Amortized Analysis

1. Amortized Analysis

- So far in this course, we’ve seen many data structures and use worst case running time or average case running time to measure the performance of operations.
- Sometimes, the cost of an operation may vary differently, so the worst case running time is not always a good measurement.
- For the average case running time, it depends on the distribution of the input or the random choices made by the algorithm.
- In fact, more often we want to measure the cost of a sequence of operations or the cost of a single operation averaged over a sequence of operations. That leads to the idea of Amortized Analysis.
- **Definition:** If any sequence of $n$ operations on a data structure takes $\leq T(n)$ time, the amortized cost per operation is $T(n)/n$.
- Note that the Amortized analysis is not the same as Average case analysis:
  - Average cost: the expected cost of each operation.
  - Amortized cost: the average cost of each operation in the worst case. No probability involves in the Amortized cost!

2. Methods for Amortized analysis

Let’s examine different Amortized analysis methods through a simple example: Stack with MultiPop.

- We know that a Stack is a data structure that supports 2 operations:
  - Push: insert a new element on top of the Stack.
  - Pop: delete the element on top of the Stack.
- Stack can be easily implemented (using a Linked List) so that we can achieve Push and Pop in $O(1)$.
- Consider adding a new operation to Stack:
  - MultiPop$(k)$: pop the top $k$ elements off the Stack: $O(k)$ time.
• Given any arbitrary sequence of n operations (Push, Pop, Stack), what is the total amortized cost? What is the amortized cost per operation?

Let compute the cost using three different methods:

2.1. **Aggregate Method**:
- **Idea**: Follow the definition of amortized cost.
- **Methodology**:
  - Step 1: analyze the total running cost for a sequence of n operations $T(n)$.
  - Step 2: compute the amortized cost per operation: $T(n)/n$.
- Apply to our Stack example:
  - One will think this way: because each MulitPop can take at most $O(n)$, the total amortized cost for any sequence of n operations therefore is bounded by $O(n^2)$.
  - Technically it’s correct, but it’s overly pessimistic.
  - In fact, the total amortized cost is bounded by $O(n)$! Why?
  - Because each object can be Popped at most 1 for each time it’s Pushed. Thus, the number of Pops (including calls by MultiPop) can be called on a nonempty Stack is at most the number of Pushes, which is bounded by $O(n)$. Thus the total amortized cost is bounded by $O(n)$. Therefore, the amortized cost per operation is $O(n)/n = O(1)$!
  - Note: in Aggregate method, all operations have the same amortized cost (total cost divided by n).

2.2. **Accounting Method**:
- **Idea**: Assign extra cost for inexpensive operations to pay for expensive operation.
  - Each operation has real cost $c_i$ and amortized cost $\hat{c}_i$.
  - When $c_i < \hat{c}_i$: store credit in the data structure to pay for future operations.
  - When $c_i > \hat{c}_i$: use stored credit to pay for the operation.
  - The assignment is valid if: $\Sigma c_i \leq \Sigma \hat{c}_i$
• **Methodology:**
  
  o Step 1: Assign the amortized cost for each operation based on its type.
  
  o Step 2: Show that the assignment is valid.
  
  o Step 3: Show that $\Sigma \hat{c}_i \leq X$ for some $X$, which means $\Sigma c_i \leq X$.

• **Apply to the Stack example:**

<table>
<thead>
<tr>
<th></th>
<th>$c_i$</th>
<th>$\hat{c}_i$</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(k)</td>
<td>k</td>
<td>0</td>
</tr>
</tbody>
</table>

  o Think of Stack as a Bank. Whenever you Push an item to the bank, you use $1$ to pay for the Push operation and store $1$ within the Item.
  
  o When this item is popped (by a Pop or MultiPop), we use that $1$ stored with it to pay for the operation. Thus, we don’t need to pay any extra cost for Pop or Multiple. Thus, the assignment is valid.
  
  o We can easily see that: $\Sigma \hat{c}_i \leq 2n$. Thus, $\Sigma c_i \leq 2n = O(n)$.

• **Note that the key in the Accounting method is to choose a good amortized cost for each operation.**

2.3. **Potential Method**

• **Idea:** keep track of the potential energy of the data structure.
  
  o Let $D_i$ be the state of the data structure after the $i^{th}$ operation.
  
  o Define potential function: $\Phi: D_i \rightarrow R$. $\Phi(D_i)$: potential value associated with $D_i$.
  
  o Let $c_i$ be the real cost of the $i^{th}$ operation, define the amortized cost: $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$.
  
  o It’s easy to prove that $\Sigma c_i \leq \Sigma \hat{c}_i$.

• **Methodology:**
  
  o Step 1: Define a potential function.
  
  o Step 2: Define the amortized cost.
Step 3: Show that $\sum \check{c}_i \leq X$ for some $X$, which means $\sum c_i \leq X$.

- **Apply to the Stack example:**
  - Define the potential function: $\Phi(D_i) = \text{size of the Stack after the } i^{\text{th}} \text{ operation}$.
  - Push: $\check{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = 1 + 1 = 2$.
  - Pop: $\check{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = 1 + (-1) = 0$.
  - MultiPop: $\check{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = k + (-k) = 0$.
  - Therefore: $\sum \check{c}_i \leq 2n$. Thus, $\sum c_i \leq 2n = O(n)$.

- Note that in the Potential Method, the **key** is to choose a good potential function.
- The Accounting method charges each operation a fixed cost based on its type, while the Potential method focuses on the effect of a particular operation at a particular point.

### 3. Union-Find

Now, let’s consider a more complicated data structure named Union-Find. When we implement the Kruskal’s Algorithm for Minimum Spanning Tree (MST) or the Karger’s Contraction Algorithm for global min cut, we need to determine whether both end points of an edge are in the same set of vertices. Let’s say we have a collection of dynamic (can be changed) disjoint sets $S = \{S_1, S_2, ..., S_k\}$ of total $n$ elements.

We need a data structure to support the following 3 operations:

- **MakeSet(x):** create a new set that contains $x$, return the name of that set.
- **Find(x):** return the name of the set that contains $x$.
- **Union(x, y):** return the union of 2 sets: $S_1 \cup S_2$ such that $x \in S_1, y \in S_2$.

#### 3.1. Linked List implementation (Easy Find):

- Use a Linked List to represent a set. For each node, we have an extra pointer that points to the head of the List. The head contains the name and the size of the set.
- It’s easily to see that MakeSet(x) and Find(x) only take $O(1)$. That’s why this implementation is called Easy Find.
• For Union(x, y):
  o \( S_1 = \text{Find}(x), S_2 = \text{Find}(y) \).
  o To merge \( S_1 \) and \( S_2 \), first we determines which set is smaller (let’s say \( S_2 \)).
  o Update the size of \( S_1 \) to be \(|S_1| + |S_2|\).
  o Insert the first element in \( S_2 \) to the beginning of \( S_1 \). Traverse through each node of \( S_2 \), update to backward pointer to point to the head of \( S_1 \). Link the last node of \( S_2 \) to the first node of \( S_1 \).
  o The cost for a Union: \( O(\min(|S_1|, |S_2|)) \)

• Given a sequence of \( m \) operations (MakeSet, Find, Union), what is the total running cost? We will prove that it’s bounded by \( O(m + n \log n) \).
  o Observation: no matter how many times we do Union, the size of the largest set after \( m \) operations is \( n \).
  o Each time we do a Union, we walkthrough the smaller list, the size of that list is at least double. So if a node is update \( l \) times, it’s in a set of at least \( 2^l \). Because \( 2^l \leq n \Rightarrow l \leq \log n \). Thus, every element is update at most \( \log n \) times. We have at most \( n \) elements, so the total cost is bounded by \( O(n \log n) \).
  o Each time we do a Union, it takes \( O(1) \) to do Find(x), Find(y).
  o Thus, the total cost of any sequence of \( m \) operations is bounded by \( O(m + n \log n) \).

3.2. Reverse Tree implementation (Easy Union).

• Use Reverse Tree to represent a list: each node in the tree point to its parent, the root point to itself.
• It’s easily see that MakeSet(x) takes \( O(1) \), Find(x) takes \( O(\log n) \).
• For Union, we do Union(x, y) by rank (upper bound of the height of a node): the root that has lower rank will be the child of the root that has higher rank. It only takes \( O(1) \) to merge 2 trees.
• Theorem (no proof): any sequence of \( m \) operations (MakeSet, Find, Union) is bounded by \( O(m \log^* n) \), where \( \log^* n \) is the number of times you can do \( \log n \) until it’s \( \leq 0 \).