Amortized Analysis
Amortized Analysis

Suppose we perform $n$ operations on a data structure, and the total running time is $T(n)$

The “average time” per operation is $T(n)/n$

- this is not the “average” in the sense of probability theory — there are no probabilities
- some individual operations may take much more time than the average time, and some much less
Example: incrementing a binary counter

<table>
<thead>
<tr>
<th>counter value</th>
<th>bit representation</th>
<th># of flips</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0 0 0 0 0 0 0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0 0 0 0 0 0 1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0 0 0 0 1 0 0</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>0 0 0 0 1 1 1</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>0 0 0 1 0 0 0</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>0 0 0 1 0 1 1</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>0 0 0 1 1 0 0</td>
<td>2</td>
</tr>
</tbody>
</table>

Running time = \(O(# \text{ of flips})\)

\# of flips = \# of low order 0-bits of result + 1

Crude analysis: \# flips per increment = \(O(\log n)\)

total cost = \(O(n \log n)\)
A better analysis

Total cost for $n$ increments:

\[
\begin{align*}
&n + \# \text{ of even numbers among } 1 \ldots n \\
&+ \# \text{ of multiples of 4 among } 1 \ldots n \\
&+ \# \text{ of multiples of 8 among } 1 \ldots n \\
&+ \cdots \\
&= \sum_{i=0}^{\infty} \left\lfloor \frac{n}{2^i} \right\rfloor \leq \sum_{i=0}^{\infty} \frac{n}{2^i} = n \sum_{i=0}^{\infty} \frac{1}{2^i} = 2n
\end{align*}
\]

More structured analysis:

- the accounting method
- potential functions
The Accounting Method

For $i = 1 \ldots n$, let $c_i :=$ actual cost of operation $i$

For each operation, we define a certain amount $\hat{c}_i$, called the amortized cost of the operation

When we perform operation $i$:

- “borrow” $\hat{c}_i$ units of credit,
- save these credits in the data structure
- remove $c_i$ credits from the data structure to “pay” for “current expenses”

We require that at each operation, we can find the appropriate credits in the data structure

Total cost is at most $\sum_{i=1}^{n} \hat{c}_i$
Example: Binary Counter

\[ c_i = \# \text{ of bits flipped in operation } i \quad \cdots \quad 0 \quad 1 \quad 1 \quad 1 \quad 1 \]

\[ \cdots \quad 1 \quad 0 \quad 0 \quad 0 \quad 0 \]

Define \( \hat{c}_i := 2 \)

We store one unit of credit on each “1” in the counter.

Consider a single increment operation:

- each time we flip a “1” into a “0”, we pay for this flip using the credit stored on the “1”
- in the last step, we turn a “0” into a “1”, using our two borrowed credits \( \hat{c}_i \) to do this: one to pay for the actual cost of the flip, and one to store a unit of credit on the resulting “1”
The Potential Method

Generalizes the accounting method

One defines a potential function $\Phi$ on the data structure

The amortized cost for operation $i$ is defined to be

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

By a “telescoping sum” argument, we have

$$\sum_{i=1}^{n} c_i = \sum_{i=1}^{n} \hat{c}_i + \Phi(D_0) - \Phi(D_n)$$

Assuming $\Phi(D_n) \geq \Phi(D_0)$, we have

$$\sum_{i} c_i \leq \sum_{i} \hat{c}_i$$
Example: Binary Counter

Define $\Phi(\text{counter}) := \# \text{ of 1-digits in counter}$

Consider $i$th increment

\[
\begin{align*}
\cdots & 0 \; 1 \; 1 \; 1 \; 1 \\
\cdots & 1 \; 0 \; 0 \; 0 \; 0
\end{align*}
\]

$c_i = \# \text{ of flips}$

$\Delta \Phi = 1 - (c_i - 1) = 2 - c_i$

$\hat{c}_i = c_i + \Delta \Phi = c_i + 2 - c_i = 2$

$\therefore \text{amortized cost} = 2$
Example: Generalized Binary Counter

Suppose that each operation may specify an arbitrary increment position

The original \textit{ad hoc} analysis is no longer valid

However, using either the accounting or potential method, the analysis goes through \textit{unchanged}

\[ \therefore \text{total cost is } \leq 2n \]

Another generalization: assume counter initially has \( m \)

1’s, so \( \Phi(D_0) = m \)

Then total cost is \( \leq 2n + m \)
Example: dynamic arrays

An array that grows over time

At any time the dynamic array has a capacity and a size, and holds a fixed-length array A of length capacity

- elements $A[0..size)$ contain actual data
- elements $A[size..capacity)$ are unoccupied
- initially, capacity = 1 and size = 0
Operations $append(x)$:

$$\begin{align*}
\text{if } size < capacity \text{ then} \\
\quad A[size] & \leftarrow x, \quad size ++ \\
\text{else} \\
\quad \text{allocate a new fixed-length array } B \\
\quad \quad \text{of length } capacity + \Delta \\
\quad \text{copy } A[0..size) \text{ into } B[0..size) \\
\quad \text{replace } A \text{ with } B \\
\quad capacity & \leftarrow capacity + \Delta \\
\quad A[size] & \leftarrow x, \quad size ++
\end{align*}$$

Question: How big should $\Delta$ be?

- if it is too big, we waste space
- if it is too small, we waste time
Consider a sequence of $n$ append operations starting from $n$

measure the cost as the number of data copies
(including $A[size] \leftarrow x$)

if $\Delta = 1$, every append triggers a new allocation, so the
$i$th append operation costs $\approx i$

**better idea:** when we re-allocate, *double* the capacity

we will not waste too much space ($A$ is always at
least half full)

amortized cost of doubling strategy: 3
Amortized analysis of doubling strategy

for each "append" operation:

- general strategy:
  - 1 credit pays for the copy $A[size] \leftarrow x$
  - 1 credit goes on $A[size]$
  - 1 credit goes on another element of $A$ that does not hold credit

- when we re-allocate:
  - all elements of $A$ hold a credit, which we use to pay for the copy into $B$

- key observation:
  - immediately after a re-allocation, there are
    - $size - 1$ unoccupied array elements
    - $size - 1$ occupied array elements without credit