6 7

Amortized Analysis

1. Definition:

- Amortized analysis is a worst-case analysis of a sequence of operations. It is used to obtain a tighter bound on the overall or average cost operation in the sequence than what is obtained by separately analyzing each operation.

- Amortized analysis is an upper bound. It is the average performance of each operation in the worst case.

- Amortized analysis is concerned with the overall cost of a seq of operations. It does not say anything about the cost of a specific operation in that sequence.

- Amortized analysis does not involve probability. This is how amortized analysis differs from the average-case time complexity of one operation.

- The worst-case seq complexity of a seq of m operations is the max total time over all seqs of m operations. Therefore, the worst-case seq complexity is less than or equal to m times the worst-case time complexity of a single operation in any seq of m operations.

Hilroy

- The amortized seq complexity of a seq of m operations is:

$$\text{amortized complexity} = \frac{\text{worst-case seq complexity}}{m}$$

- There are 3 basic methods for computing amortized complexity:

1. Aggregate Method
2. Accounting Method / Banker's Method
3. Potential Method / Physicist's Method

2. Aggregate Method

- Computes the worst-case seq complexity of a seq of operations and divides by the number of operations in the seq.
- E.g. Suppose we have an augmented stack with the operations:

1. Push(S,x): $\Theta(1)$
2. Pop(S): $\Theta(1)$
3. MultiPop(S,k): Removes the top k elements from the stack. $\Theta(k)$

Consider performing a total of n operations from among push, pop and multipop on a stack that is initally empty. The stack can contain at most n elements. Furthermore, we know that the cost of each operation for multipop is $O(n)$. Therefore, in total, the cost is $O(n^2)$. However, we can use the aggregate method to get a much tighter upper bound. We know that we can only pop an element if we've pushed the element first. Since there can be at most n pushes, there can be at most n pops, including counting the appropriate number of pops for multipop. This implies that the total time taken for the entire sequence is at most $O(n)$. This gives us that each operation takes on average $O(1)$ time.

- Note: The aggregate method applies the cost to each operation, even when there are several types of operations in the sequence.

3. Accounting (Banker's) Method:

- In the accounting method, we do the analysis as if we were an intermediate service providing access to the data structure.
  - The cost to us for each operation is the operation's worst-case running time.
  - We charge the customer for each operation s.t. we cover our costs with what we earn in charges.
  - We aim for a total charge as close as possible to the total cost. This will give us the best estimate of the true complexity.

- The charge is the approximate amortized complexity per operation. Typically, we will charge more for some types of operations and nothing for other types. When we charge more than the cost, the the leftover amount can be stored with the elements in the data structure as credit. When we perform a free operation on an element, we can use the credit stored with that element to pay for the cost of the operation.

- I.e. In the accounting method, we assign differing charges to diff operations, with some operations charged more or less than they actually cost. We call the amount we charge an operation its amortized cost. When an operation's amortized cost exceeds its actual cost, we assign the difference to specific objects in the data structure as credit. Credit can help pay for later operations whose amortized cost is less than its actual cost. Thus, we can view the amortized cost of an operation as being split between its actual cost and the credit that is either deposited or used up. Different operations may have different amortized costs. This method differs from the aggregate analysis, in which all operations have the same amortized cost.

- We must choose the amortized costs of operations carefully. We want to assign charges and distribute credits carefully s.t. we can ensure that each operation's costs will be payed and that the total credit stored in the data structure is never negative.

Hilroy

- The total amount charged for a seq of operations is an upper bound on the total cost of the seq. This means that we can use the total charge to compute an upper bound on the amortized complexity of the seq.

- E.g.

1. Consider the augmented stack with multipop from before.

   Recall that the cost of each operation (representing the time complexity of each operation) is as follows:

   - Cost(Push(S, x)) = 1
   - Cost(Pop(S)) = 1
   - Cost(Multipop(S, k)) = min(k, |S|)

   However, each element can take part in at most two operations, one push and one pop/multipop. Therefore, the total cost for one element is 2.

   We can assign charges like this:

   - Charge(Push) = 2
   - Charge(Pop) = 0
   - Charge(Multipop) = 0

This way, every time we push an element, we will have enough credit to pop/multipop it.

The total charge for m operations is at most 2m, so the total cost is O(m). To get the amortized complexity, we divide the total cost by the number of operations. In this case, dividing O(m) by m gets us an amortized complexity of O(1) per operation.

2. Dynamic Arrays:

- Consider an array of fixed size and two operations:

1. Append: Store an element in the first free position of the array.

2. Delete: Remove the element in the last occupied position of the array.

- We can use a stack to implement an array.
- The adv of using a stack is that accessing elements is very efficient.
- The disadv is that the size is fixed.
- To get around the disadv, we can use dynamic arrays.
- When trying to append an element to an array that's full, we

1. Create a new array that is twice the size of the old one.
2. Copy all the elements from the old array into the new one.
3. Carry out the append operation.

- To calculate the amortized cost, think about the cost of performing n operations for appending elements, starting from an empty array of size 1.

Size of Array	# of Elements in Array	Cost of Adding 1	Result Size	Total Cost
1	0	1	1	1
1	1	2	2	3
2	2	3	4	6
4	3	1	4	7
4	4	5	8	12
8	5	1	8	13
8	6	1	8	14
8	7	1	8	15
8	8	9	16	24
⋮	⋮	⋮	⋮	⋮
2^n	2^n-1	1	2^n	$2^{n+1}-1$

- The amortized cost is $\frac{2^{n+1}-1}{2^n}$, which equals $2-\frac{1}{2^n}$.
- Lets visit the dynamic array with the accounting method. The cost of appending an element if we don't need to increase the array size is 1. The cost of appending if we do need to increase the array size is 1 + the current size of the array. Therefore, we should charge 3 for each append. Every time we don't have to create a new array and copy elements over, we get a credit of 2.

Note that the number of credits is never negative. Furthermore, after n appends, the amortized cost is O(n).

- One adv the accounting method has over the aggregate method is that diff operations can be assigned diff charges, representing more closely the actual amortized cost of each operation.

4. Potential (Physicist's) Method:

1. Definition:

- Suppose we can define a potential function ϕ of a data structure with the following properties:

1. $\phi(h_0) = 0$, where h_0 is the inital state of the data structure.

I.e. Our data structure starts with no potential.

2. $\phi(h_t) > 0$ for all states h_t of the data structure occurring during the course of the computation.

- This means that there is never negative potential.
- The potential function is designed to keep track of the pre-charged time or cost at any point in the computation.
- The potential function measures how much saved-up time is available to pay for expensive operations. This is analogus to the bank credit in the accounting method. The difference is that with the potential method, it depends only on the current state of the data structure and not the history of the computation that got it into that state.
- We define the amortized time $T_A(i)$ of operation i as the actual time plus the change in potential.

$$T_A(i) = c_i + \underbrace{\Phi(h_{i+1}) - \Phi(h_i)}_{\text{Change in potential}}$$

c_i is the actual cost of the operation
h_i is the state of the data structure before the operation.
h_{i+1} is the state of the data structure after the operation.

- Ideally, ϕ should be defined so that the amortized time of each operation is small. Because of this, the change in potential should be positive for low cost operations and negative for high cost operations.

- For the potential method to be valid, we need the amortized time to be an over-estimate of the actual cost.

Proof:

- The total amortized time is the sum of the individual amortized times.

- Lets sum up the amortized cost for a seq of n operations.

$$\sum_{i=0}^{n-1} TA(i) = \sum_{i=0}^{n-1} \left(C_i + \phi(h_{i+1}) - \phi(h_i)\right)$$

$$= \sum_{i=0}^{n-1} C_i + \sum_{i=0}^{n-1} \phi(h_{i+1}) - \sum_{i=0}^{n-1} \phi(h_i)$$

$$= \sum_{i=0}^{n-1} C_i + \underbrace{\sum_{i=1}^{n} \phi(h_i) - \sum_{i=0}^{n-1} \phi(h_i)}_{\text{Telescoping Sum}}$$

$$= \sum_{i=0}^{n-1} c_i + \phi(h_n) - \phi(h_0)$$

$$= \text{Total cost} + \phi(h_n)$$

2. Example:

- Consider a dynamic array. We need ϕ to depend on the current state of the array. This means that the more full the array is, the higher the potential should be and the more empty the array is, the more time there is to build potential.

- We need the number of items in the array, n, to play a role. When the array doubles, we need the current potential to decrease. In fact, when the array doubles, the potential should be 0 or close to 0.

- If we let m be the size of the array, then we know that $m = 2n$ when the array doubles. From this, we can get the function $\phi(h) = 2n - m$.

Hilroy

- Let's check if $\Phi(h)$ satisfies all the reqs.

1. An array of length 0:

   - We want $\Phi(h_0) = 0$.

   - Since the array is of length 0, n and m are 0.
     $\therefore 2n - m = 0$, as wanted.

   - Once we add an element, we have an array of size 1 and we always double the array when it's full. This means that $2n \geq m$ at all times, so $\Phi(h_t) \geq 0$.

2. What's the cost of appending an element?

   Case 1 $n < m$:

   - The actual cost is 1.
   - Furthermore, n increases by 1 and m doesn't change. So, the potential increases by 2.
   - $\therefore$ The amortized time is $1 + 2 = 3$.

Case 2 $n = m$:

- The array is doubled so the actual cost is $n+1$.

- The potential before the double is:
  $2n - m$
  $= 2n - n$
  $= n$

- The potential after the double is:
  $2(n+1) - 2(n)$
  $= 2$

- So, the amortized time is $1+2=3$.