Honors Algorithms
G22.3520-001 Fall 2006

Lecture 10
Read: CLRS 17
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 $T(n)/n$.

- there is really no probability distribution to speak of.
- 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>0</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</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>0 0 0 0 1 1</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>0 0 0 1 0 0</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>0 0 0 1 0 1</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>0 0 0 1 1 0</td>
<td>2</td>
</tr>
</tbody>
</table>

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

$\# \text{ of flips} = \# \text{ of low order 0-bits of result} + 1$

Crude analysis: $\# \text{ flips per increment} = O(\log n)$

Total cost = $O(n \log n)$
A better analysis

Total cost for $n$ increments:

\[ n + \# \text{ of even numbers among } 1 \ldots n \]
\[ + \# \text{ of multiples of 4 among } 1 \ldots n \]
\[ + \# \text{ of multiples of 8 among } 1 \ldots n \]
\[ + \cdots \]

\[ = \sum_{i=0}^{\infty} \left\lfloor \frac{n}{2^i} \right\rfloor \leq \sum_{i=0}^{\infty} \frac{n}{2^i} = n \sum_{i=0}^{\infty} \frac{1}{2^i} = 2n \]

More structured analysis:

- the accounting method
- potential functions
The Accounting Method

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

For each operation, we “borrow” a certain amount $\hat{c}_i$, called the \textit{amortized cost} of the operation.

We may have $\hat{c}_i < c_i$, but we require that

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

for each $m = 0 \ldots n$

This is done by saving “credits” with specific objects in the data structure.
When we perform operation $i$, we “borrow” $\hat{c}_i$ units of credit, and we

- add these credits to the data structure
- spend $c_i$ credits to “‘pay” for “‘current expenses”

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

$c_i =$ # of bits flipped in operation $i$  
\[ \begin{array}{c}
\cdots 0 1 1 1 1 1 \\
\cdots 1 0 0 0 0 0
\end{array} \]

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

Consider \( i \)th increment

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

\( c_i = \# \text{ of flips} \)

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

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

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

Suppose that each operation may specify an arbitrary increment position

The original *ad hoc* analysis is no longer valid

However, using either the accounting or potential method, the analysis goes through *unchanged*

∴ 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 hash tables

Suppose we double the # of slots in a hash table when its factor reaches 1

When this happens, a new hash function is selected, and all items are re-hashed into the new, larger table

Accounting method: borrow 2 units for each insertion ($\hat{c}_i = 2$)

- one unit is “stored” with the new item
- one unit is “stored” with an old item that currently has no credit stored with it

when the table is expanded, every item can “pay for itself”
Example: The move-to-front heuristic

Reference: Sleator and Tarjan [1985 CACM 28(2)]

Suppose we want to maintain a set of items using an *unsorted list*

The operations are *access*, *insert*, and *delete*:

- to access an item, we scan the list from left to right, until the item is located
- to insert an item, we scan the list from left to right, and if it is not present, we insert it at the far right
- to delete an item, we scan the list from left to right, and if present, we delete it
We may also perform a swap: exchanging two consecutive elements in the list

Costs:

- Cost of access, insert, delete is the number of items scanned in the list
- After performing an access or an insert, if the item is located in position $i$, we allow the item to be moved to the left any number of positions at no cost
  
  These are called free swaps

- Cost of any other swap is 1
  
  These are called paid swaps
The move-to-front (MF) heuristic:

- whenever an item is accessed or inserted, it is moved to the front (i.e., the left end) of the list (leaving the relative order of other items unchanged)
- these are all free swaps
- no paid swaps are performed

Intuition:

- Items that are accessed most frequently should be near the front
- MF should keep frequently accessed items near the front
We will show: On any sequence of operations, the cost of $MF$ is within a constant factor of the cost of the best strategy for that sequence.

Definitions: for an algorithm $A$ and sequence of operations $s$, define

- $C_A(s) := \text{cost of } A \text{ on } s, \text{ excluding both free and paid swaps}$
- $X_A(s) := \# \text{ of paid swaps by } A \text{ on } s$
- $F_A(s) := \# \text{ of free swaps by } A \text{ on } s$

**Theorem.** For every algorithm $A$, and any sequence $s$ of operations (starting with the empty set), we have

$$C_{MF}(s) \leq 2C_A(s) + X_A(s) - F_A(s)$$
Proof. We consider parallel execution of A and MF. Suppose that at a given moment, A’s list is \((a_1, \ldots, a_n)\).

Let \(\pi_i := \text{position of } a_i \text{ in } MF\text{’s list}\).

We call a pair \((i, j)\) an inversion if \(i < j\) and \(\pi_i > \pi_j\).

We define the potential function \(\Phi\) to be the current number of inversions.

Consider an access operation, and assume that the element accessed is \(a_i\).

Let \(c_A := A\’s \text{ cost } = i\).

Let \(c_{MF} := MF\’s \text{ cost } = \pi_i\).

MF’s amortized cost is \(\hat{c}_{MF} := c_{MF} + \Delta\Phi = \pi_i + \Delta\Phi\).
Let $x_i := \#$ of items that precede $a_i$ in MF’s list, but follow $a_i$ in A’s list

Let $y_i := \#$ of items that precede $a_i$ in both lists

$y_i = \#$ of items common to $\{a_1, \ldots, a_{i-1}\}$ and the lower shaded region

We have \[ x_i + y_i = \pi_i - 1 \]
Consider effect on $\Phi$ when we move $a_i$ to the front of $MF$’s list

- Creates $y_i$ new inversions
- Destroys $x_i$ other inversions

Therefore, $\Delta \Phi = y_i - x_i = \pi_i - 1 - 2x_i$ and

$$\hat{c}_{MF} = c_{MF} + \Delta \Phi$$

$$= \pi_i + (\pi_i - 1 - 2x_i) = 2(\pi_i - x_i) - 1$$

We also have

$$\pi_i - x_i = y_i + 1 \leq (i - 1) + 1 = i = c_A,$$

and hence

$$\hat{c}_{MF} \leq 2c_A - 1$$
We need to take into account the effect of $A$’s swaps on $\Phi$

- A paid swap by $A$ increases $\Phi$ by at most 1
- A free swap by $A$ decreases $\Phi$ by 1

Analysis of *insert* and *delete* is similar

Summary:

- For each operation, we have $\hat{C}_{MF} \leq 2C_A$
- Each paid swap increases $\Phi$ by 1 and each free swap decreases $\Phi$ by 1
- Summing it all up, we obtain the theorem:

$$C_{MF}(s) \leq 2C_A(s) + X_A(s) - F_A(s)$$

QED
Related Results

Suppose list initially contains \( n \) items, instead of being empty.

Since there are at most \( n(n - 1)/2 \) inversions, we obtain

\[
C_{MF}(s) \leq 2C_{A}(s) + X_{A}(s) - F_{A}(s) + n(n - 1)/2
\]

Idea: our original proof used the fact that \( \Phi \) was initially zero (and is never negative).

But now, \( \Phi \) is not initially zero, but is no larger than \( n(n - 1)/2 \).
As a special case, suppose there are no insertions and deletions, and items are accessed according to some (unknown) distribution.

An algorithm A that “knows” the access distribution can order the items to minimize its own cost (placing most frequently accessed item first, second most frequently accessed item second, etc.)

This analysis shows that for long enough sequences, MF will effectively “learn” the distribution.