Amortized Analysis

In the analysis of algorithm, especially in algorithms related to a data structure, one needs to bound the cost of a single operation. E.g., the insertion of an element in a length-n sorted list costs $O(n)$, but the same operation for a heap costs $O(\log n)$.

Often this cost of a single operation differs widely from operation to operation. Consider a FIFO queue that is backed with an array A of size n. The data structure also has two pointers for the start and end of the queue. While the queue is not full, adding an element to the end of the queue only costs $O(1)$ time. But when the queue is full, adding an element will have to first copy the whole content of A to a larger-sized array. Thus, adding an element costs from $O(1)$ to $O(n)$ time.

If worst case analysis is used, we'll have to sadly say that the data structure costs $O(n)$ per operation. You may risk the loss of the job.

You can also do average analysis – arguing that in usual case you will both add and remove elements from the queue, making the queue not too long on an average case. But this assumes an arbitrary distribution of the input. That's not as appealing. Amortized analysis provides a better way. It tries to analyze the **worst average cost** of an operation **over any sequence** of operations. On one hand, it is average. On the other hand, the performance guarantee does not rely on any probabilistic distribution of the input. The proved bound holds for any sequence of operation – even if the sequence is designed by an adversary.

Think the array-backed FIFO queue example again. Suppose every time we need to resize the array, we double the size. Then for any sequence that has n operations, there are at most n adding operations, the total number of copying is bounded by

$$1 + 2^1 + 2^2 + \cdots + 2^{\lfloor \log_2 n \rfloor} = O(n)$$

This cost averaged to each insertion is only $O(1)$. So we say that the *amortized cost* for insertion in the array-backed queue is $O(1)$. Note that this bound holds for any sequence of add and remove operations.

This simple strategy is used in Java's ArrayList and HashMap data structures. The capacity of the data structure is increased automatically when it is full or almost full. They all have amortized constant time for adding an element. Amortized analysis is not always good when the system needs real-time response, but makes no difference for offline computations.

Three Common Methods for Amortized Analysis

There are three common methods for amortized analysis:

1. Aggregate Method
2. Accounting Method
3. Potential Method

Rather than defining these methods right now, we examine examples directly and explain these methods with the examples.

Multipop Stack

A multipop stack S adds one more operation to a regular stack: $multipop(S, k)$. This operation removes k elements from the top of the stack, or if the stack has fewer than k elements, the stack is emptied.

Suppose we simply use a regular stack to back this new data structure, and use the following pseudo code for the multipop operation:

```
Multipop(S, k):
While k>0 and S is not empty
   pop(S).
```

A multipop operation costs in worst case $O(k)$ in the pseudocode. But if we add up the cost of a sequence of n operations, or in another word, **aggregate**, things are different.

Since each pushed element is popped at most once in the pseudo code, the aggregated pop cost is bounded by $O(n)$. The amortized cost for one operation (regardless of operation type) is therefore $O(1)$.

So the aggregate methods is simply adding the costs of a series of operations.

Incrementing a Binary Counter

A binary number x is represented by an array $A[0..k-1]$ of bits. Here k is the length of the array. The lowest-order bit is in $A[0]$.

Increment(A)

1. $i \leftarrow 0$
2. While $i < k$ and $A[i] = 1$
2.1 $A[i] \leftarrow 0$
2.2 $i \leftarrow i + 1$
3. If $i < k$ then $A[i] \leftarrow 1$

The time complexity is clearly bounded by $O(k)$ bit flips for each increment. So, if the counter is increased n times, the total time complexity is $O(nk)$. Notice that to maintain a counter with value at most n, $k = \lceil \log_2(n+1) \rceil$ bits are sufficient.

But we can show that the amortized complexity for each increment is only $O(1)$ – much better than $O(k) = O(\log_2 n)$. We will provide three different analyses for this.

Aggregate method

For all the increments that increase the counter from 0 to n, the lowest bit is flipped each time. The second lowest bit is flipped every other time. In general, the i-th bit is flipped every 2^i times. Thus, the total number of flips is bounded by $\sum_{i=0}^{k-1} \left\lfloor \frac{n}{2^i} \right\rfloor < n \sum_{i=0}^{k-1} \frac{1}{2^i} = 2n$ times.

Accounting method

Imagine the algorithm is some kind of a service company. We have to pay it $1 to flip a bit. We just need to estimate how much we need to pay the algorithm to carry out all the computation. Instead of paying the exact amount for each bit flip, we set up an "account". We voluntarily pay some additional dollars for some operations. But the extra dollars will be saved in the account so that we can pay few dollars for some future operations.

Let's examine one paying scheme: We pay $2 whenever we ask the algorithm to flip a bit to 1, and $0 whenever we ask the algorithm to flip a bit to 0.

Since each bit flip costs $1, and a bit must be flipped to 1 first before it can be flipped to 0, our account balance is always nonnegative. Thus, the total money we pay is an upper bound of the actual charges, i.e., the number of bit flips.

How much would we pay? Examine the pseudo code. Each increment only sets one bit to 1. So, we pay at most $\$2n$. Thus, n increments need at most $2n$ bit flips.

A more formal description of the proof simply replaces the "dollar" analog with the "amortized cost" for an operation. It will do three things:

1. Propose the amortized cost for each operation.
2. Prove that the total amortized cost is always no less than the total actual cost.
3. Upper bound the total amortized cost.

Potential Method

Potential method is essentially an advanced accounting method. But the "account balance" is now a function of the data structure's states. This function is called the "potential" of the data structure.

Let D_i indicate the data structure state after i increments of the binary counter. Define the potential function, $\Phi(D_i)$, as the number of bits that are set to 1.

Let c_i be the number of bits flipped at the i-th increment. Then 1 bit is flipped to 1 and $c_i - 1$ bits are flipped to 0. Therefore the potential energy is reduced by

$$\Phi(D_{i-1}) - \Phi(D_i) = c_i - 2.$$

Add up the first equation for every i will give us

$$0 \geq \Phi(D_0) - \Phi(D_n) = \sum_{i=1}^{n} c_i - 2n$$

QED.

A more standard way to do this is to define an "amortized cost"

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

Then we show that

1. $\Phi(D_0) = 0.$ (Initial energy is 0.)
2. $\Phi(D_i) \geq 0$ for every i. (Energy won't be lower than initial state.)
3. $\tilde{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = 2.$ (Amortized cost is bounded.)

And a conclusion is drawn immediately from the fact that the amortized cost is a constant.

To summarize, a potential method does the following:

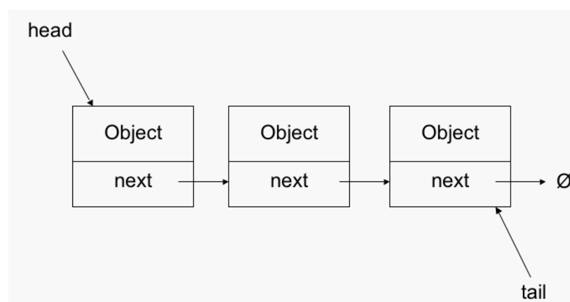
1. Defines a potential (energy) function Φ of the data structure. Φ is such that $\Phi_0 = 0$ and $\Phi_{\text{i}} \geq 0.$
2. For each operation, the "amortized cost" is the actual cost plus the change of potential (energy). Bound the amortized cost.

Remark: The three methods – aggregate, accounting and potential – are just three different ways to count (bound) the same amount. The difference is only technical.

Self-Adjusting Linked List

By using very simple data structures, we've already illustrated the three methods to estimate the amortized cost. Next let us examine a less trivial data structure.

A linked list is a common data structure.



A query requires the comparison with each object in the list sequentially. If all objects are queried with uniform frequencies, each query in a length-n linked list takes on average $\frac{n}{2}$ comparisons. But very often the objects stored in the linked list are queried with different

frequencies. For example, some English words have a much higher frequency than others. Can we adjust the order of the list so that the query is faster?

Assume we know that element i would be queried with probability $p_i > 0$. The optimal ordering for a static linked list will be such that $p_1 \geq p_2 \geq \cdots \geq p_n$. The expected number of comparisons for m queries is

$$S_{opt} = m \cdot \sum_{i=1}^{n} i \cdot p_i$$

The difficulty is that often before the queries finish, we do not know p_i. This knowledge is only computed in hindsight. After all m queries are completed,

$$p_i = \frac{\#\ of\ i\ is\ queried}{m}.$$

But since the queries have already finished one cannot go back to adjust the order any more.

Under such a situation, the idea is to adjust the linked list on-the-fly. Each time a query is performed, the order of the list is adjusted hoping that future queries will be faster. **Can we achieve anything closer to S_{opt}?**

Note that the adjustment can be based on all past queries, but does not have access to future queries. This situation is very like the online algorithm. The previous question asks us to use a dynamic online data structure to compete with a static offline data structure. So, it is not strictly the competitive analysis in online algorithm.

Possible Strategies:

- **MTF (Move to Front)**: The found element is moved to the front of the list.
- **Transpose**: The found element is swapped with the previous element.

Note that transpose is good for array backed list. But unfortunately it is disastrous in terms of competitive ratio. Keep querying the last two elements alternatively will make the competitive ratio as bad as n.

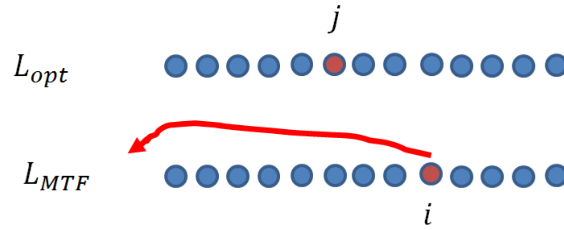
Theorem: For any access sequence, the cost of MTF satisfies $cost_{MTF} \leq 2 \cdot S_{opt} + \frac{n(n-1)}{2}$. Thus, for sufficiently long sequence, MTF is a 2-competitive algorithm against the optimal static list.

Proof: For any access sequence, suppose the optimal static list is L_{opt}.

Suppose at a moment, MTF gives a list L_{MTF}. L_{MTF} is a permutation of L_{opt}. Define the potential function $\Phi(L_{MTF})$ as the number of reversals of the permutation:

$$\Phi = \left|\{(a, b)\ |a \text{ and } b \text{ are ordered differently in } L_{opt} \text{ and } L_{MTF}\}\right|$$

Let o be the next accessed object, which is ranked at place i in L_{MTF} and j in L_{opt}. We examine the potential change caused by move o to the front of L_{MTF}.



For every object in L_{MTF} after i, the MTF does not change Φ.

For every object in L_{MTF} before i, whether they are reversed with i will become the opposite way. Among these elements, suppose k of them are not reversed before (so there are $i-k-1$ reversed). Then after MTF, there would be k elements reversed. Thus,

$$\Delta\Phi = k - (i - k - 1) = 2k - i + 1$$

Let the access cost be c. Define the amortized cost $\tilde{c} = c + \Delta\Phi$. Then

$$\tilde{c} = c + \Delta\Phi = i + (2k - i + 1) = 2k + 1 \leq 2j = 2 \times \text{cost of querying } L_{opt}.$$

Adding up all the amortized cost will give us

$$cost_{MTF} + \Phi_{\text{final}} - \Phi_{\text{init}} \leq 2 \cdot S_{OPT}$$

Since $\Phi_{\text{init}} \leq \frac{n(n-1)}{2}$, the theorem is proved.

QED.

So we have shown that MTF can compete with an offline optimal static list with a competitive ratio 2. How about the competitiveness against an offline optimal dynamic list?

If the cost function is just the number of comparisons, then we have a problem. Because the offline algorithm can always move the next accessed object to the front. So, we'll have to charge for the element movement too. If the cost is defined to be the element distance from the front plus the distance of element movements, then MTF is 4-competitive against an offline adversary. Note that this is the situation of an array backed list. We ignore the proof here.