1 Introduction

Amortized analysis is a technique for measuring the time needed to perform a sequence of operations on a data structure.

Your textbook describes three overlapping methods of amortized analysis:

- **Aggregate method**: Sum the total work across any sequence of $n$ operations, and divide by $n$.
- **Accounting method**: Add extra costs to early, less expensive operations, to “prepay” for later, more expensive operations.
- **Potential method**: Define a “potential function” on the complete data structure, and sum the actual cost with the change in potential.

We will focus only on the potential method, which is more powerful than the other two.

**Key idea**: Amortized analysis captures the idea that “expensive” operations are rare enough to be acceptable, by analyzing sequences rather than individual operations.

2 Example data structure: Multipop Stack

Consider a stack-like data structure with the following operations:

- **PUSH($x$)**
- **POP()**
- **MULTIPOP($k$)** – try to pop $k$ times, but stop if stack is empty.

How long does each operation take?

How long can a sequence of $n$ operations take?

3 Goal of amortized analysis

We want to assign an amortized cost to each operation.

Notation:
• Actual cost of operation $i$: $c_i$
• Amortized cost of operation $i$: $\hat{c}_i$

We need to guarantee that, for any sequence of $n$ operations,

$$\sum_{i=1}^{n} \hat{c}_i \geq \sum_{i=1}^{n} c_i.$$  

4 Potential method

Let $D_i \in D$ denote a ‘snapshot’ of the data structure after operation $i$.

1. Define a potential function $\Phi$ that maps data structure snapshots to real numbers.
   $$\Phi : D \rightarrow [0, \infty)$$

Intuition: The potential should represent the amount of “prepayment” that has been done.
   • Inexpensive, common operations generally increase the potential.
   • Expensive but infrequent operations generally decrease the potential.

5 Valid potential functions

2. Verify that that potential function has these two properties:
   • The initial data structure has zero potential:
     $$\Phi(D_0) = 0$$
   • The potential is never negative:
     $$\Phi(D_i) \geq 0 \text{ for all } i$$

6 Computing amortized costs

3. Compute the amortized cost of operation $i$ as the actual cost plus the change in potential:
   $$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$$
7 Why the potential method works

This process is useful because the sum telescopes:

\[ \sum_{i} \tilde{c}_i = \sum_{i=1}^{n} (c_i + \Phi(D_i) - \Phi(D_{i-1})) = \sum_{i=1}^{n} c_i + \Phi(D_n) - \Phi(D_0) = \sum_{i=1}^{n} c_i + \Phi(D_n) \geq \sum_{i=1}^{n} c_i. \]

For any sequence of operations, the actual cost is less than or equal to the amortized cost.

8 Multipop Stack: Potential method

1. Choose a potential function:
   \[ \Phi(S) = \text{number of items in stack } S \]

2. Verify that the potential function is valid:
   - Do we have \( \Phi(D_0) = 0 \)?
   - Do we have \( \Phi(D_i) \geq 0 \) for all \( i \)?

3. Compute amortized costs:
   - Push: \( \tilde{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = 1 + 1 = 2 = \Theta(1) \)
   - Pop: \( \tilde{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = 1 - 1 = 0 = \Theta(1) \)
   - Multipop:
     \[ \tilde{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = \min(k, s) - \min(k, s) = 0 = \Theta(1) \]

9 Example data structure: Dynamic tables

Consider an array-like data structure with these operations:
   - \text{INSERT AT END}(k)
   - \text{LOOKUP}(i)

Implement using arrays, and reallocating a new bigger array when needed.
### TABLE\textsc{INSERT}(x)

if $T$.size = 0 then
  allocate $T$.table with 1 slot
  $T$.size ← 1
else if $T$.num = $T$.size then
  allocate $N$ with $2T$.size slots
  insert all items from $T$.table into $N$
  free $T$.table
  $T$.table ← $N$
  $T$.size ← $2T$.size
end if
insert $x$ into $T$.table
$T$.num ← $T$.num + 1

---

#### 10 Dynamic tables: Potential method

1. Choose a potential function:

   $\Phi(T) = 2T$.num − $T$.size

2. Verify that the potential function is valid:

   - Do we have $\Phi(D_0) = 0$?
   - Do we have $\Phi(D_i) \geq 0$ for all $i$?

3. Compute amortized costs:

   - \textsc{INSERT} (elementary):
     \[ c_i + \Phi(D_i) - \Phi(D_{i-1}) = 1 + 2 = 3 = \Theta(1). \]
   - \textsc{INSERT} (reallocation):
     \[ \hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) \]
     \[ = n_i + (2n_i - s_i) - (2n_{i-1} - s_{i-1}) \]
     \[ = n_i + 2n_i - 2n_{i-1} - 2n_{i-1} + n_{i-1} \]
     \[ = 3n_i - 3n_{i-1} = 3 = \Theta(1) \]
   - \textsc{LOOKUP}:
     \[ c_i + \Phi(D_i) - \Phi(D_{i-1}) = 1 + 0 = 0 = \Theta(1). \]

   One way to understand this potential function is that we want something
   - equal to the table size when the table is full, and
   - zero right after the table is reallocated.

   That captures the idea that simple insert operations should increase the
   potential to ‘save up’ for the expensive reallocate step in the future.
   When computing the amortized costs, note that $s_{i-1} = n_{i-1}$ (since the table
   was full), and $n_i - n_{i-1} = 1$ (since we’ve inserted a single item).