Time functions

Reading chapters 2 and 3 of Cormen et al.

Suppose we run a deterministic algorithm a on input x. The time taken to finish is $Time_a(x)$.

But on what computer?

Let's define a standard computer called a **Random Access Machine**

- The RAM has an infinite store of memory location.
- Each location can hold an arbitrary integer.
- **Uniform cost model:** We'll assume that each machine instruction takes 1 unit of time.
- Machine instructions include
  - ∗ store, fetch, indirect store, indirect fetch
  - ∗ add, subtract, multiply, shift left, shift right, and, bitwise and, or and not.
  - ∗ branch, conditional branch, call, return

The exact details won't matter because most of the time we'll consider only the asymptotic complexity (see below).

Deficiencies of the RAM model

— Camelot!
—It's only a model.

In some ways this model is rather unrealistic.

Real machines do not take the same amount of time for each instruction. In particular a fetch instruction may take 10s of times more if the word is not in cache — millions of times more if the word needs to be paged in.

Real memory locations are not infinitely big. For example consider

```
$p := 2$
$i := 0$
while $i < n$ do $p := p \times p; i := i + 1$ end while
```

In time of about $13n + 4$ it computes $2^{(2^n)}$. For large n this is not accurate as, on any real-world machine, we will need several words to represent p and each multiplication will take time dependent on i (or n).

[One way around this problem is to charge $\max(\log_2 |n|, 1)$ for each operation where n is the number of bits involved. This **logarithmic cost model** is less open to abuse, but in most ways less realistic than the uniform cost model.]

Advantages of the RAM model

If not abused, the model gives reasonable results up to a change of coefficients.

It is simple.

Worst-case, average-case, best-case

It is tedious to work out the time function for every input.

It is usually enough to know how time grows as a function of input size.

Each input x is associated with a size $S_a(x)$. Usually this is the number of bits or words required to represent the input.

For example, if we represent a graph (V, E) by an adjacency list representation we need $|V| + 2|E|$ words, provided the number of nodes and edges is not absurdly large.

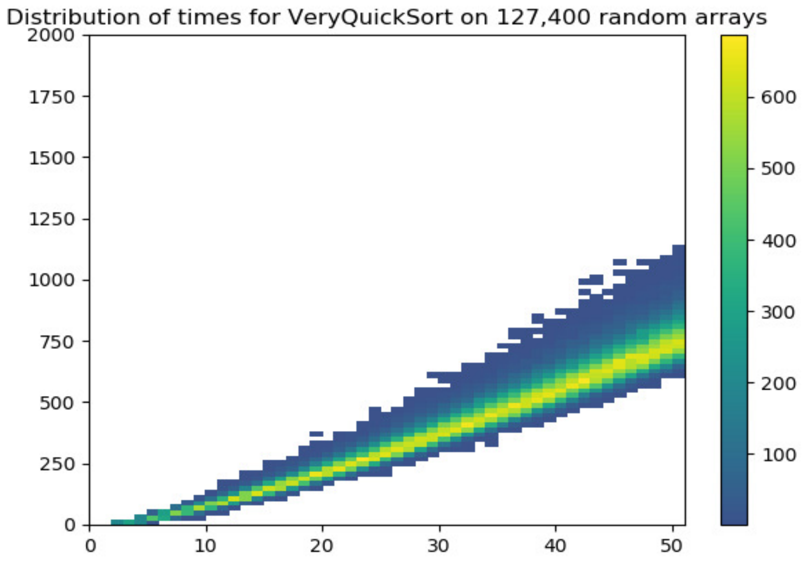
For some problems (e.g. matrix problems) it is conventional to use other measures (e.g. the square root of the number of words).

Let $I_a(n) = \{x \mid S_a(x) = n\}$

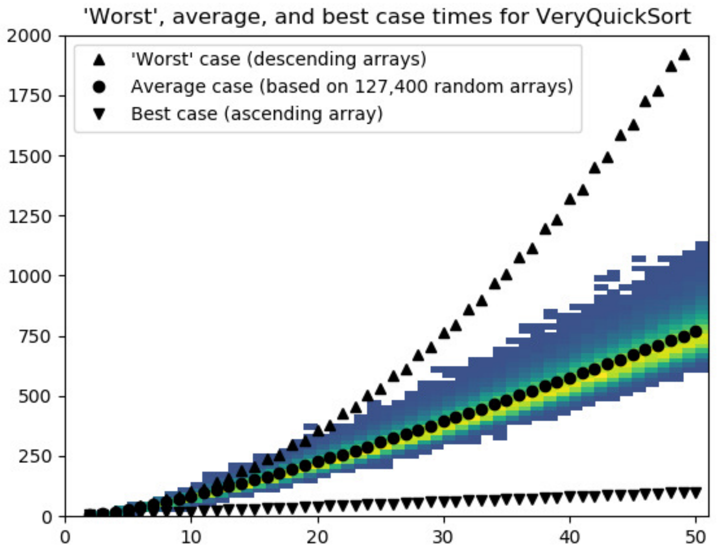
- Worst case time function
$$T_a(n) = \max_{x \in I_a(n)} Time_a(x)$$
- Best case time function
$$BT_a(n) = \min_{x \in I_a(n)} Time_a(x)$$
- Average case time function (uniform distribution)
$$AT_a(n) = \frac{1}{|I_a(n)|} \sum_{x \in I_a(n)} Time_a(x)$$
- Average case time function (for distribution p)
$$AT_{a,p}(n) = \sum_{x \in I_a(n)} p(x) \times Time_a(x)$$

An experiment

I ran a sorting algorithm (VeryQuickSort) on a large number of randomly generated arrays. More popular times are more yellow.[1]



I also used arrays constructed to require the least or the maximum[2] amount of time.



[1] Source available at https://github.com/theodore-norvell/VeryQuickSort
Only fetches, stores, and comparisons were counted. Input size is the array length.

[2] The inputs that I constructed to represent the worst case don't actually aways take the most time. But they are very close to the worst.

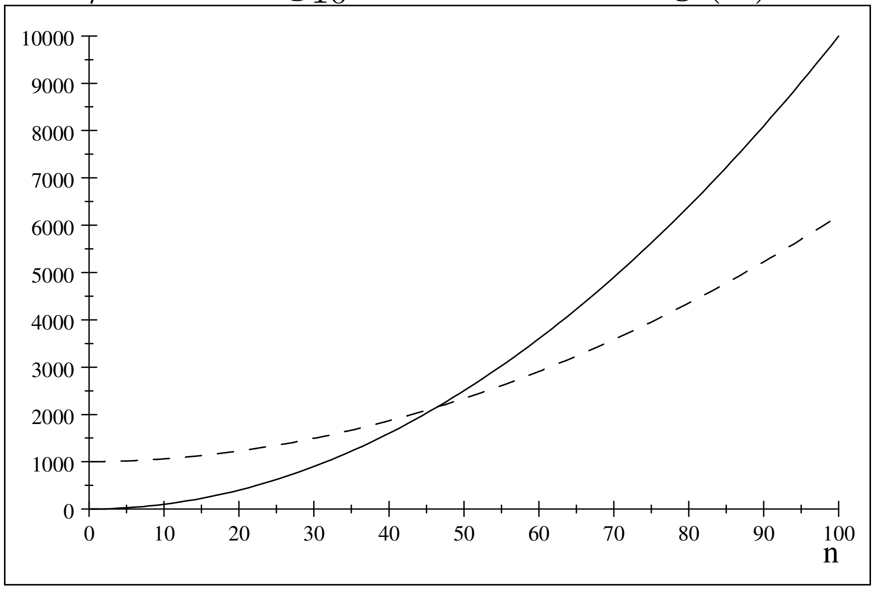
Complexity of functions

Asymptotically similar functions

Two similar functions

Throughout this section we'll consider functions f and g in $\mathbb{N} \to \mathbb{R}$ that are ultimately positive. (I.e. that for all sufficiently large n, $f(n)$ and $g(n)$ are positive.)

Let $f(n) = n^2/2 + n \log_{10} n + 1000$ and $g(n) = n^2$



— n^2 - - - $n^2/2 + n \log_{10} n + 1000$

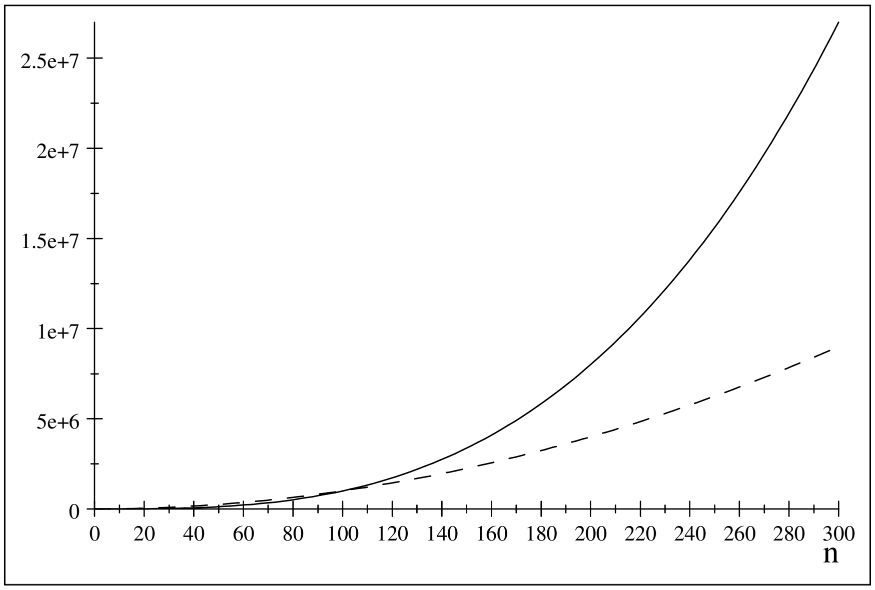
As n approaches infinity, both $f(n)$ and $g(n)$ approach infinity.

f is approaching infinity half as fast in the limit.

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \frac{1}{2}$$

Two dissimilar functions

Now consider $p(n) = 100n^2$ and $q(n) = n^3$



— n^3 - - -$100n^2$

We can see that, p is approaching infinity much slower than q. Infinitely slower in the sense that

$$\lim_{n\to\infty}\frac{p(n)}{q(n)} = 0$$

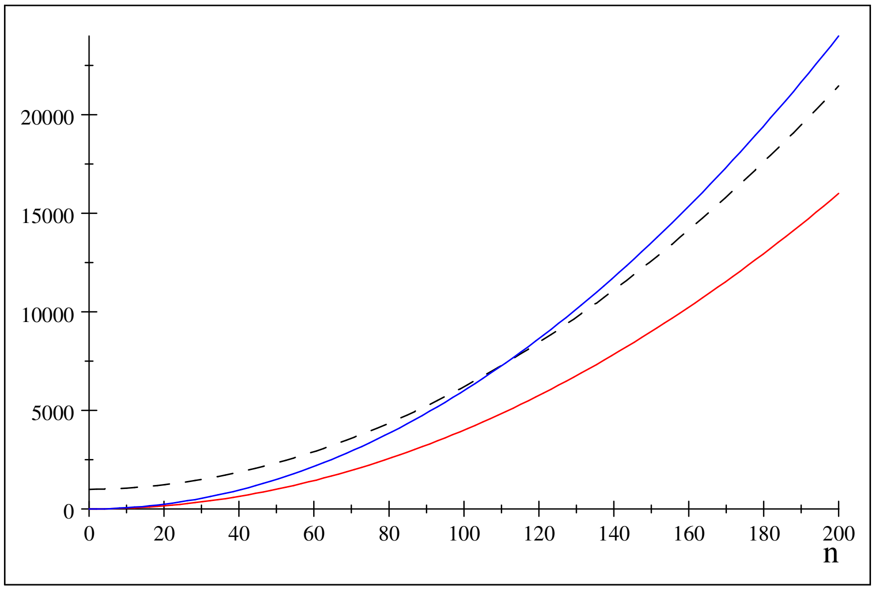
Definition of "asymptotically similar to"

When $0 < \lim_{n\to\infty} f(n)/g(n) < \infty$, we have the following property:

- There are positive numbers a and b such that $f(n)$ is eventually trapped between $a \times g(n)$ and $b \times g(n)$

Consider $f(n) = n^2/2 + n\log_{10} n + 1000$ and $g(n) = n^2$.

Take $a = 0.4$ and $b = 0.6$.



red $0.4 \times n^2$ blue $0.6 \times n^2$ - - - $n^2/2 + n \log_{10} n + 1000$

Eventually we have $0.4 \times g(n) < f(n) < 0.6 \times g(n)$

Definition: Function f is **asymptotically similar to** g if and only if there exist positive numbers a, b, and m such that

$$\forall n \in \mathbb{N} \cdot n > m \Rightarrow a \times g(n) < f(n) < b \times g(n)$$

Notation: We write $f \asymp g$ to mean that f is asymptotically similar to g.

[This definition has the advantage it works even when the limit is not defined.]

Exercise: Show that, for any functions p and q, if there are positive numbers a, b, and m such that

$$\forall n \in \mathbb{N} \cdot n > m \Rightarrow a \times q(n) < p(n) < b \times q(n)$$

there are also positive numbers a', b', and m' such that

$$\forall n \in \mathbb{N} \cdot n > m' \Rightarrow a' \times p(n) < q(n) < b' \times p(n)$$

This shows that $\asymp$ is symmetric, I.e., for all functions f and g, $f \asymp g$ if and only if $g \asymp f$

Big-Theta notation

Suppose g is a function, the set of all functions that are asymptotically similar to g is written $\Theta(g)$.

That is, for any f and g,

$$f \asymp g \text{ if and only if } f \in \Theta(g)$$

[**Some fishy notation:** Rather than writing, for example,

"$\Theta(g)$ where $g(n) = n^3$",

we can use an anonymous function like this

"$\Theta(\lambda n \cdot n^2)$".

But usually people simply write

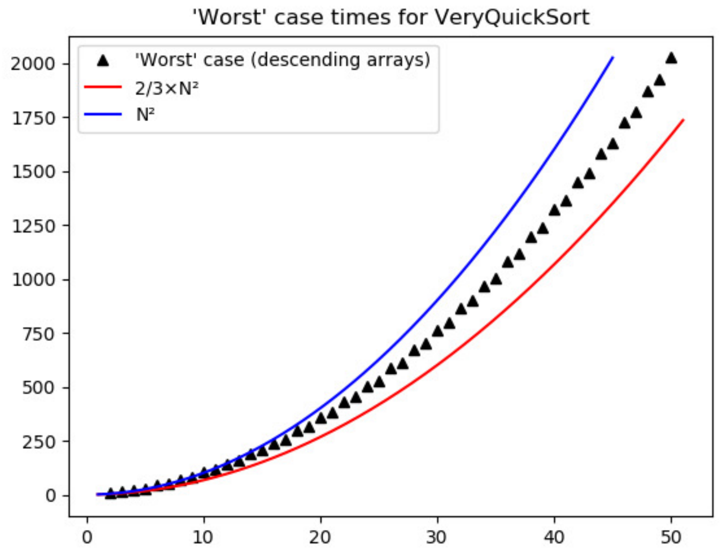
"$\Theta(n^2)$".

The n^2 in this context is refering to the function $\lambda n \cdot n^2$. I.e., n is not a free variable here.]

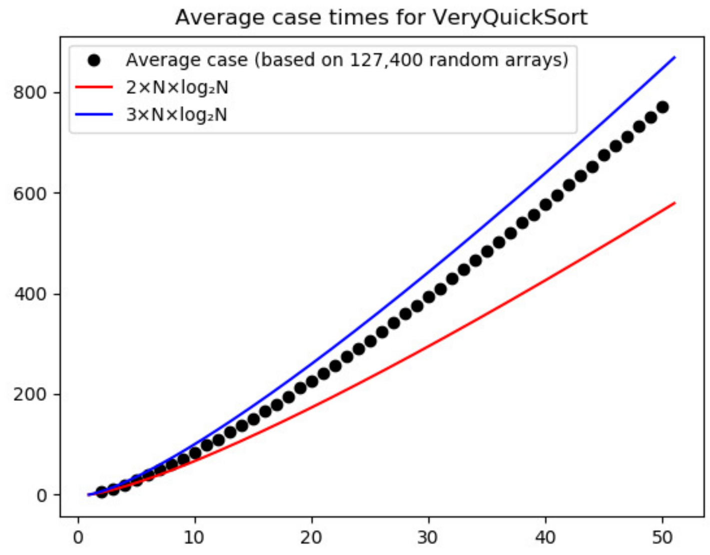
Examples

From the experimental data for VeryQuickSort:

- It seems that the worst case time function is in $\Theta(n^2)$.



- It seems that the average case time function is in $\Theta(n \log_2 n)$



- [[More examples]]

Some properties of Θ

Smaller terms don't matter. For example

$$\Theta(n^2 + n) = \Theta(n^2)$$

The reason is that $n \prec n^2$, so eventually the n just doesn't matter.

Constant coefficients don't matter. For example

$$\Theta(3n^2) = \Theta(n^2)$$

Conventionally we omit constant coefficients and write $\Theta(c)$ as $\Theta(1)$ for c a constant.

Bases of logs don't matter. For example

$$\Theta(\log_2 n) = \Theta(\log_2 10 \times \log_{10} n) = \Theta(\log_{10} n)$$

Conventionally we write $\log n$ rather than $\log_b n$ for a fixed b.

Dominated functions are excluded

$$\Theta(n) \nsubseteq \Theta(n \log n) \nsubseteq \Theta(n^2) \nsubseteq \Theta(n^3) \nsubseteq \Theta(2^n)$$

Some time-complexity results

Working out the asymptotic time-complexity of an algorithm is usually quite simple.

Focus on the most common operation. If you can show that an operation x is done fewer times than an operation y, then you can just ignore y.

Selection sort

```
var $i := 0$
{ inv: $0 \le i \le n$ and the first $i$ items of $a$ are the $i$ smallest
and are sorted }
while $i < n$ do
    var $j := i + 1$
    var $k := i$
    var $m := a(k)$
    { inv $i \le k < j \le n$ and $m = a(k)$ and $m \le^* a\{i, ..j\}$
    }
    while $j < n$ do
        if $a(j) \le m$ then $k := j$ $m := a(k)$ end if
        $j := j + 1$
    end while
    $a(k) := a(i)$
    $a(i) := m$
    $i := i + 1$
end if
```

A most common operation is $j < n$. This is executed $\sum_{i=0}^{n-1} (n - i) = \frac{n^2+n}{2}$ times. And $\frac{n^2+n}{2} \in \Theta(n^2)$.

Merge sort

To avoid recursion, we can simply combine bigger and bigger regions of the array.

```
procedure *mergeSortNR*( var $a$ : $\mathrm{array}[T]$ )
implements *sort*( $a, 0, a.\mathrm{length}$ )
    var $grain := 1$
    inv each of the segments $a[0, ..grain]$, $a[grain, ..2 \times grain]$, $a[2 \times grain, ..3 \times grain]$ etc on up to and including $a[\left\lfloor \frac{a.length}{grain} \right\rfloor \times grain, ..a.\mathrm{length}]$ is sorted
    while $grain < a.\mathrm{length}$ do
        var $p := 0$
        while $p < a.\mathrm{length}$ do
            val $q := \min(p + grain, a.\mathrm{length})$
            val $r := \min(q + grain, a.\mathrm{length})$
            merge( $a, p, q, r$ )
            $p := r$
        end while
        $grain := grain \times 2$
    end while
end *mergeSortNR*
```

Each merge operation is $\Theta(r - p)$ (best and worst case).

- So the inner loop is $\Theta(n)$ where $n = a.\mathrm{length}.$
- The outer loop is executed $1 + \lfloor \log_2 n \rfloor$
- So the best and worst (and hence average) time complexity is $\Theta(n \log n)$.

Comparing merge sort to selection sort

(Warning completely made-up numbers ahead)

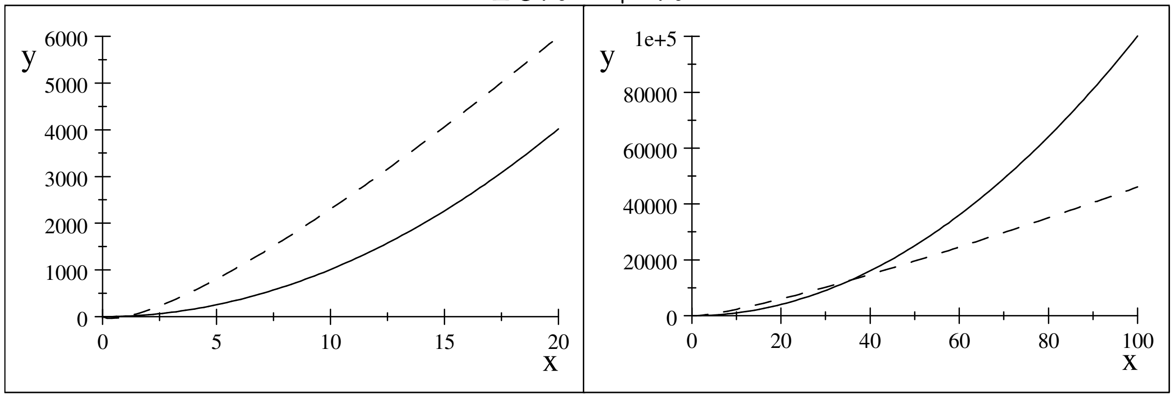
Suppose, for the sake of argument, that, on a particular real machine

$$T_{selectionSort}(n) = 10n^2 + n \text{ ns}$$

and

$$T_{mergeSort}(n) = 100n \log n \text{ ns}$$

$$10n^2 + n$$



- - - merge — selection - - - merge — selection

QuickSort

```
procedure quickSort( var $a$ : array$[T]$; $p, r$ : $Int$ )
implements sort($a, p, r$)
    if $r - p > 1$ then
        val $i$ := any value from $\{p, ..r\}$
        val $x := a(i)$
        var $q$
        partition( $a, p, r, x, q$ )
        { $p \le q < r$
            everything in $a\{p, ..q\}]$ is $\le x$
            and $a(q) = x$
            and everything in $a\,\{q + 1, ..r\}$ is $\ge x$ }
        quickSort( $a, p, q$ )
        quickSort( $a, q + 1, r$)
    end if
end quickSort
```

Worst case

$\text{partition}(a, p, r, x, q)$ takes $\Theta(r - p)$.

In the worst-case $q = p$ (or $q = r - 1$). Then the partitions that need to be done are on intervals of size n, $n{-}1$, , $n{-}2$, ... , 2. So the time is in

$$\begin{aligned} & \Theta(n) + \Theta(n{-}1) + \cdots + \Theta(2) \\ = \ & \Theta(\frac{n^2 + n}{2} - 1) \\ = \ & \Theta(n^2) \end{aligned}$$

Average case (Rough argument)

Consider only data comparisons (the most frequent operation).

Assume i is chosen randomly and that partition is fair.

Partition requires $r - p - 1$ comparisons.

If you break a 1m stick in two, what is the expected length of the longer piece?

On average the array will break into pieces of size $\frac{1}{4}(r - p)$ and $\frac{3}{4}(r - p)$ so the depth of the call tree is about $\log_{4/3} n$. If partition is implemented well, the number of comparisons at each level is less than n. So the total is less than $n \log_{4/3} n = n \times \frac{\ln n}{\ln 4/3} \cong 3.5 n \ln n$.

You can see that this analysis is a bit pessmistic. We can not rely on a consistant $25 : 75$ split. Half the time it will be better and half the time it will be worse. When it is better (closer to $50 : 50$) this helps a lot. When it is worse (closer to $0 : 100$), this does not hurt much.

[A more careful analysis gives the expected number of comparisons as $2n \ln n$]

So quicksort takes average time $\Theta(n \log n)$.

The robustness of asymptotic results

The exact speed that an algorithm takes depends on

- The details of how the algorithm is coded in a high-level language
- The details of how that coding is translated to machine code
- The properties of the processor (e.g. clock speed and instructions per clock)
- Which input is used.
- Happenstance (e.g. cache misses)

By looking at asymptotic time complexity of the worst-case (or average-case) time function on a RAM machine.

We get a result that

- Does not depend on the details of how the algorithm is coded.
  - ∗ But will depend on data-structure selection
- Does not depend on the details of the translation to machine code
- Does not depend on the properties of any processor
- Does not depend on which input is used.
- Does not depend on happenstance.

In general, the Θ time complexity on a RAM model will be the same as the Θ time complexity on any real hardware, provided the real machine does not run out of space.

(Most of) The over simplifications of the RAM model just don't matter, when combined with the over simplifications of asymptotic complexity.