Today’s Lecture:
- Amortized Analysis
- Aggregate Method
- Accounting Method
- Potential Method
- Binary Counter
- Table Update

**Scenario:** Maintaining data structure over a sequence of \( n \) operation.

**Operation cost:** Cost per operation may be large (e.g. \( \Theta(n) \)).

**Total cost:** But total cost may not be as much as \( n \) \( \times \) (worst-case cost of one operation).

**Amortized analysis:** How to do tight analysis in such scenarios?

---

**Three techniques**

Today we will describe three techniques that can be used to analyze such scenarios.

- **Aggregate method**
- **Accounting method**
- **Potential method**

- Motivate by the “Binary Counter” example.
- Explain the differences.
- Move to “Dynamic Table” problem.

---

**Binary Counter**

**Data Structure:** \( k \)-bit array, counting numbers from 0 to \( 2^k - 1 \).

**Operation:** Increment.

**Cost:** \# of bits flipped.

(Toy problem! Motivates the ideas clearly!)

**Example:**

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td>Cost = 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00001</td>
<td>Cost = 2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00010</td>
<td>Cost = 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00111</td>
<td>Cost = 3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Notice: Costs vary! Max cost = \( \Theta(\log n) \).
**Aggregate method**

**Idea:** Count the total cost for \( n \) operations directly.

**Technique:** Ad-hoc. Possibly try to count from a different way.

Example

\[
\begin{align*}
0 & 0 & 0 & 0 & \text{Cost} = 1 \\
0 & 0 & 0 & 1 & \text{Cost} = 2 \\
0 & 0 & 1 & 0 & \text{Cost} = 3 \\
0 & 0 & 0 & 1 & \text{Cost} = 4 \\
\vdots & \ & \ & \ & \text{Cost} = 5 \\
1 & 1 & 1 & 1 & \\
\uparrow & \uparrow & \uparrow & \uparrow & \\
1 & 2 & 4 & 8 & 16 \leftarrow \text{Cost}
\end{align*}
\]

**Aggregate method (contd).**

- Count per row varies. Hard to sum up.
- Instead count by columns.
- I.e., count how many times \( i \)th bit is flipped.
- Add the costs.

For a sequence of \( n \) increments

\[
\text{ith l.s.b. flipped } \lceil n/2^i \rceil \text{ times.}
\]

\[
\begin{align*}
\text{Total Cost } &= \sum_{i=0}^{\lfloor \log_2 n \rfloor} \lceil n/2^i \rceil \\
&\leq \sum_{i=0}^{\lfloor \log_2 n \rfloor} n/2^i \\
&= n \sum_{i=0}^{\lfloor \log_2 n \rfloor} 1/2^i \\
&= 2n
\end{align*}
\]

Thus total cost = \( O(n) \).

Amortized cost per insertion = \( O(n)/n = O(1) \).

---

**Amortized cost**

Definition: Average cost of an operation over a sequence of \( n \) operations, maximized over all \( n \) and all sequences.

**Warning:**
Not average case analysis!
No assumptions about input sequence.
Amortization is still a worst-case principle!
In aggregate method, counted the total to bound the amortized cost. Usually do this the other way.

---

**Accounting Method**

- Data structure comes with a “bank account”.
- Every operation allotted a fixed $ cost (its amortized cost).
- If actual cost less than allotted amount, deposit extra $’s into bank.
- If actual cost more than allotted amount, withdraw from bank to pay for the operation.
- Prove: Always have a non-negative balance.
- Conclude: Sequence of \( n \) operations costs at most \( n \) times the amortized cost!
Example: Binary Counter

- Amortized Cost of flipping 0 → 1 = 2$
- Amortized Cost of flipping 1 → 0 = 0$

- When flipping 0 to 1: Pay 1$ for operation, and put 1$ in bank account (earmarked to the newly set bit).
- When flipping 1 to 0: Withdraw from the bank to pay for the operation.
- Invariant: Every 1 in the counter has 1$ earmarked for it in the bank account. So always have money to pay for the operation.
- Finally: Increment makes only one 0 → 1 flip. So amortized cost of increment = 2.

Potential Method

- In this method, we associate a potential energy with every data structure.
- Potential energy is "Potential to do damage".
- Amortized cost = actual cost + new potential - old potential.
- I.e., Must pay to increase the potential of the data structure.
- If operation has large actual cost but reduces the potential a lot, then amortized cost is low.
- How to find potential: Look for what makes a data structure bad.

Potential functions

- Basic rules:
- Must always be non-negative.
- Must start at zero.
- Implies a sequence of n operations cost at most n times the amortized cost.

Proof: Suppose n operations modify data structure from D₀ to D₁ to ... Dₙ.
Let Φ(D) be the potential of data structure D.
Let cᵢ be actual cost of ith operation.
Let  cᵢ be amortized cost of ith operation.

Then  cᵢ = cᵢ + Φ(Dᵢ) - Φ(Dᵢ₋₁).

Summing we get.

\[ \sum_{i=1}^{n} cᵢ = \sum_{i=1}^{n} cᵢ + \sum_{i=1}^{n} (\Phi(Dᵢ) - \Phi(Dᵢ₋₁)) \]
\[ = \sum_{i=1}^{n} cᵢ + \Phi(Dₙ) - \Phi(D₀) \]
\[ = \sum_{i=1}^{n} cᵢ + \Phi(Dₙ) \geq \sum_{i=1}^{n} cᵢ \]

Example: Binary Counter

Data structure is bad if it has a lot of 1’s in it.
Let Φ(COUNTER) = # 1’s in COUNTER.
Potential increase on increment
= # (0 → 1) flips - # (1 → 0) flips
= 1 - # (1 → 0) flips

Thus amortized cost of increment
= Actual cost + Potential increase
= (1 + # (1 → 0) flips + (1 - # (1 → 0) flips
= 2

Thus amortized cost = 2, and cost of n increments is at most 2n.
Example: Dynamic tables. (cont’d)

- Idea: “Grow” table with system call to allocate more memory. Reinsert old items.

Sequence of \( n \) Inserts:

\[
\text{Worst-case cost} = \Theta(n) \\
\text{Total cost} \leq n \cdot \Theta(n) = \Theta(n^2) = \Theta(n)
\]

Let \( c_i = \text{cost of } i^{th} \text{ Insert} \)

\[
= \begin{cases} 
  i & \text{if } i - 1 = 2^l \\
  1 & \text{if otherwise}
\end{cases}
\]

Aggregate Analysis:

\[
\text{Cost of } n \text{ inserts} = \sum_{i=1}^{n} c_i \\
\leq n + \sum_{j=1}^{\lfloor \log n \rfloor} 2^j \\
\leq 3 \cdot n
\]

Accounting analysis

- Charge each operation a fictitious amortized amount.
- Amount not immediately used is stored in bank.
- Later operations use bank reserve.
- Balance must not go negative.

Must have

\[
\sum_{i=1}^{n} c_i \leq \sum_{i=1}^{n} \hat{c}_i \forall n
\]

Dynamic table:

Charge \( \$3 \) for \( i^{th} \) Insert.

Amortized cost \( \hat{c}_i \)

\( \$1 \) pays for immediate Insert \( \$2 \) stored for later.

When table doubles:

\( \$1 \) reinserts item $1$ reinserts old item

\[
\begin{array}{c}
  0 \ 0 \ 0 \ 0 \\
  2 \ 2 \ 2 \ 2
\end{array}
\]

Therefore, balance is never negative.

Amortized costs are upper bound on true costs.

Advantage of accounting over aggregate:

- Each operation can be assigned a specific amortized cost.
- Note: Different amortized costs may work.

Expansion and Contraction

Inserts and Deletes

- Table overflows $$\Rightarrow$$ double
- Table $$< \frac{1}{2}$$ full $$\Rightarrow$$ halve (causes thrashing)
  $$\rightarrow \frac{1}{4}$$

Accounting analysis:

\[
\hat{c}_i = \begin{cases} 
  3 & \text{if Insert} \\
  2 & \text{if Delete}
\end{cases}
\]

Always can pay $$\Rightarrow$$ \( n \) operations cost \( O(n) \)

Potential analysis:

- Stored work viewed as potential energy of data structure.
Framework: Start with data structure $D_0$
operation $i$ transforms $D_{i-1}$ to $D_i$
cost of operation $i$ is $c_i$

Idea: Define potential function $\Phi: \{D_i\} \rightarrow \mathbb{R}$ such
that $\Phi(D_0) = 0$ and $\Phi(D_1) \geq 0 \ \forall i$.

Amortize cost $\hat{c}_i$ defined by:

$$\hat{c}_i = c_i + \frac{\Phi(D_i) - \Phi(D_{i-1})}{\text{potential difference} \ \Delta \Phi}$$

- If $\Delta \Phi > 0$, then $\hat{c}_i > c_i$, and operation $i$ stores
  work in data structure.
- If $\Delta \Phi < 0$, then $\hat{c}_i < c_i$, and operation $i$ delivers
  up work from data structure to help pay for $c_i$.

Total amortized cost of $n$ operations is

$$\sum_{i=1}^{n} \hat{c}_i = \sum_{i=1}^{n} \left( c_i + \Phi(D_i) - \Phi(D_{i-1}) \right)$$

$$= \sum_{i=1}^{n} c_i + \sum_{i=1}^{n} \left( \Phi(D_i) - \Phi(D_{i-1}) \right) \geq 0$$

So amortized costs upper bound true costs.

Key: Find useful potential function.

Example: Table doubling (Inserts only)

Defin: $\Phi(D_i) = 2i - 2^{[\log i]}$ (Assume $2^{[\log 0]} = 0$)

Note: $\Phi(D_0) = 0$ and $\Phi(D_i) \geq 0 \ \forall \ i$

ANOTHER TABLE of 0’s and 2’s.

$2(i - 2^{[\log i]} - 1)$

$2(6 - 2^{3} - 1) = 4$

Amortized cost of $i$th Insert

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$$

$$d_{ij}^{(0)} = \left\{ \begin{array}{ll}
0 & \text{if } i = j \\
\infty & \text{if } i \neq j
\end{array} \right. $$

$$= \left\{ \begin{array}{ll}
i + (2i - 2^{[\log i]})
- (2(i - 1) - 2^{[\log (i-1)]}) & \text{if } (i - 1) = 2^l \\
1 + (2i - 2^{[\log i]})
- (2(i - 1) - 2^{[\log (i-1)]}) & \text{if otherwise}
\end{array} \right. $$

- Case 1: $2^{[\log i]} = 2 \cdot 2^{[\log (i-1)]} = 2(i-1)$
  $$\Rightarrow \hat{c}_i = i + 2 - 2(i - 1) + (i - 1) = 3$$
- Case 2: $2^{[\log i]} = 2^{[\log (i-1)]]}$
  $$\Rightarrow \hat{c}_i = 1 + 2i - 2(i - 1) = 3$$

$n$ Inserts cost $\Theta(n)$ in worst case.

More complicated $\Phi$ for Deletes (see book).