# Algorithms and their Performance

## Amortized Analysis, Randomized Algorithms and Summary

**Joshua Knowles**

School of Computer Science

The University of Manchester

COMP26120 Semester 1, Week 10 LT 1.1, November 29th 2013

## Question: Who Is the Youngest?

Who is the youngest person in the room?

Let’s write an algorithm for doing it:

```
youngest := me
**while** (not asked person) **do**
    person.age := response(what is your birthdate?)
    **if** person.age $<$ youngest.age
        youngest.age := person.age
        youngest := person
output: youngest
```

What is the complexity of the algorithm?

## Question: Who Is the Youngest?

Complexity was $O(n)$ as we need to ask everyone...

## Question: Who Is the Youngest?

BUT, let’s say I run the algorithm and ***store the answer***.

THEN if someone asks me who is the youngest person in the room, and no one else has entered the room, and no one has been born in the room, and no one has died !, then how long will it take me to answer?

$O(1)$. I just look up the stored answer.

## Question: Who Is the Youngest?

Now it could be that some University officer, or the UKBA, or some other bureaucratic entity keeps asking this question, over and over and over again, and in fact they ask me it $n$ times or more. Then I could say

Well that is very annoying !

But I could also say, well

***On average***, it has only cost me $O(n)$ time in total to answer this question, and $O(1)$ per answer, ***including*** the initial calculation I did, where I asked everyone's birthdate:

$$O(n) + O(1) + O(1) + ... = O(n) \tag{1}$$
$$O(n)/n = O(1) \tag{2}$$

The fact that they asked me $n$ times the same question means that the cost of answering it “on average” went down and it was in fact very cheap.

***Amortized Analysis*** is about just such cases as this: a sequence of operations is considered, and the **worst case** ***average*** time for them is calculated.

(The slight difference is that we are normally considering some repeated operations on a *data structure*.)

# Lecture Outline

- Part 1: Amortization
- More realistic example: a stack
- How to calculate amortized complexity formally:
  - method 1: banker's method
  - method 2: physicist's method
- Part 2: Randomized algorithms and expected runtime
- Part 3: Complexity topics (revision)

## Amortized Analysis — Brief Definition

> In an amortized analysis, we average the time required to perform a sequence of data-structure operations over all the operations performed. With amortized analysis, we can show that the average cost of an operation is small, ... even though a single operation within the sequence might be expensive.

(Introduction to Algorithms, 3rd edition, Cormen et al)

## Amortized Analysis — Brief Definition

> Amortization (finance): the paying off of a debt in equal installments composed of gradually changing amounts of principal and interest.

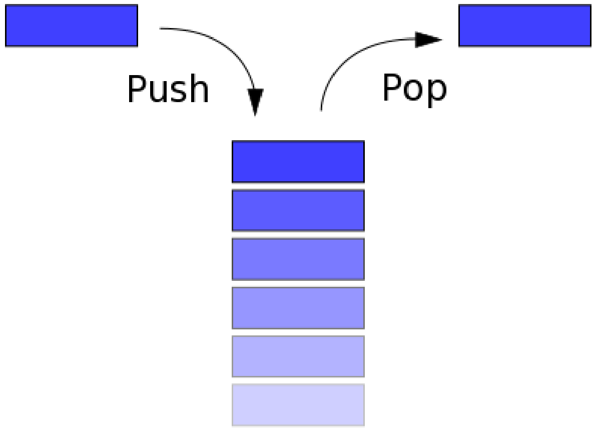
(thefreedictionary.com)

# A Stack

A stack is an abstract data type (ADT).

It gets used in lots of things, for example one is used whenever a function is called with arguments. All the arguments get put on the stack.

Defined by
push($i$) : add a data item $i$ to the stack
pop($n$) : remove $n$ items from the stack



## A Stack

Q. What is the time complexity of the pop($n$) operation? (easy, right?)

A. Assuming the stack is implemented as a linked list, and accessing an address is taken to be $O(1)$ time, then to pop $n$ items will take $n \times O(1)$ time, which is $O(n)$.

Notice: we would normally conclude that $O(n^2)$ time would be needed (worst case) for a series of $n$ operations. (But this is not necessarily the case ...)

## A Stack: Amortized Analysis

Q. What is the ***amortized*** time complexity of the pop operation, for a sequence of $n$ operations (push and pop), starting from an initially empty stack?

**What does the question mean?**

Amortized analysis asks: what is the average time ***per operation*** of a specified group (or sequence) of operations, in the worst case?

In this example, the group of operations is any allowed sequence of $n$ pushes and/or pops starting from the empty stack. We are to work out the average cost of the pop operations.

To answer, we can work out the total time and then divide through by $n$ to give the average.

## *Amortized Analysis basic formula:*

$$\text{Amortized complexity} = \frac{\text{Total time for series of operations}}{\text{number of operations}}$$

( *time* here means worst-case number of primitive operations, as usual)

## A Stack: Amortized Analysis

Q. What is the ***amortized*** time complexity of the pop operation, for a sequence of $n$ operations (push and pop), starting from an initially empty stack?

**Informal Answer**

We cannot pop more items off the stack than are on it. So, we could imagine the most expensive single (pop) operation we could do would be if we push $n-1$ times, and then in the last step we pop off $n-1$ items.

That would be $2(n-1)$ memory accesses = $O(n)$ for $n$ operations. Thus the average is $O(1)$.

Mmmm.... but could there be a sequence that is more expensive than this?

## Stack: Banker's Method

Q. What is the ***amortized*** time complexity of the pop operation, for a sequence of $n$ operations (push and pop), starting from an initially empty stack? **Use the banker's method for your analysis**

The Banker's method (also known as the Accounting method) is just a technique for accounting that helps give confidence that the worst case scenario has been properly calculated.

In the method, we attach a cost to primitive operations (say 1 dollar). Then, you charge other (high-level) operations a dollar amount too, in order to cover the cost of the lower-level operations. The trick is to charge the high-level operations enough (but not too much) so that altogether all the primitive operations are definitely covered.

That way, the total (and hence average) cost of any series of operations can be calculated.

## Stack: Banker's Method

Q. What is the ***amortized*** time complexity of the pop operation, for a series of $n$ operations (push and pop), starting from an initially empty stack? **Use the banker's method for your analysis**

A. We assume that to pop $k$ items there must be at least $k$ items on the stack

Further assume that each memory access costs 1 dollar.

Now, if we charge each push operation 2 dollars, then 1 dollar is left with the item on the stack (the other dollar is used to pay for the memory access of pushing it). Now, when a pop happens we know that the dollar left with the item can be used to pay for the memory access of the item by the pop operation.

Hence we can see that since any sequence of push and pop operations (starting from an empty stack) is fully paid for by charging 2 dollars per push, a constant number. Hence, the pop operation has amortized complexity of $O(1)$.

## Stack: Physicist’s Method

Q. What is the ***amortized*** time complexity of the pop operation, for a series of $n$ operations (push and pop), starting from an initially empty stack? **Use the physicist’s method (potential method) in your answer**.

The physicist’s method, or potential method, is another way of accounting for the cost of sequences of operations.

It measures how much time an operation uses, and also measures how much “potential” time it could also incur as a result of changes to the data structure it acted on.

## Stack: Physicist's Method

Q. What is the ***amortized*** time complexity of the pop operation, for a series of $n$ operations (push and pop), starting from an initially empty stack? **Use the physicist's method (potential method) in your answer**.

Answer. We use the basic equation:

$$t'_i = t_i + \Phi_i - \Phi_{i-1}$$

where $t'_i$ is the amortized time for the $i$th operation, $t_i$ is its actual time, and $\Phi_i$ represents the "potential" stored in total in the data structure at operation $i$.

We have:

$$\begin{aligned}
T &= \sum_{i=1}^{n} t_i && (3)\\
&= \sum_{i=1}^{n} (t'_i + \Phi_{i-1} - \Phi_i) && (4)\\
&= \sum_{i=1}^{n} t'_i + \sum_{i=1}^{n} (\Phi_{i-1} - \Phi_i) && (5)\\
&= T' + \sum_{i=1}^{n} (\Phi_{i-1} - \Phi_i) && (6)\\
&= T' + \Phi_0 - \Phi_n && (7)
\end{aligned}$$

where $T$ is the actual time taken for $n$ operations, and $T'$ is the amortized time.

As long as the potential stored in the stack is non-negative ($\Phi_0 - \Phi_n \leq 0$), then $T \leq T'$ and the actual time is less than or equal to the amortized time.

Thus, if we say that a push operation has amortized time $t' = 2$ and stores one unit of potential in the stack, and a pop operation takes $j$ units of time, but removes $j$ units of potential from the stack, we can see that any sequence of $n$ operations takes $O(n)$ time (worst case), and hence pop also has complexity $O(1)$.

## Homework Question

What is the amortized analysis for an *extendable array*?

An ***extendable array*** is one where it has an initial size. If more data is stored in it than its original size, a new array of double the initial size is allocated from memory and all the elements in the current array are copied across.

Try it, using the banker's method and physicist's method.

Then see the analysis on p39–41 of your textbook.

## Amortized Analysis is NOT Average-Case Analysis

As we have seen, amortized analysis is a worst-case analysis for a series of operations (usually involving a data-structure).

In contrast, average-case analysis works out the average complexity, assuming something about the distribution of inputs. It is more complicated and requires more assumptions.

***Amortized Analysis Example:***
**Operations on a Stack**. We can see that in the worst case $n$ operations can take only $O(n)$ time in total, even though a single operation can also be $O(n)$.

***Average-Case Analysis Example:***

**Mobile-phone Predictive text**. What is the relative efficiency of predictive text compared to standard text input (number of keys pressed)?

To answer this question, we need to assume some distribution over the words being typed. For example, a standard word frequency:

| word | rel. frequency |
|---|---|
| you | 0.1222421 |
| I | 0.1052546 |
| to | 0.0823661 |
| the | 0.0770161 |
| a | 0.0563578 |
| and | 0.0480214 |
| that | 0.0413389 |
| it | 0.0388320 |
| of | 0.0332038 |
| me | 0.0312326 |

# More info on Amortized Analysis

Your text book (Goodrich and Tamassia): pages 36–41

Introduction to Algorithms, 3rd edition (Cormen et al): Chapter 17

http://www.cs.princeton.edu/ fiebrink/423/AmortizedAnalysisExplained_Fiebrink.pdf

## Randomization

## Randomization

Many algorithms make use of random numbers in their operation.

Such *randomized algorithms* can be simple and efficient, and may lead to less dependence on the input than deterministic algorithms.

For example, ***randomized quicksort*** has expected runtime of $O(n \log n)$, even on a sorted list (see p237–238 of course text book).

## Randomized algorithms and expected running time

Randomized algorithms usually use a pseudorandom number generator (not truly random), but we can still think of them as using a ***coin toss***, ***die roll*** or other random process.

We often want to know how long something will take ***in expectation*** (i.e., on average), given the use of the random process.

A simple fact from probability is that the expected waiting time for an event to occur $k$ times, is $m.k$, if the event occurs with independent probability of $1/m$ per time-step.

Question: what is the expected waiting time to roll 3 sixes on a six-sided die?

## Randomized algorithms and expected running time

**Randomized quicksort** (pp 227–228, Goodrich and Tamassia).

Recall that quicksort makes use of ***pivot*** elements, which are selected from the input.

Sorting occurs fastest when the pivot used at each partition level of the list, partitions it evenly (into two lists of roughly equal size).

Randomized quicksort **selects the pivots at random**.

Given a list of $n$ different elements, what is the chance that a randomly chosen pivot creates two sublists of size at least $n/4$ and at most $3n/4$?

## Randomized algorithms and expected running time

Given a list of $n$ different elements, what is the probability that a randomly chosen pivot creates two subsequences of size at least $n/4$ and at most $3n/4$?

The answer is 1/2. (Imagine the input numbers laid out in descending order. A pivot value selected anywhere after the first 1/4 and before the last 3/4 of the list will work. But the input order doesn't matter since the choice is made randomly.)

1 2 3 4 5 6 7 8 9 10 11 12
E.g. choose the pivot 4:
1 2 3 // **4** 5 6 7 8 9 10 11 12 (list is split into sizes 1/4 and 3/4)

Any of the pivots
4,5,6,7,8,9 would have achieved a split as good (or better) than this

If we keep on partitioning the list into a list 3/4 the size it was before, we will only need to do this $\log_{4/3} n$ times to get down to lists of size 1.

Since this ‘good’ split happens half the time, the expected number of pivot choices is $2 \log_{4/3} n$, which is $O(\log n)$.

Since each use of the pivot to create a partition takes $O(n)$ time, randomized quicksort runs in $O(n \log n)$ expected time.

## Linearity of Expectation

Expected values are easy to work with, as they combine linearly.

So, if you have two events for which you need to compute the expected time, then the expected time of the sum of the two events is just the sum of the expected times of each event.

Rules for working with expected values:

$$E[X + c] = E[X] + c \tag{8}$$

$$E[X + Y] = E[X] + E[Y] \tag{9}$$

$$E[cX] = c.E[X] \tag{10}$$

where $X$ and $Y$ are random variables, and $c$ is any constant.

## Randomized algorithms — summary

Analysing the running time of randomized algorithms

A simple analysis can be done by considering the following, and stating your assumptions:

- Where is the randomization occurring ?
- How does this affect the worst case ?
- Can I work out how long the randomized step(s) will take ***on average*** ?

Then:

- Combine any steps using the (linear) rules of expectations.
- Report the complexity as the *expected time* to terminate.

## Revision / Practice

## Complexity: Reasoning about two variables

The complexity of graph algorithms (see next semester) is often expressed in terms of $|V|$ the number of vertices, and $|E|$ the number of edges, in the graph.

Two algorithms for finding a minimum spanning tree in a graph are Prim's algorithm and Kruskal's algorithm.
Prim's algorithm has running time $O(|V|^2)$
Kruskal's algorithm has running time $O(E \log V)$

True of False? (Discuss and explain your answers)

- Prim's algorithm can be described as quadratic in $V$

- Kruskal's algorithm is intractable
- For constant $V$, Kruskal's algorithm is linear

When is Kruskal's algorithm likely to be quicker than Prim's? (Explain)

When is Prim's algorithm likely to be quicker than Krukal's? (Explain)

## Revision topics — Complexity

Basic level:

- Definition of an algorithm (must terminate with result in finite time)
- Recognition of relative sizes of growth rates
- Simplify big-oh expressions
- Derive big-oh expressions from pseudocode (space and time complexity), focusing on worst case
- Interpret experimental results on complexity, such as log-log plots

## Revision topics — Complexity

Advanced topics:

- Work with big-oh expressions of ***two or more variables***
- Work with **big-theta** and **big-omega**
- Use ***amortized analysis*** on basic data-structures
- Reason about the complexity of **randomized algorithms** (expected running time)

Thanks !

## Final Word About Labs

**Please note: you should get all of your lab exercises except lab 9 (Pangolins) marked before the end of semester.**

**No marking of labs 2–8 will be done next semester!**