies over every element of the old list. **Cost:** $O(n)$

Array-doubling List

Doubling algorithm:

- Maintain an array of some *capacity* $c \geq 1$
- The first n slots contain the list items, so $c \geq n$
- To append:
  - If $c = n$, allocate a new array with capacity $c' = 2c$, then move existing items
  - Place the new item at position n and increment n

GROW OPERATION

Complexity:

- **Best case:** $O(1)$
- **Worst case**: $O(n)$

Amortized Analysis

Key idea: Analyze the cost of a **sequence of operations** on the data structure, instead of focusing on the cost of a **single operation**.

The aggregate method: Take a **worst-case sequence** of operations and compute the **total cost**. The amortized cost of each operation is the average cost, i.e., the total divided by the number of operations.

Example: If in any sequence of m operations, the total cost is at most 5m, then the amortized cost of an operation is at most 5

Cost model

- We need to choose a **cost model** to work with. We *could* just use the word RAM, except then we must deal with lots of unknown arbitrary constants (since the word RAM doesn't care about constants).
- **Array cost model:**
  - Writing a value into the array costs 1
  - Moving an item from one array to another costs 1
  - Everything else is free

The analysis

Lemma: The cost of any sequence of m append operations using the doubling algorithm is at most $3m$

- After m appends, c = $\left\lceil \lceil m \rceil \right\rceil$ (smallest power of 2 at least m)
- $\left\lceil \lceil m \rceil \right\rceil < 2m$
- The final grow operation costs $c/2$
- The previous grow costs $c/4$ and so on
- Total cost for grows is less than m + m/2 + m/4 + … < 2m
- Total cost for appends is m

The analysis

Theorem: The amortized cost (using the aggregate method) of append using the doubling algorithm is 3

Proof: The total cost of m *appends is* $3m$, therefore by the aggregate method, the amortized cost of append is 3

The Potential Function Method

The Potential Function Method

The potential method: We define a potential function Φ, where

$$\Phi : \{\text{data structure states}\} \to \mathbb{R}$$

Say that an operation takes the data structure from state S_{i-1} to S_i, then we define the *amortized cost* of operation i:

$$ac_i = c_i + \underbrace{\Phi(S_i) - \Phi(S_{i-1})}$$

Amortized cost (of operation i) — ac_i

Actual cost (of operation i) — c_i

Change in potential — $\Phi(S_i) - \Phi(S_{i-1})$

Why is this useful?

Claim: If $\Phi(S_m) \geq \Phi(S_0)$, then

$$\sum_{i=1}^{m} c_i \leq \sum_{i=1}^{m} ac_i$$

$$\sum_{i=1}^{m} ac_i = \sum_{i=1}^{m} (c_i + \Phi(S_i) - \Phi(S_{i-1}))$$

$$= \sum_{i=1}^{m} c_i + \Phi(S_m) - \Phi(S_0)$$

So $\sum_{i=1}^{m} c_i = \sum_{i=1}^{m} ac_i + \Phi(S_0) - \Phi(S_m)$

Analyzing lists using potentials

- ***Strategy***: Educated guessing + trial and error

Key idea:

- Cheap operations should increase the potential (their amortized cost will be higher than their actual cost)
- Expensive operations should decrease the potential (their amortized cost will be lower than their actual cost)

Analyzing lists using potentials

Lists: State = $\{n\text{: number of elements},\ c\text{: capacity}\}$

- Cheap operation: Insert item. Φ goes up

- Expensive operation: Grow! Φ goes down

- ***Useful trick***: Split complex operations into smaller “sub-operations”
  - We split append into “grow” and “insert”
  - Append is grow + insert
  - Insert always costs 1, no matter what, no cases needed
  - Results in fewer cases to consider

Analyzing lists using potentials

- Let's guess a potential function!
- Insert: *potential goes up,* grow: *potential goes down*

Guess #1

$$\Phi(\mathrm{n}, \mathrm{c}) = \quad \mathrm{n} - \mathrm{c}$$

Potential is always non-positive (why?)

Let's try to come up with a potential where the math is easier

Analyzing lists using potentials

Guess #2

$$\Phi(n, c) = \quad n - \frac{c}{2}$$

Non-negative for n ≥ 1. Why?

	ac	cost	$\Phi(S_{i-1})$	$\Phi(S_i)$	$\Delta\Phi$
Insert	2	1	n-c/2	(n+1)-c/2	1
Grow	n/2	n	n/2 = c/2	0	-n/2

Analyzing lists using potentials

Guess #3 (Correct)

$$\Phi(n,c) = 2(n - \frac{c}{2})$$

	ac	cost	$\Phi(S_{i-1})$	$\Phi(S_i)$	$\Delta\Phi$
Insert	3	1	2(n-c/2)	2(n+1-c/2)	2
Grow	0	n	n = c	0	-n

ac_append $\leq$ ac_insert + ac_grow = 3 + 0

The final potential

$$\Phi(n, c) = 2(n - \frac{c}{2})$$

$$\Phi(S_m) \geq \quad 0$$

$$\Phi(S_0) = \quad -1$$

A *more-dynamic* array

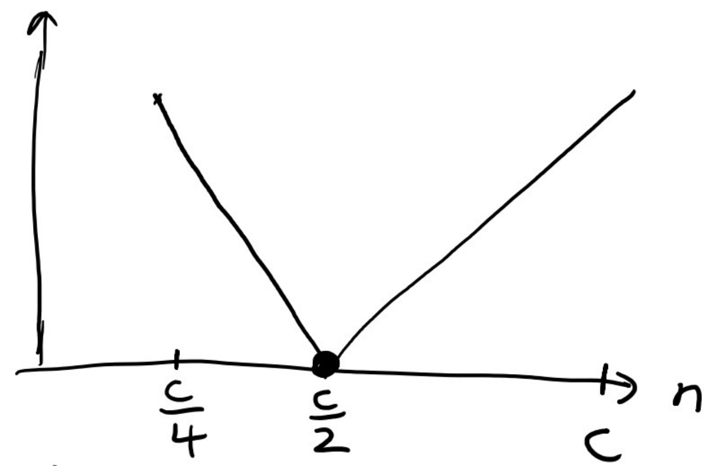
- We can add a "pop" operation to our list: removes the last item
- We want to not use space larger than $\Theta(n)$ to store a list of size n
    - ***append***(x)*: Insert x at the end of the list*
        - ***insert***(x)*: Place x in position* n*, then increment* n
        - ***grow***()*: Double the capacity* c *then move the* n *items*
    - ***pop***()*: Remove the last element of the list*
        - ***erase***()*: Erase the last item and decrement* n ⟸ **costs 1**
        - ***shrink***()*: Reduce the capacity* c *by half then move the n items*
- **Question**: *When* to shrink? n = c/4 (why not n = c/2?)
- **Property:** After a grow or shrink, $n = c/2$

Engineering the potential function

- **Design requirements:**
  - When appending above ½ capacity, potential goes up
  - Grow "spends" the potential and brings it back to zero
  - When popping below ½ capacity, potential goes up
  - Shrink "spends" the potential and brings it back to zero

$$\Phi(\mathrm{n}, \mathrm{c}) = 2\left(\mathrm{n} - \frac{\mathrm{c}}{2}\right) \quad \text{if } \mathrm{n} \geq \frac{\mathrm{c}}{2}$$

$$= \left(\frac{\mathrm{c}}{2} - n\right) \quad \text{if } \mathrm{n} < \frac{\mathrm{c}}{2}$$



The analysis

$$\Phi(n,c) = \begin{cases} 2\left(n - \frac{c}{2}\right) & \text{if } n \geq \frac{c}{2} \\ \left(\frac{c}{2} - n\right) & \text{if } n < \frac{c}{2} \end{cases}$$

	ac	c	$\Phi(S_{i-1})$	$\Phi(S_i)$	$\Delta\Phi$
Insert	≤ 3	1	-	-	≤ 2
Grow	0	n	n=c	0	-n
Erase	≤ 2	1	-	-	≤ 1
Shrink	0	n	n=c/4	0	-n

The result

$$\Phi(n,c) = \begin{cases} 2\left(n - \frac{c}{2}\right) & \text{if } n \geq \frac{c}{2} \\ \left(\frac{c}{2} - n\right) & \text{if } n < \frac{c}{2} \end{cases}$$

ac_pop $\leq$ ac_erase + ac_shrink $\leq$ 2 + 0

Wait! What about $\Phi(S_m)$ and $\Phi(S_0)$. Don't forget those!

$\Phi(S_m) \geq 0$, but $\Phi(S_0) = 1$ if say, we initialize with c = 2 and n = 0

So the total cost is at most 3 ·(# operations) + 1

Can just say initialization costs 1, so total cost $\leq$ 3·(# operations)

Summary

- Potential functions are useful, but tricky
- Designing them requires careful guessing and checking. Usually not obvious!
- Design potentials so that they **go up for cheap operations,** and **down for expensive operations**