Amortized Analysis

S. Halim YJ. Chang

School of Computing
National University of Singapore

CS3230 Lec08; Tue, 08 Oct 2024

Overview

Why do we need 'amortized analysis'?

Suppose there is a *sequence* of n operations $o_1, o_2, \ldots, o_n$.
Let $f(n)$ be the *worst-case* time complexity of *any* operation.

Let $t(i)$ be the time complexity of the i-th operation o_i, and
Let $T(n)$ be the time complexity *of all n operations*, i.e.,

$T(n) = \sum_{i=1}^{n} t(i)$

Without amortised analysis, if we only think worst-case each time, we may analyse

$T(n) = n \cdot f(n)$

This $n \cdot f(n)$ could be grossly wrong (i.e., too high/not tight).

Binary Counter

A classic motivating example for introducing amortized analysis.
Real-life application: Unix time.
PS: Yes, we will spend half-lecture talking about `++counter`...

Open `https://visualgo.net/en/bitmask` and do these steps:

- Set $S = 0$ and click 'Go'
- Click 'Increment' many times, e.g., after 11x, result = 1011

PS: The animation/pseudocode at VisuAlgo vs this lecture note are not 100% identical as we use bit manipulation at VisuAlgo.

k-bit Binary Counter

```
INCREMENT(A) // A is a bitmask (array of bits)
  i = 0       // we read A from right to left
  while i < length(A) and A[i] = 1 do // slow loop?
    A[i] = 0 // flip 1 to 0
    i = i+1
  if i < length(A) then
    A[i] = 1 // flip 0 to 1
```

Objective of amortized analysis: count the total number of bit flips ($0 \to 1$ and $1 \to 0$) during the n increments.

Let $T(n)$ be the total number of bit flips during the n increments. Our aim is to get a *tight* bound on $T(n)$.

Number of bit flips (Attempt 1)

i	$A[j]$ 3210	cost	total
0	0000	0	0
1	0001	1	1
2	0010	2	3
3	0011	1	4
4	0100	3	7
5	0101	1	8
6	0110	2	10
7	0111	1	11
8	1000	4*	15
9	1001	1	16
10	1010	2	18
n=11	1011	1	19

Attempt 1: Let $t(i)$ be the *number of bit flips of i-th op*, thus $T(n) = \sum_{i=1}^{n} t(i)$

In the worst case, $t(i) = k$ (all k bits are flipped), (e.g., from $i = 7$ to 8*), so we have $T(n) = n \cdot k \in O(n \cdot k)$

But is this a tight bound?

Number of bit flips (Attempt 2)

i	$A[j]$ 3210	cost	total
0	0000	0	0
1	0001	1	1
2	0010	2	3
3	0011	1	4
4	0100	3	7
5	0101	1	8
6	0110	2	10
7	0111	1	11
8	1000	4	15
9	1001	1	16
10	1010	2	18
n=11	1011	1	19

Attempt 2: Let $f(j)$ be the *number of times the j-th bit flips*, thus $T(n) = \sum_{j=0}^{k-1} f(j)$

Let's see the pattern:

$f(0) = n = \frac{n}{1}$ (black, $j = 0$)

$f(1) = \frac{n}{2}$ (blue, $j = 1$)

$f(2) = \frac{n}{4}$ (green, $j = 2$)

$f(3) = \frac{n}{8}$ (red, $j = 3$)

$f(j) = \frac{n}{2^j}$

$T(n) = n \cdot (\frac{1}{1} + \frac{1}{2} + \frac{1}{4} + \ldots + \frac{1}{2^{k-1}})$

$T(n) < n \cdot (\frac{1}{1} + \frac{1}{2} + \frac{1}{4} + \ldots)$

$T(n) < 2 \cdot n$ ($\sum$ of ∞ geo series)

This is much better than $O(n \cdot k)$ as k can be $\approx \log n$

Amortized Analysis

Amortized analysis is a strategy for analyzing a sequence of operations to show that the *average* cost per operation is small, although a *few* operations within the sequence *might be expensive*.

Note that there is *no* probability involved
Do not get confused with the average-case analysis.

An amortized analysis *guarantees* the *average performance* of each operation *in the worst-case*.

In our binary counter example, the average cost per increment is

$\frac{T(n)}{n} < \frac{2 \cdot n}{n} < 2$, and is thus $\in O(1)$.

We say the amortized cost of each increment is $\in O(1)$.

Question 1 at VisuAlgo Online Quiz

When we say that an operation is amortized $\Theta(1)$, we mean that:

A. n operations run in total $\Theta(n)$ time in the worst-case

B. n operations run in total $\Theta(n)$ time in the expected case

C. n operations run in total $\Theta(n)$ time in the best-case

D. n operations run in total $\Theta(1)$ time in the expected case

Types of Amortized Analyses

There are three common amortization arguments:

- *Aggregate* method (we have just seen this earlier)
- *Accounting* (Banker's) method (coming up next)
- *Potential* method (also coming soon)

The aggregate method, though simple, lacks the 'precision' on the other two methods. In particular, the accounting and potential methods allow a specific *amortized cost* to be allocated to each operation (and thus, more flexible).

Accounting Method

Instead of charging the true cost, we charge the i-th operation a fictitious *amortized cost* $c(i)$, assuming \$1 pays for 1 unit of work (time), e.g., 1 bit-flip in the binary counter example.

This fee is consumed to perform the operation
and any amount that is not immediately consumed
is stored in the *bank* to be used by subsequent operations.

The idea: *impose an extra charge on inexpensive (but frequent) operations* and *use it to pay for expensive (but rare) operations* later on. At all times, the bank balance *must not go negative*, i.e., we must ensure that $\sum_{i=1}^{n} t(i) \leq \sum_{i=1}^{n} c(i), \forall n$.

Thus, the total amortized costs provides an upper bound on the total true costs.

Accounting Method for Binary Counter (1)

We now show how to use accounting method for Binary Counter. First, we identify what are the expensive (but rare) operations?

```
1011111 // 95 in Decimal
 vvvvvv
1100000 // 96 in Decimal, 6 bits are flipped
```

Observe that 5 bits are reset ($1 \rightarrow 0$), and only 1 bit is set ($0 \rightarrow 1$).

Can we do the expensive reset ($1 \rightarrow 0$) 'free of charge (foc)'?

Accounting Method for Binary Counter (2)

We charge \$2 for each set ($0 \rightarrow 1$).
\$1 to pay for the actual bit setting (so, this is the minimum).
\$1 is stored in the bank (so, this is the 'extra').

Observation: At any point, every 1 bit in the counter has contributed \$1 to the overall savings in the bank.

We will later to use those savings to pay for resetting ($1 \rightarrow 0$) so that we can treat this expensive reset as 'foc'. Example:

```
1*0 1*1*1*1*0  // each $1$ bit has $1 saving (*)
               v
1*0 1*1*1*1*1* // INCREMENT, amortized cost = $2
  v | | | | |  // 1 to 0 resets are all 'foc'
1*1*0 0 0 0 0  // INCREMENT, amortized cost = $2 too
```

Accounting Method for Binary Counter (3)

Invariant that we need to keep: Bank balance *never drops below 0*. If this is true, then the sum of the amortized costs provides an upper bound on the sum of the true costs, and we are done.

Recall this observation: At any point, every 1 bit in the counter has contributed \$1 to the overall savings in the bank.

Claim: After i increments, the amount of money in the bank is also the number of 1s in the binary representation of i. Proof:

- Every time we set a bit $0 \rightarrow 1$, we pay \$2.
- \$1 is used to flip the bit, while \$1 is stored in the bank.
- Every time we reset a bit from $1 \rightarrow 0$, we use \$1 from bank.
- Hence, the amount of money in the bank is always equal to the number of 1s in the binary representation of i.

Accounting Method for Binary Counter (4)

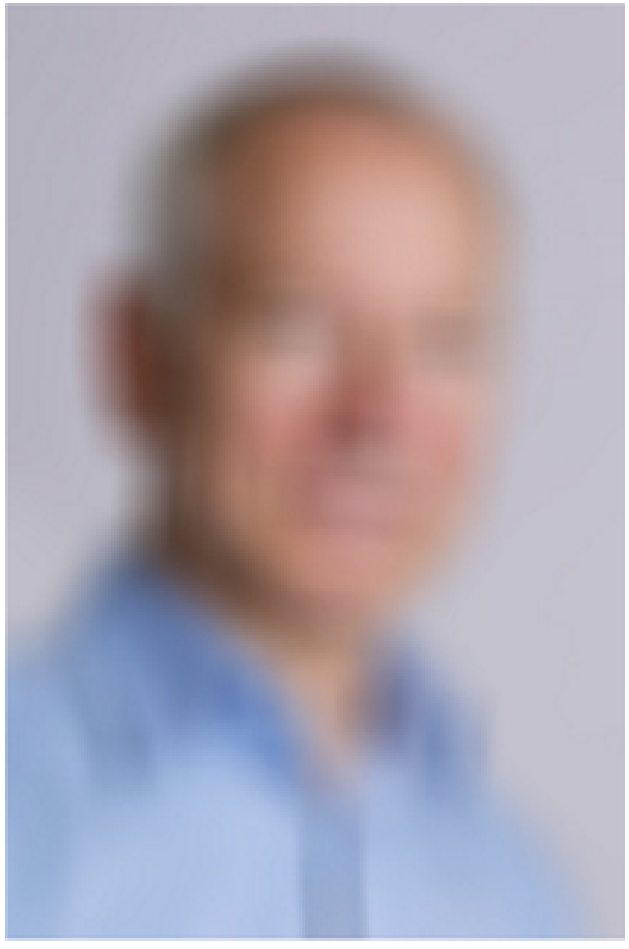
Since the number of 1s in the binary representation of i is obviously non-negative at all times, the previous slide shows that the bank balance is always non-negative too.

Therefore the conclusions are:

- The amortized cost for *each increment* $= \$2 \in O(1)$.
- The actual cost for *each increment* $\in O(1)$, because actual cost $\leq$ amortized cost.

Question 2 at VisuAlgo Online Quiz

Who is the master of algorithms pictured below?



A). David Sleator

B). Ron Rivest

C). John Hopcroft

D). Robert Tarjan

Potential Method (1)

The third method that we will learn today is the potential method.
Let:
ϕ be the potential function associated with the DS/algo
$\phi(i)$ be the potential at the end of the i-th operation

This $\phi(i)$ must fulfil two conditions:
$\phi(0) = 0$ (at the beginning, the potential is 0), and
$\phi(i) \geq 0, \forall i$ (for all operations, the potential is non-negative).

Potential Method (2)

The amortized cost of the i-th operation is defined as
$=$ actual cost of the i-th operation plus $(\phi(i) - \phi(i-1))$.
This $(\phi(i) - \phi(i-1))$ is called the potential difference $\Delta\phi(i)$.

The amortized cost of n operations is defined as
$= \sum_i$ amortized cost of the i-th operation
$=$ actual cost of n operations $+ (\phi(n) - \phi(0))$ // via telescoping
$=$ actual cost of n operations $+ \phi(n)$ // as $\phi(0) = 0$
$\geq$ actual cost of n operations

To show that the *actual cost* of n operations is $\in O(g(n))$,
just show that the *amortized cost* of n operations is $\in O(g(n))$.

Potential Method - Recipe

The hardest part to use potential method is to find a suitable potential function ϕ. The heuristic is as follows: Try to select a suitable ϕ, so that for an expensive i-th operation, $\Delta\phi(i)$ is *negative* to such an extent that it *nullifies* or *reduces* the effect of actual (expensive) cost.

Thus, try to observe the expensive operation and see if there is some quantity that is *decreasing* during that expensive operation.

Potential Method for Binary Counter (1)

There is only one operation: increment,
and sometimes it is costly (flips many bits).
(although usually it does not (flips only a few bits)).
Is there anything that is decreasing?
Hint: Observe the 1s...

```
1011111
 v|||||
1100000
```

Answer: The number of 1s decrease.
So, $\phi(i)$ is the number of 1s in the counter after the i-th increment.

Potential Method for Binary Counter (2)

The actual cost of the i-th increment $= l_i + 1$.
where l_i is the length of the longest suffix with all 1s ($1 \to 0$).
the $+1$ is because we set just one $0 \to 1$ in this case.

The $\Delta\phi(i) = -l_i + 1$,
because $\phi(i) = x - l_i + 1$ (l_i bits $1 \to 0$ and 1 bit $0 \to 1$),
and $\phi(i-1) = x$,
where x is the number of 1s.

Therefore:

Actual cost	$\Delta\phi(i)$	Amortized cost
$l_i + 1$	$-l_i + 1$	$(l_i + 1) + (-l_i + 1) = 2 \in O(1)$

Thus, the amortized cost of *each increment* $= 2 \in O(1)$,
so we have shown that the actual cost of *each increment* $\in O(1)$.

Dynamic Table

An important amortized analysis with real-life application.
PS: Yes, another half-lecture talking about `list.append(x)`...

Open `https://visualgo.net/en/array?mode=array` and do:

- Click 'Create(M,N)', set $M = 1$ and $N = 0$, and then click 'Random'.
- Click 'Insert', and then 'Append' any value v, e.g., 7, **repeatedly**.

Whenever $M = N$ (the table/array is full), the next call of append operation will trigger the table-doubling process (make $M = 2 \cdot N$), copying the existing content (N elements), before we append v at index N; otherwise when $M > N$, the append operation is very fast.

How large should a table/array be?

Goal: Make the table as small as possible, but large enough so that it will not overflow (otherwise there are too many wasted spaces).

Problem: What if we don't know[1] the proper size in advance?

Solution: Dynamic tables.
Idea: Whenever the table overflows, we 'grow'* it by allocating (in C++, we use `malloc` or `new`) a new, larger table.
We move all items from the old table into the new one, and then we free the storage for the old table.
* how to grow is subject of discussion soon...

Dynamic tables are implemented as C++ std::vector, Python list, or Java ArrayList auto-resizing capabilities.

[1] In Competitive Programming, we usually know this upper bound.

Some Notations

Let:
N: the number of elements in the table.
$createTable(M)$: a system-call that creates a table of size M and returns its pointer (assumed $O(1)$.
$size(T)$: the size of table T (assumed $O(1)$).
$copy(T, T')$: copies the contents of table T into table T' (assumed $O(size(T))$).
$free(T)$: free the space (return the space to Operating System) that was previously occupied by table T (assumed $O(1)$).

A *trivial* way to perform `Insert(v)`

```
if (n = 0)
  T = createTable(1)
else
  if (n = size(T)) // full
    T' = createTable(n+1) // is this good?
    copy(T,T')
    free(T)
    T = T'
insert v to the back of T
n = n+1
```

PS: There is no equivalent VisuAlgo /array animation, but it should be easy to see that the time complexity of n calls of `Insert(v)` is $O(n^2)$

A *better* way to perform `Insert(v)`

```
if (n = 0)
  T = createTable(1)
else
  if (n = size(T)) // full
    T' = createTable(2*n) // is this NOW good?
    copy(T,T')
    free(T)
    T = T'
insert v to the back of T
n = n+1
```

Now switch to `https://visualgo.net/en/array?mode=array`, and witness the live animation.

Worst-case Analysis

For the double-when-full strategy,
consider a sequence of n insertions.

If the worst-case time to execute *one* insertion is $O(n)$,
is the worst-case time for n insertions is $n \cdot O(n) \in O(n^2)$?
Again, is this tight?

Amortized Analysis

Observe, once the table is full,
we create a table of *double the size*.

It will take $O(1)$ time for the next *many* insertions
that fill in the empty slots, until the table is full again.

So, the heavy operation (of copying the table T into new table T'),
will only occur only when $n - 1$ is a power of 2.

Aggregate Method for Dynamic Table

i	$size_i$	$t(i)^1$	$t(i)^2$
1	1	-	1
2	2	1	1
3	4	2	1
4	4	-	1
5	8	4	1
6	8	-	1
7	8	-	1
8	8	-	1
9	16	8	1
10	16	-	1

Let $t(i)$ be the cost of the i-th insertion; it can be split into two:

$t(i)^1 = i$ if $i - 1$ is an exact power of 2 (cost of doubling and copying due to full table)

$t(i)^2 = 1$ for all i (cost of insertion)

Aggregate Method, Summary

Cost of n insertions $T(n)$ is thus:

$$T(n) = \sum_{i=1}^{n}(t(i)^1 + t(i)^2)$$

$$T(n) = \sum_{i=1}^{n} t(i)^1 + \sum_{i=1}^{n} t(i)^2$$
$$T(n) = \sum_{j=0}^{\log(n-1)} 2^j + \sum_{i=1}^{n} 1$$
$$T(n) = 2^{\log(n-1)+1} - 1 + n$$
$$T(n) \leq 2 \cdot (n-1) + n$$
$$T(n) \leq 3 \cdot n$$

Thus, the average cost of each insertion in dynamic table is:
$\frac{T(n)}{n} = \frac{3 \cdot n}{n} = 3 \in O(1)$.

Accounting Method for Dynamic Table

Is reserved for Tut08 Q2, try it

Potential Method for Dynamic Table

Before *Insert*(v)	`123j`, $j = i - 1$ `????`
After *Insert*(v)	`1234i67k`, $k = 2(i - 1)$ `????v...`

Insert(v)	Actual cost	$\Delta\phi(i)$	Amortized cost
Case 1: Full	i	?	?

When the table is full, the actual cost of the i-th *Insert*(v) is $(i-1)$ to *copy*(T, T'), plus 1, so $(i-1)+1 = i$ operations.

What should be the suitable potential function ϕ?
Is there anything that *has decreased* during this Case 1?
Seems like everything including the *size*(T) has increased...
What about $-$*size*(T)?
Can we set $\phi = -$*size*(T)?

Question 3 at VisuAlgo Online Quiz

Does the function $\phi = -size(T)$ satisfies all the properties of a potential function ϕ?

A. Yes

B. No

Potential Method for Dynamic Table (Case 1)

Before *Insert*(v)	`123j`, $j = i - 1$ `????`
After *Insert*(v)	`1234i67k`, $k = 2(i - 1)$ `????v...`

Now that $\phi = 2i - size(T)$, we have:

Insert(v)	Actual cost	$\Delta\phi(i)$	Amortized cost
Case 1: Full	i	$3 - i$	$i + (3 - i) = 3 \in O(1)$

Q: How to derive $\Delta\phi(i) = 3 - i$ for this Case 1?

Potential Method for Dynamic Table (Case 2)

Before *Insert*(v)	`12j`, $j = i - 1$ `???.`
After *Insert*(v)	`123i` `???v`

Now that $\phi = 2i - size(T)$, we have:

Insert(v)	Actual cost	$\Delta\phi(i)$	Amortized cost
Case 1: Full	i	$3 - i$	$i + (3 - i) = 3 \in O(1)$
Case 2: Not full	1	2	$1 + 2 = 3 \in O(1)$

Q: How to derive $\Delta\phi(i) = 2$ for this Case 2?

Potential Method for Dynamic Table (Conclusion)

Now that $\phi = 2i - size(T)$, we have:

$Insert(v)$	Actual cost	$\Delta\phi(i)$	Amortized cost
Case 1: Full	i	$3 - i$	$i + (3 - i) = 3 \in O(1)$
Case 2: Not full	1	2	$1 + 2 = 3 \in O(1)$

We have shown: the amortized cost of each insertion $= 3 \in O(1)$.
Therefore, the actual cost of each insertion $\in O(1)$.

Conclusions

Amortized costs can provide a clean abstraction of data-structure performance.

Amortized analysis can be performed using either of the three methods discussed in this lecture: aggregate, accounting, or potential method. However, each method has some situations where it is arguable the simplest or the most precise.

Different schemes may work for assigning amortized costs $c(i)$ in the accounting method, or choosing potential function ϕ in the potential method, sometimes yielding radically different bounds.

Acknowledgement

The slides are modified from previous editions of this course and similar course elsewhere

List of credits: Erik D. Demaine, Charles E. Leiserson, Surender Baswana, Ken Sung, Arnab Battacharya, Diptarka Chakraborty, Sanjay Jain.