Amortized Analysis

Understanding Average Cost Over Sequences

Prof. Antonio Khalil Moretti

SCIS 313: Data Structures and Algorithm Analysis

Spelman College

The Problem: When Worst-Case Doesn't Tell the Whole Story

Consider C++ vector's push_back():

```
vector<int> v;
for (int i = 0; i < n; i++) {
    v.push_back(i);  // What's the complexity?
}
```

Analysis attempt:

- Most push_back operations: $O(1)$ - just add to end
- Occasionally (when full): $O(n)$ - resize and copy all elements

Question: Is the total loop $O(n)$ or $O(n^2)$?

The Problem: When Worst-Case Doesn't Tell the Whole Story

Consider C++ vector's push_back():

```
vector<int> v;
for (int i = 0; i < n; i++) {
    v.push_back(i);  // What's the complexity?
}
```

Analysis attempt:

- Most push_back operations: $O(1)$ - just add to end
- Occasionally (when full): $O(n)$ - resize and copy all elements

Question: Is the total loop $O(n)$ or $O(n^2)$?

Answer: $O(n)$ total! But why?

Today: We'll prove it using *amortized analysis*

Three Ways to Analyze Algorithm Performance

Type	What It Measures
Worst-case	Maximum time for a single operation Example: One push_back could take $O(n)$
Average-case	Expected time assuming random input Requires probability distribution
Amortized	Average time per operation over worst-case sequence Guarantees performance over any sequence

Key distinction:

- **Average-case:** Assumes randomness in input
- **Amortized:** No assumptions - worst possible sequence, averaged

Amortized analysis gives stronger guarantees!

What is Amortized Analysis?

Definition: Average performance per operation over a worst-case sequence

Analogy: Gym membership

- Some days: Pay $100
- Most days: Pay $0
- 20 visits/month: Amortized = $5/visit

For algorithms:

- Some ops: Expensive $O(n)$
- Most ops: Cheap $O(1)$
- Over n ops: Amortized $O(1)$

Key: Individual operations can be expensive, but the average is cheap!

Dynamic Arrays: How They Grow

Problem: Arrays are fixed, we want dynamic

Solution: When full, double size

```
void push_back(int x) {
  if (size == capacity) {
    // Double capacity
    int* newArr =
      new int[capacity * 2];
    for (int i = 0; i < size; i++)
      newArr[i] = arr[i];
    delete[] arr;
    arr = newArr;
    capacity *= 2;
  }
  arr[size++] = x;
}
```

Analysis:

- Most times: $O(1)$
- When full: $O(n)$ to copy

Question:

Cost of n consecutive push_back operations?

Naive: $n \times O(n) = O(n^2)$?

Dynamic Arrays: How They Grow

Problem: Arrays are fixed, we want dynamic

Solution: When full, double size

```
void push_back(int x) {
  if (size == capacity) {
    // Double capacity
    int* newArr =
      new int[capacity * 2];
    for (int i = 0; i < size; i++)
      newArr[i] = arr[i];
    delete[] arr;
    arr = newArr;
    capacity *= 2;
  }
  arr[size++] = x;
}
```

Analysis:

- Most times: $O(1)$
- When full: $O(n)$ to copy

Question:

Cost of n consecutive push_back operations?

Naive: $n \times O(n) = O(n^2)$?

Actually: $O(n)$!

Dynamic Array Growth Pattern

Starting with capacity = 1, insert n elements:

Insert	Action	Cap	Copies
1	Insert	1	0
2	Resize to 2	2	1
3	Resize to 4	4	2
4	Insert	4	0
5	Resize to 8	8	4
6-8	Insert	8	0
9	Resize to 16	16	8
⋮	⋮	⋮	⋮

Pattern: Resize when size = power of 2

Copy costs: $1 + 2 + 4 + 8 + \cdots$ (geometric series)

Method 1: Aggregate Analysis

Sum total cost, divide by number of operations

Goal: Find total cost of n push_back operations, starting from empty array

Step 1: Count resize operations

Resizes happen at sizes: $1, 2, 4, 8, 16, \ldots, 2^k$ where $2^k \leq n < 2^{k+1}$

Step 2: Sum copy costs

$$\begin{aligned}\text{Total copies} &= 1 + 2 + 4 + 8 + \cdots + 2^k \\ &= 2^{k+1} - 1 \quad \text{(geometric series)} \\ &< 2 \cdot 2^k \\ &\leq 2n\end{aligned}$$

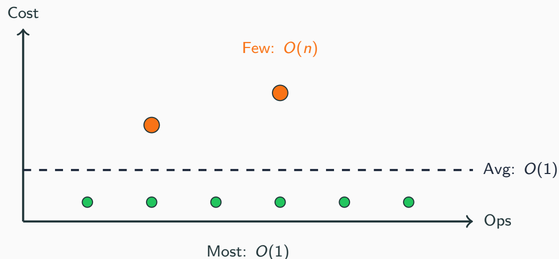
Step 3: Calculate total cost

$$\text{Total cost} = n \text{ insertions} + 2n \text{ copies} < 3n = O(n)$$

Amortized cost per operation: $O(n)/n = O(1)$

Aggregate Method: The Big Picture

What we proved:



Method:

1. Total cost $T(n)$
2. Amortized $= \frac{T(n)}{n}$

Pros: Simple

Cons: Same cost for all ops

Method 2: Accounting (Banker's) Method

Operations can save credit for future expensive operations

Idea: Assign different amortized costs to different operations

Rules:

1. Some operations **overpay** (amortized cost $>$ actual cost) $\rightarrow$ build credit
2. Some operations **use credit** (amortized cost $<$ actual cost)
3. **Credit must never go negative!**

Analogy: Bank account

- Cheap operations deposit money
- Expensive operations withdraw money
- Account balance must stay ≥ 0

If credit never goes negative, the amortized cost is an upper bound on actual cost

Accounting Method: Dynamic Array Example

Using "credits" to track costs

The accounting metaphor: Think of credits like money in a bank account

Assign amortized costs:

- Every push_back "pays" 3 credits
- Even though actual cost varies:
  - No resize: costs 1 (insert)
  - Resize: costs n (copy all)

Where do 3 credits go?

1. 1 credit: Pay for insert NOW
2. 2 credits: SAVE for future resize

We "overpay" by 2 to build savings!

When resize happens:

Resize from size n to $2n$:

- Need: n copies
- Have saved: n elements $\times$ 2 credits $= 2n$ credits
- Use n credits, keep n extra!

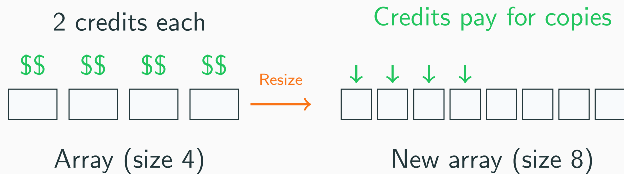
Key rule:

Credit balance ≥ 0 always!

This proves we saved enough.

Result: Amortized $= 3 = O(1)$

Accounting Method: Credit Flow



Key: "Overpaying" on cheap operations saves credit for expensive ones

Accounting Method: Stack with Multipop

Operations:

```
void push(int x);         // O(1)
int pop();                // O(1)
void multipop(int k);     // O(min(
    k,size))
```

Question: Cost of n operations?

Naive: Each multipop could be $O(n)$, so n ops $= O(n^2)$

Accounting Method: Stack with Multipop

Operations:

```
void push(int x);          // O(1)
int pop();                 // O(1)
void multipop(int k);      // O(min(
    k,size))
```

Question: Cost of n operations?

Naive: Each multipop could be $O(n)$, so n ops $= O(n^2)$

Better: Use accounting method

Assign amortized costs:

- **push:** 2 credits
  - 1: pay for push now
  - 1: save for future pop
- **pop/multipop:** 0 credits
  - Use saved credit

Total: n operations $\times$ 2 credits $= O(n)$

Amortized: $O(1)$ per operation!

Method 3: Potential Method

The most general and powerful method

Idea: Define potential function $\Phi =$ "stored energy"

Setup:

- $D_i =$ state after op i
- $\Phi(D_i) =$ potential
- $\Phi(D_0) = 0$ (initial)
- $\Phi(D_i) \geq 0$ always

Formula:

$$\hat{c}_i = c_i + \Delta\Phi$$

where $\Delta\Phi = \Phi(D_i) - \Phi(D_{i-1})$

Variables:

- $\hat{c}_i =$ amortized cost
- $c_i =$ actual cost
- $\Delta\Phi =$ change in potential

Interpretation:

- $\Delta\Phi > 0$: Storing credit
- $\Delta\Phi < 0$: Using credit
- $\Delta\Phi = 0$: No change

Potential Method: Total Cost

Sum amortized costs over n operations:

$$\begin{aligned}\sum_{i=1}^{n} \hat{c}_i &= \sum_{i=1}^{n} [c_i + \Phi(D_i) - \Phi(D_{i-1})] \\ &= \sum_{i=1}^{n} c_i + \sum_{i=1}^{n} [\Phi(D_i) - \Phi(D_{i-1})] \\ &= \sum_{i=1}^{n} c_i + [\Phi(D_n) - \Phi(D_0)] \\ &= \sum_{i=1}^{n} c_i + \Phi(D_n) \quad (\text{since } \Phi(D_0) = 0)\end{aligned}$$

If $\Phi(D_n) \geq 0$, then:

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

Amortized cost is an upper bound on actual cost!

Potential Method: Dynamic Array

Potential function: $\Phi(D_i) = 2 \times \text{size} - \text{capacity}$

Case 1: Insert without resize

Array has room, just add element.

- Actual cost: $c_i = 1$
- Size increases by 1, capacity unchanged
- Change: $\Delta\Phi = 2$
- Amortized: $\hat{c}_i = 1 + 2 = 3$

Result: $O(1)$

Potential builds up as we fill the array.

Case 2: Insert with resize

Array full, must double capacity.

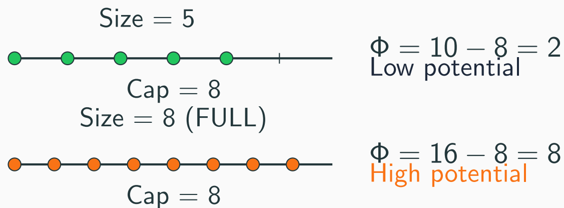
- Actual: $c_i = s + 1$ (copy s, insert 1)
- Before: size s, cap s, so $\Phi = s$
- After: size $s + 1$, cap $2s$, so $\Phi = 2$
- Change: $\Delta\Phi = 2 - s$
- Amortized: $\hat{c}_i = (s + 1) + (2 - s) = 3$

Result: $O(1)$

Potential drops, paying for expensive copy!

Potential Method: Intuition

Potential: $\Phi = 2 \times \text{size} - \text{capacity}$



Potential = "stored energy" that pays for expensive operations

Comparing the Three Methods

	Aggregate	**Accounting**	**Potential**
Approach	Sum total cost, divide by n	Assign credits to operations	Define potential function
Per-op cost?	All ops get same cost	Different ops can have different costs	Different ops can have different costs
Ease of use	Simplest	Intuitive	Most flexible
Best for	Simple problems	When operations have clear "cheap" and "expensive" distinction	Complex data structures

All three methods give the same answer!

Common Misconceptions

1. **"Amortized = average-case"**

✗ Average-case assumes random input; amortized assumes worst-case sequence

2. **"If amortized is O(1), every operation is O(1)"**

✗ Individual ops can be expensive; average over sequence is O(1)

3. **"Amortized analysis is just a trick"**

✗ It's rigorous math with provable guarantees

Where Amortized Analysis is Used

1. **Dynamic Arrays**
    - C++ `vector`, Java `ArrayList`, Python `list`
    - push_back is O(1) amortized
2. **Hash Tables**
    - When load factor gets too high, resize and rehash
    - Amortized O(1) insert even with resizing
3. **Union-Find (Disjoint Set)**
    - With path compression and union by rank
    - Operations are nearly O(1) amortized (actually $O(\alpha(n))$ where α is inverse Ackermann - basically constant)
    - We'll study this later!

Practice Problem 1

Binary Counter:

Consider a binary counter that starts at 0. We have one operation: `increment()`, which adds 1.

```
int counter[k] = {0, 0, 0, ..., 0};  // k bits
void increment() {
    int i = 0;
    while (i < k && counter[i] == 1) {
        counter[i] = 0;  // flip bit
        i++;
    }
    if (i < k) counter[i] = 1;
}
```

Question: What is the amortized cost of n increments?

Hint: Use aggregate analysis. How many times does each bit flip?

Practice Problem 1 - Solution

Aggregate analysis:

Over n increments (from 0 to $n-1$):

- Bit 0 flips every increment: n times
- Bit 1 flips every 2: $n/2$ times
- Bit 2 flips every 4: $n/4$ times
- Bit 3 flips every 8: $n/8$ times

Total flips: $n + \frac{n}{2} + \frac{n}{4} + \frac{n}{8} + \cdots < 2n$

Amortized cost: $\frac{2n}{n} = 2 = O(1)$

Though worst-case flips $O(\log n)$ bits, amortized is $O(1)$!

Practice Problem 2

Implement a queue using two stacks.

```
class Queue {
    Stack s1, s2;  // s1 for enqueue, s2 for dequeue
    void enqueue(int x) {
        s1.push(x);  // Always O(1)
    }
    int dequeue() {
        if (s2.isEmpty()) {
            while (!s1.isEmpty()) {
                s2.push(s1.pop());  // Move all elements
            }
        }
        return s2.pop();
    }
};
```

Question: What is the amortized cost of dequeue?

Practice Problem 2 - Solution

Accounting method:

Costs:

- **Enqueue:** 2 credits
  - 1: pay push to s1
  - 1: save for move to s2
- **Dequeue:** 1 credit
  - 1: pay pop from s2
  - Move uses saved credit

Why credit stays ≥ 0:

- Each element saves 1 credit when enqueued
- Move from s1$\rightarrow$s2 uses that credit
- Pop from s2 costs only 1

Both O(1) amortized!

Summary

Three methods:

1. **Aggregate:** Total cost / n operations (simple, same cost all ops)
2. **Accounting:** Assign costs, build credit (intuitive, credit ≥ 0)
3. **Potential:** $\hat{c}_i = c_i + \Delta\Phi$ (most general, $\Phi \geq 0$)

Key: Expensive ops are rare; average cost is low!

Applications: Dynamic arrays, hash tables, union-find, splay trees

What's Next

Today:

- Worst-case vs average-case vs amortized
- Three methods: aggregate, accounting, potential
- Dynamic arrays are O(1) amortized

Next: Balanced Trees (AVL)

- Why balancing matters
- Rotations
- Guaranteed O(log n)

Later: Union-Find with path compression (uses amortized analysis!)

Homework: Amortized analysis problems on Canvas