Amortized Complexity

Different kinds of complexities

Average Case complexity

= average complexity over all *possible inputs/situations*

(we need to know the likelihood of each of the input!)

Worst Case complexity

= the worst complexity over all *possible inputs/situations*

Best Case complexity

= the best complexity over all *possible inputs/situations*

Amortized complexity

= average time taken over a sequence of *consecutive* operations

We will often see algorithms where the worst case complexity is much more than the average case complexity. It depends on the usage, if you're processing a lot of data, occasional bad performance might be OK. However, if we are serving a customer and (in the worst case) she has to wait a long time, that's not good for the company's reputation.

In average-, worst- or best-case complexities, we are concerned with the performance of **one** (independent) operation. Amortized complexity is different: Here we consider the average complexity among a number of **successive operations**.

The idea is that, sometimes, you deliberately put extra effort in some operations, in order to speed up subsequent operations. For example, you might spend some time cleaning up or reorganizing your data structure in order to improve the speed of the coming operations.

Whenever you are reorganizing your data structure, it might slow you down now but you'll benefit from this later (and hopefully many times).

Example: Linear search

What is the time complexity of linear search, where the searched value is stored in `x`? Assume that the length of the array is n.

- **Worst Case:** `x` is at the end of the array

  $\Longrightarrow$ we need to traverse n elements
- **Best Case:** `x` is at the beginning of the array

  $\Longrightarrow$ we only compare with the first one
- **Average Case (assuming that `x` is in the array)** – we consider two scenarios:

  (1) The likelihood of `x` being on any position is uniform. In other words, the chance that `x` is on position 0 is the same as for 1 or any other position.

  Then, the average number of traversed elements is equal to

$$\frac{1+2+3+...+n}{n} = \frac{\frac{n(n+1)}{2}}{n} = \frac{n+1}{2}$$

Example: Linear search

- **Average Case (assuming that `x` is in the array)**
  (2) The likelihood of `x` being on any position is **NOT** uniform. In this case the average number of traversed elements is computed as

$$1P(1) + 2P(2) + 3P(3) + \cdots + nP(n)$$

  where $P(i)$ denotes the likelihood (or probability) that `x` is stored on the `i` th position. This means that $0 \leq P(i) \leq 1$ and the sum of all likelihoods is equal to 1, i.e.

$$P(1) + P(2) + \ldots + P(n) = 1.$$

  (In the uniform case $P(i) = \frac{1}{n}$ for every position.)

Note: The sum $1P(1) + 2P(2) + \ldots + nP(n)$ can be also written as

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

We assume that `x` is in the array and it is there exactly once! Otherwise, we would have to compute the average complexity slightly differently.

Recall that we have a formula for *triangular numbers*:

$$1 + 2 + \ldots + n = \frac{n(n+1)}{2}$$

The *arithmetic progression* $1 + 2 + \ldots + n$ can be also written as $\sum_{i=1}^{n} i$.

Another example of a progression for which we have an exact formula is the following *geometric progression*:

$$1 + a + a^2 + a^3 + \cdots + a^{k-1} = \frac{a^k - 1}{a - 1}$$

This is easily shown by letting $S = 1 + a + a^2 + \cdots + a^{k-1}$

Then $(a-1)S = a^k - 1 \Rightarrow S = \frac{a^k-1}{a-1}$

Amortized complexity: Dynamic array (first attempt)

Naive approach:

1. initially allocate an array of 1000 entries
2. whenever the array becomes full, increase its size by 100

To insert n entries, starting from empty, how long does it take?

For simplicity assume that $n = 1000 + 100k$ (for some k).

$$
\begin{array}{r}
1000 \text{ insertions} + \\
1000 \text{ copies} + 100 \text{ insertions} + \\
1100 \text{ copies} + 100 \text{ insertions} + \\
1200 \text{ copies} + 100 \text{ insertions} + \\
1300 \text{ copies} + 100 \text{ insertions} + \\
\ldots
\end{array}
$$

In total:

- insertions: $1000 + 100k$
- copies:

  $1000k$

  $+ 100 \times (1 + 2 + \ldots + (k-1))$

  $= 1000k + 50k(k-1)$

At the beginning we have

```
MAXSIZE = 1000;
arr = new int[MAXSIZE];
stored = 0;
```

We add elements to it by storing them at the end and increasing `stored` by one. Then, anytime `stored == MAXSIZE` (i.e. `arr` becomes full), we have to allocate a new array of size `MAXSIZE = MAXSIZE + 100` and copy all elements from `arr` into it.

Recall that $1 + 2 + \ldots + n = \frac{n(n+1)}{2}$ therefore

$$1 + 2 + \ldots + (k-1) = \frac{(k-1)k}{2}$$

The following analysis, however, does not depend on the exact choice of the parameters. If we started with an array of length 10 and increased its size by 5 every time it becomes full, for example, the resulting amortized complexity would still be the same.

Copies : $1000k + 50k(k-1)$

Insertions: $1000 + 100k$

Copies and insertions together: $1000(k+1) + 100k + 50k(k-1)$

Amortized cost of one insertion:

$$\frac{1000(k+1) + 100k + 50k(k-1)}{1000 + 100k}$$

The numerator is in $\theta(k^2)$ and denominator is in $\theta(k)$

$\Longrightarrow$ the whole fraction is in $\theta(k)$.

But $O(n) = O(k)$ because $n = 1000 + 100k$

$\Longrightarrow$ the amortized complexity of insertion is $O(n)$.

The amortized cost is computed as the average number of operations needed for one insertion. In our case:

$$\frac{\text{number of copying and inserting}}{\text{number of inserting}}$$

We see that copying is the problem. In the following smarter approach we try to suggest a different strategy which makes sure that copying happens less often.

Amortized complexity: Dynamic array (= Java's ArrayList)

Smart approach:

1. initially allocate an array of 1000 entries
2. whenever the array becomes full, double its size

To insert n entries, starting from empty, how long does it take?
For simplicity assume that $n = 1000 \times 2^k$ (for some k).

$$
\begin{array}{r}
1000 \text{ insertions} + \\
1000 \text{ copies} + 1000 \text{ insertions} + \\
2000 \text{ copies} + 2000 \text{ insertions} + \\
4000 \text{ copies} + 4000 \text{ insertions} + \\
8000 \text{ copies} + 8000 \text{ insertions} + \\
\ldots
\end{array}
$$

In total:

- insertions:
  1000
  $+ 1000 \times (1 + 2 + 4 + \ldots + 2^{k-1})$
- copies:
  $1000 \times (1 + 2 + 4 + \ldots + 2^{k-1})$

Copying and inserting together:

$$1000 + 2{\times}1000{\times}(1 + 2 + 4 + \ldots + 2^{k-1})$$

Because $1 + 2 + 4 + \ldots + 2^{k-1} = 2^k - 1$, this is equal to

$$1000 + 2{\times}1000{\times}(2^k - 1)$$

Amortized cost of one insertion:

$$\frac{2{\times}1000{\times}2^k - 1000}{n}$$

Because $n = 1000 \times 2^k$, the numerator is in $\Theta(n)$ and denominator is in $\Theta(n)$

$\Longrightarrow$ the whole fraction is in $\Theta(1)$

$\Longrightarrow$ amortized complexity of insertion is $\Theta(1)$

Comparison

Inserting at the end of an array

	Average Case	Best Case	Worst Case	Amortized
Naive alg.	—	$O(1)$	$O(n)$	$O(n)$
Smart alg.	—	$O(1)$	$O(n)$	$O(1)$

(Average Case complexity doesn't make sense to consider here.)

Search in a sorted array

	Average Case	Best Case	Worst Case	Amortized
Linear srch	$O(n)$	$O(1)$	$O(n)$	$O(n)$
Binary srch	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$

(Amortized complexity is the same as Average Case complexity because the previous searches don't have any effect on the next one.)

We see that the best case and worst case complexities of the Naive algorithm and the Smart algorithm are the same. The only difference is the amortized complexity. The reason why the amortized complexity of the naive algorithm is worse is because the worse case happens too often.