AMORTIZED ANALYSIS

Reading: CLRS Chpt 17

When computing running time on operations of data structures, different types of operations can have different costs. Successful procedures may require selecting correct sequence of operations and then executing them.

Example

\[ \begin{array}{cccccc} 1 & 2 & 3 & 4 & \rightarrow & 1 & 2 & 3 & 4 & 5 & 11 \end{array} \]

It is convenient to assign average cost per operation, because usually can construct situations where expensive operations are sufficiently rare that the average cost per operation is quite small. "An amortized analysis guarantees the average performance of each operation in the worst case."

There are three main techniques:

1. Aggregate analysis
2. Accounting method
3. Potential method
Aggregate Analysis

- Each operation is assigned the same avg cost, independent of type. [different from other methods]
- Procedure: Get total running time for sequence of \( n \) operations and then divide by \( n \).

Ex: Stack operations

\[
\begin{align*}
& \text{Push}(S, x) & \text{− pushes object } x \text{ onto stack } S \\
& \text{Pop}(S) & \text{− pops top object from stack } S \text{ and returns object} \\
& \text{MULTIPOP}(S, k) & \text{− pops top } k \text{ objects from stack (or until empty)} \quad \text{while not STACK-EMPTY}(S) \quad \text{and } k > 0 \\
& \quad \quad \text{\quad \text{\& returns last object} } \\
\text{min}(s, k) & \left\{ \begin{array}{l}
\text{Pop}(S) \\
\text{\quad \quad } k = k - 1
\end{array} \right.
\end{align*}
\]

Imagine now we wish to analyze running time for sequence of \( n \)
Push, Pop, and MULTIPOP operations.

- Assume stack initially empty

Before this lecture, might say...

- Worst-case analysis
  → worst individual step: MULTIPOP on full stack \( \Theta(n) \)
  → the stack has at most \( n \) objects
  → worst sequence of steps would be \( n \) such steps in a row
  \[ \Rightarrow \text{Worst-case is } \Theta(n^2) \]

But a little more thought reveals this sequence is not possible...

- How would the \( n \) objects get in the stack in the first place?
- Sequence of \( \Theta(1) \) push operations would be necessary
  \[ n \text{ suggests } \Theta(n) \]
More formally, the total # of calls to \texttt{Push}, whether direct or through \texttt{MULTIPOP} on an initially empty stack can not exceed the total # of calls to \texttt{Push}, which is at most \( n \).

Thus, aggregate execution time is \( \Theta(n) \) for \( n \) operations, and average cost per operation is \( \Theta(n)/n = \Theta(1) \) and we literally assign \( \Theta(1) \) to each of the three operations in an amortized sense.

Also, although this is an average, we did not make a probabilistic argument. Rather, it is a worst case argument — no adversary could create a worse outcome.

---

**Accounting Method**

- Different costs assigned to different types of operations
- Amortized cost may be actual or greater or less, but if greater a credit is assigned to objects in the data structure that “pays” for future operations.

**Requirement:** Sum of amortized costs of operations are always greater than or equal to actual costs of those operations, for all sequences of operations

\[
\sum_{i=1}^{n} \hat{C}_i \geq \sum_{i=1}^{n} C_i
\]

- Moreover, the total credit stored in the data structure is exactly equal to the difference \( LHS - RHS \) above, and must always be non-negative to maintain inequality above, \( \Rightarrow \) like in financial accounting, credits and debits must balance.
Ex: Stack operations

<table>
<thead>
<tr>
<th>Operation</th>
<th>Actual cost, c_i</th>
<th>Amortized cost, ( \bar{c}_i )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Push</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Pop</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Multipop</td>
<td>( \min(s, k) )</td>
<td>0</td>
</tr>
</tbody>
</table>

Think of the stack as a stack of plates in a cafeteria. Use dollar-bills to represent the costs.

Every time we push a "plate" onto the stack, its amortized cost is $2 but the actual is only $1. Use $1 to pay the cost of the push, and store the second $1 on the plate on the stack.

- At all times, each plate on the stack has a $1 credit on it.
- The $1 credit on each plate serves as pre-payment for the Pop or Multipop that has yet to come but has no associated amortized cost.

---

Potential Method

- Similar in spirit to Accounting method.
- Prepaid credit is associated not with elements of data structure, but with entire data structure as a whole.
- Prepaid credit is thought of as "potential energy" or just "potential."
Consider sequence of operations indexed by 
\[ i = 1, 2, \ldots, n \]
with actual costs \[ c_1, c_2, \ldots, c_i, \ldots, c_n \]
and that transforms data structure from initial state \( D_0 \)
through sequence \( D_1, D_2, \ldots, D_i, \ldots, D_n \)

Associate potential function \( \Phi(D_i) \) mapping data structure state to potential,

\[ \hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) \quad \text{"Conservation of Energy"} \]

Sum over sequence of \( n \) operations

\[
\sum_{i=1}^{n} \hat{c}_i = \sum_{i=1}^{n} \left( c_i + \Phi(D_i) - \Phi(D_{i-1}) \right) \\
= \left( \sum_{i=1}^{n} c_i \right) + \left( \Phi(D_n) - \Phi(D_0) \right) \quad \text{telescoping}
\]

Often set = 0 for convenience

\[ \Phi(D_n) \text{ must be } \geq \Phi(D_0) \text{ to} \]

insure amortized cost is never less than actual cost.

- The increase in potential over \( \Phi(D_0) \) represents the total "credit" stored in the data structure

- In any single operation, if the amortized cost exceeds the actual cost, the potential increases. If the amortized cost is less than the actual cost, the potential decreases.
Ex: Stack Operations

Define \( D(S) = s \), the number of objects currently on the stack.

\[
D(\emptyset) = 0
\]

\( D(\emptyset) \geq 0 \) because \# of elements is non-negative

Consider the operation as \underline{push} onto stack of \( s \) objects:

\[
D(D_i) - D(D_{i-1}) = (s+1) - (s) = 1 \]

\[
C_i = C_i + D(D_i) - D(D_{i-1}) = 1 + 1 = 2 \quad \text{amortized cost}
\]

Consider the operation as \underline{multipop} \((s, 1)\), resulting in \( k' = \min(k, s) \) objects popped off stack.

\[
D(D_j) - D(D_{j-1}) = -k' \]

\[
C_j = C_j + D(D_j) - D(D_{j-1}) = k' - k' = 0 \quad \text{amortized cost}
\]

Pop \( \Rightarrow C_j = 0 \) also

Thus:

<table>
<thead>
<tr>
<th>Operation</th>
<th>( C_i )</th>
<th>( \bar{C}_i )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Push</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Pop</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Multipop</td>
<td>( \min(k, s) )</td>
<td>0</td>
</tr>
</tbody>
</table>

The amortized cost of each \( O(1) \)

and so sequence of \( n \) is \( O(n) \).
Larger Example: Dynamically Growing (and Shrinking) Table

Problem - Application creates storage for table. Elements are added (and removed). Want table to grow when out of space. (Also want table to shrink if too little is being used.)

- When table grows (or shrinks), must allocate new space and copy elements from old table to new, which can be a large but infrequent cost.

\[
\text{load factor}(T) = \frac{T\text{num} - \text{filled elements}}{T\text{size} - \text{total capacity}} \Rightarrow \text{load factor}(T) \approx 1 \text{ for } T\text{size} = 0
\]

\[\text{TABLE-INSERT}(T, k) \quad - \text{insert element } k \text{ into table } T\]

if $T\text{size} = 0$
  allocate $T\text{table}$ with 1 slot
  $T\text{size} = 1$

if $T\text{num} = T\text{size}$ - if table is full
  allocate new-table with $2 \times T\text{size}$ slots
  insert all items in $T\text{table}$ into new-table
  free $T\text{table}$

  $T\text{table} = \text{new-table}$
  $T\text{size} = 2 \times T\text{size}$

  insert $x$ into $T\text{table}$
  $T\text{num} = T\text{num} + 1$

Assume that these elementary insertions dominate the run time at cost of $a_i$ each per item inserted.
Consider the insertion event in a string of insertions

\[ C_i = \begin{cases} 1 & \text{if table has space} \\ i & \text{if table has to be expanded} \end{cases} \]

\((i-1)\) to copy the previous table

+1 to insert the new element

The "Chicken Little" analysis would say the worst case operation in a sequence of \( n \) operations is \( O(n) \), and so a sequence of \( n \) such operations would be \( O(n^2) \). This is not tight however, because not all operations can be worst case

Aggregate analysis

\[ C_i = \begin{cases} i & \text{if } i-1 \text{ is an exact power of } 2 \\ 1 & \text{otherwise} \end{cases} \]

\[ \sum_{i=1}^{n} C_i \leq n + \sum_{j=0}^{\log n} 2^j \]

\(\leq n + 2n \)

\(= 3n\)

Total cost of \( n \) operations is at most \( 3n \), so the amortized cost per execution is at most 3.

Accounting analysis

The cost of an insertion \( \$3 \) because we pay \( \$1 \) for the current insertion, and we save \( \$2 \) for the next table expansion. \( \$1 \) will be to move the current element into the new table, and \( \$1 \) will be used to move a previous element (inserted before the last doubling) to the new table. When it is time to double again, there will be just enough credit built to move all elements.
Potential analysis

Choose potential function that is zero immediately after doubling and grows to size of table to pay for next expansion:

$$
\Phi(T) = 2^i \cdot \text{num} - T \cdot \text{size}
$$

- right after expansion $T \cdot \text{num} = \frac{\text{size}}{2}$, so $\Phi = 0$
- just before expansion $T \cdot \text{num} = \text{size}$, so $\Phi = T \cdot \text{num}$
- Table is always half or more full, so $\Phi(T)$ is non-negative

→ if insertion $i$ does not expand table:

$$
\hat{C}_i = C_i + \Phi_i - \Phi_{i-1}
$$

$$
= 1 + (2 \cdot \text{num}_i - \text{size}_i) - (2 \cdot \text{num}_{i-1} - \text{size}_{i-1})
$$

$$
= 1 + (2 \cdot \text{num}_i - \text{size}_i) - (2 \cdot (\text{num}_{i-1} - 1) - \text{size}_{i-1})
$$

$$
= 3
$$

→ and if insertion $i$ does expand table:

$$
\hat{C}_i = C_i + \Phi_i - \Phi_{i-1}
$$

$$
= \text{num}_i + (2 \cdot \text{num}_i - \text{size}_i) - (2 \cdot \text{num}_{i-1} - \text{size}_{i-1})
$$

$$
= \text{num}_i + (2 \cdot \text{num}_i - 2(\text{num}_{i-1} - 1)) - (2(\text{num}_{i-1} - 1) - (\text{num}_{i-1} - 1))
$$

$$
= \text{num}_i + 2 - (\text{num}_{i-1} - 1)
$$

$$
= 3
$$