Design and Analysis of Algorithms

Part 1

Program Costs and Asymptotic Notations

Tom Melham

Hilary Term 2015

Fast computers *vs* efficient algorithms

Web-based innovations (e.g. Google algorithms, wikis, blogs, etc.) rely on

- fast computers, and
- efficient algorithms.

Which is more important?

Fast computers *vs* efficient algorithms

Cost of running an algorithm is usually a function $T(n)$ of the *input size* n. Assume $T(n)$ is the number of steps an algorithm takes to compute a size-n problem, and a computer capable of 10^{10} steps per second.

Cost $T(n)$ (Complexity)	Maximum problem size solvable in		
	1 second	1 hour	1 year
n	10^{10}	3.6×10^{13}	3×10^{17}
n^2	10^5	6×10^6	5×10^8
n^3	2154	33000	680000
2^n	33	45	58

- Replacing a cubic by a quadratic algorithm, we can solve a problem roughly 180 times larger (in an hour).
- Replacing a quadratic by a linear algorithm, we can solve a problem roughly 6000000 times larger!

Why is efficient algorithm design important?

Suppose a faster computer is capable of 10^{16} steps per second.

Cost $T(n)$	Max. size before	Max. size now
n	s_1	$10^6 \times s_1$
n^2	s_2	$1000 \times s_2$
n^3	s_3	$100 \times s_3$
2^n	s_4	$s_4 + 20$

A $10^6\times$ increase in speed results in only a factor-of-100 improvement if complexity is n^3, and only an additive increase of 20 if complexity is 2^n!

Conclusions As computer speeds increase ...

1. ... it is algorithmic efficiency that really determines the increase in problem size that can be achieved.
2. ... so does the size of problems we wish to solve. Thus designing efficient algorithms become even more important!

Notations can make all the difference

Consider the Roman numerals:
$I = 1, V = 5, X = 10, L = 50, C = 100, D = 500, M = 1000.$
How would you compute

$$MDCXLVIII + DCCCXII?$$

Invented in India around AD 600, the *decimal system* was a revolution in quantitative reasoning.

Using ten symbols, combined with a *place-value system* – each symbol has ten times the weight of the one to its right, even large numbers can be written compactly.

Arabic mathematicians developed arithmetic methods using the Indian decimals (including decimal fractions), leading to the *decimal point*.

From *Algorism* to *Algorithms*

A 9th-century Arabic textbook by the Persian *Al Khwarizmi* was the key to the spread of the Indian-Arabic decimal arithmetic.

He gave methods for basic arithmetics (adding, multiplying and dividing numbers), even the calculation of square roots and digits of π.

Derived from 'Al Khwarizmi', *algorism* means rules for performing arithmetic computations using the Indian-Arabic decimal system.

The word "algorism" devolved into *algorithm*, with a generalisation of the meaning to

Algorithm: a finite set of well-defined instructions for accomplishing some task.

Evaluating algorithms

Two questions we ask about an algorithm

1. Is it correct?
2. Is it efficient?

Correctness - of utmost importance. It is easy to design a highly efficient but incorrect algorithm.

Efficiency with respect to:

- **Running time**
- Space = amount of memory used
- Network traffic
- Others. E.g. number of times secondary storage is accessed.

Measuring running time

The running time of a *program* depends on many factors:

1. The *running time* of the algorithm.
2. The input of the program. Normally the larger the input, the longer it takes to compute.
3. The quality of the implementation (e.g. quality of the code generated by the compiler).
4. The machine running the program.

We are concerned with 1.

The Sorting Problem

Input: A sequence of n numbers $a_0, \cdots, a_{n-1}$.

Output: A permutation $a_{\sigma(0)}, \cdots, a_{\sigma(n-1)}$ of the input such that

$$a_{\sigma(0)} \leq a_{\sigma(1)} \leq \cdots \leq a_{\sigma(n-1)}.$$

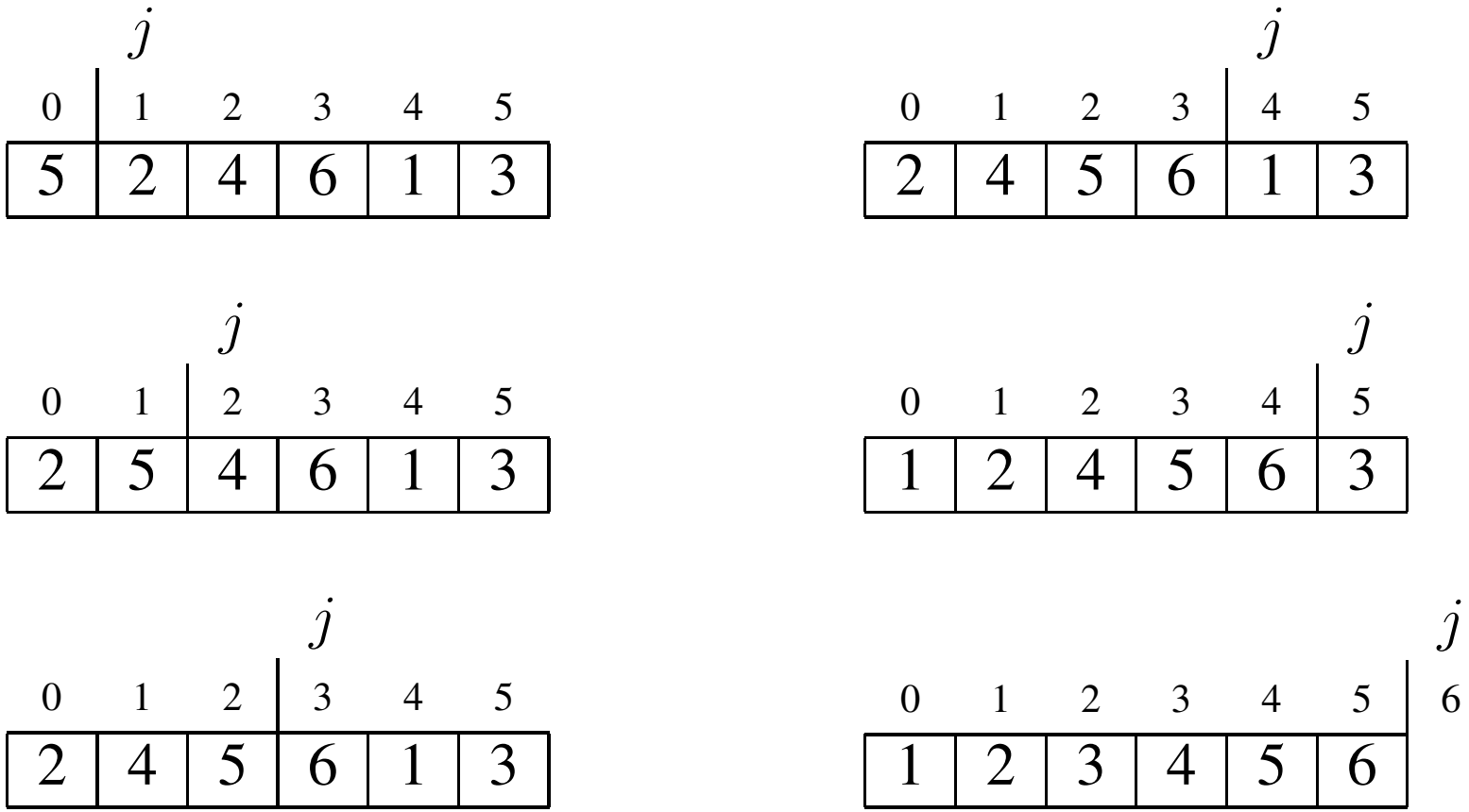
The sequences are typically stored in arrays. The numbers are sometimes called *keys*.

Insertion sort

Insertion sort: Informal description

- The sorted array (output) is built one entry at a time.
- At each iteration, the first remaining entry of the input is removed and *inserted* into the result at the right position, extending the result.
- The result array after k iterations contains the first k entries of the input array and is sorted.
- This goes on until no element is left in the input.

Example



Observation. At the start of each iteration of the outer **for** loop, the subarray $A[0\,..\,j)$ consists of the elements originally in $A[0\,..\,j)$ but in sorted order.

Expressing algorithms in pseudocode of [CLRS]

- Similar to Pascal, C and Java.
- Pseudocode is for communicating algorithms to humans: many programming issues (e.g. data abstraction, modularity, error handling, etc.) are ignored.
- English statements are sometimes used.
- "// " indicates that the reminder of the line is a comment.
  (In 2nd edition "▷" is used.)
- Variables are local to the block, unless otherwise specified.
- Block structure is indicated by indentation.
- Assignment "$x = y$" makes x reference the same object as y.
  (In 2nd edition "←" is used.)
- Boolean operators "and" and "or" are "short-circuiting".

Insertion sort in pseudocode

INSERTION-SORT(A)

Input: An integer array A
Output: Array A sorted in non-decreasing order

```
1  for j = 1 to A.length − 1
2      key = A[j]
3      // Insert A[j] into the sorted sequence A[0 . . j).
4      i = j
5      while i > 0 and A[i − 1] > key
6          A[i] = A[i − 1]
7          i = i − 1
8      A[i] = key
```

Loop-invariant approach to correctness proof

Three key components of a loop-invariant argument:

1. *Initialization*: Prove that invariant (I) holds prior to first iteration.
2. *Maintenance*: Prove that if (I) holds just before an iteration, then it holds just before the next iteration.
3. *Termination*: Prove that when the loop terminates, the invariant (I), *and* the reason that the loop terminates, imply the correctness of the loop-construct

The approach is reminiscent of mathematical induction:

1. Initialization corresponds to establishing the base case.
2. Maintenance corresponds to establishing the inductive case.
3. The *difference* is that we expect to exit the loop, but mathematical induction establishes a result for all natural numbers.

Correctness of INSERTION-SORT

Invariant of outer loop: $1 \leq j \leq n$ and the subarray $A[0\,..\,j)$ consists of the elements originally in $A[0\,..\,j)$ but in sorted order. ($n = A.length$)

Initialization: When $j = 1$, then subarray $A[0\,..\,j)$ is a singleton and therefore sorted.

Termination. The outer **for** loop terminates when $j = n$. Thus, with $j = n$, the invariant reads: $A[0\,..\,n)$ consists of the elements originally in $A[0\,..\,n)$ but in sorted order.

Maintenance. Suppose input is sequence $a_0, \cdots, a_{n-1}$. We aim to prove:

Suppose $j = \mathrm{k}$. *If at the start of an iteration* $1 \leq \mathrm{k} \leq n$ *and* $A[0\,..\,j)$ *consists of* $a_0, \cdots, a_{j-1}$ *in sorted order, then at the start of the next iteration (i.e.* $j = \mathrm{k}{+}1$*),* $1 \leq \mathrm{k}{+}1 \leq n$ *and* $A[0\,..\,j)$ *consists of* $a_0, \cdots, a_{j-1}$ *in sorted order.*

Correctness of INSERTION-SORT, contd.

Body of inner **while** loop works by moving each of $A[j-1], A[j-2], \cdots$, by one position to the right until the proper position for key is found, and where it is then inserted.
Suppose just before reaching the inner loop $A[0\,..\,j)$ contains the sorted sequence $a_0, \cdots a_{j-1}$.

Invariant of inner loop: $0 \leq i \leq j$ and $A[0\,..\,i)\ A[i+1\,..\,j+1)$ contains the sequence $a_0, \cdots a_{j-1}$ and all elements in $A[i+1\,..\,j+1)$ are bigger than key.
Initialization. For $i = j$, invariant is true as nothing has moved yet.
Termination. Either $i = 0$ or $A[i-1] \leq key$, and all elements in $A[i+1\,..\,j+1)$ are bigger than key. Hence $A[0\,..\,i)\ key\ A[i+1\,..\,j+1)$ is the same sequence as $a_0, \cdots a_{j-1}$ with key inserted in the correct position.
Maintenance. Using the fact that the body is only executed if $i > 0$ and $A[i-1] > key$ the invariant is true again after executing both statements of the body.

Running time analysis

- Assume line i takes time c_i, a constant.
- For $j = 1, \cdots, n-1$, let t_j be the number of times the test of the **while** loop is excecuted for that value of j.
- When a **for** or **while** loop exits normally, the test is executed one more time than the loop body.

Running time is

$$\sum_{\text{all statements}} (\text{cost of statement}) \cdot (\text{number of time statement executed})$$

Let $T(n)$ be the running time of INSERTION-SORT where $n = A.length$. We have:

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_3(n-1) + c_4(n-1) + c_5 \textstyle\sum_{j=1}^{n-1} t_j \\ &\quad + c_6 \textstyle\sum_{j=1}^{n-1}(t_j - 1) + c_7 \textstyle\sum_{j=1}^{n-1}(t_j - 1) + c_8(n-1) \end{aligned}$$

Worst-case analysis

- The input array contains distinct elements in reverse sorted order i.e. is strictly decreasing.
- Why? Because we have to compare $key = a_j$ with every element to left of the $(j+1)$th element, and so, compare with j elements in total.
- Thus $t_j = j + 1$. We have $\sum_{j=1}^{n-1} t_j = \sum_{j=1}^{n-1}(j+1) = \frac{n(n+1)}{2} - 1$, and so,

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5(\tfrac{n(n+1)}{2} - 1) \\ &\quad + c_6(\tfrac{n(n-1)}{2}) + c_7\tfrac{n(n-1)}{2} + c_8(n-1) \\ &= an^2 + bn + c \end{aligned}$$

for appropriate a, b and c.

Hence $T(n)$ is a *quadratic* function of n.

Best-case analysis

- The array is already sorted.
- Always find $A[i] \leq key$ upon the first iteration of the **while** loop (when $i = j$).
- Thus $t_j = 1$.

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) \\ &= (c_1 + c_2 + c_4 + c_5 + c_8)n + (c_2 + c_4 + c_5 + c_8) \end{aligned}$$

I.e. $T(n)$ is a *linear* function of n.

Average-case analysis

- Randomly choose n numbers as input.
- On average key in $A[j]$ is less than half the elements in $A[0\,..\,j)$ and greater than the other half, and so, on average **while** loop has to look halfway through the sorted subarray $A[0\,..\,j)$ to decide where to drop key.
- Hence $t_j = j/2$.
- Although average-case running time is approximately half that of the worst-case, it is still quadratic.

Moral. Average-case complexity *can* be *asymptotically* as bad as the worst!

Average-case analysis is not straightforward:

- What is meant by "average input" depends on the application.
- The mathematics can be difficult.

Features of insertion sort: A summary

Minuses

- Worst-case quadratic time.

Pluses

- Efficient on "small" data sets.
- Efficient (linear-time) on already sorted (or nearly sorted) input.
- ***Stable***: Relative order of elements with equal keys is maintained.
- ***In-place***: Only a constant amount of extra memory space (other than that which holds the input) is required, regardless of the size of the input.
- ***Online***: it can sort a list as it is received.

Insertion sort in Haskell

```
> isort = foldr insert []

> insert :: Ord a => a -> [a] -> [a]
> insert x [] = [x]
> insert x (y:ys) =
>   if x > y then y : insert x ys
>            else x : y : ys
```

Worst case analysis:

$T_{insert}(m, 0) \leq c_1$ and $T_{insert}(m, n+1) \leq T_{insert}(m, n) + c_2$

gives $T_{insert}(m, n) \leq c_2 n + c_1$

$T_{isort}(0) \leq c_3$ and $T_{isort}(n+1) \leq T_{insert}(m, n) + T_{isort}(n) + c_4$

gives $T_{isort}(n) \leq c_2(\frac{n(n-1)}{2}) + (c_1 + c_4)n + c_3$.

The Big-O Notation

Let $f, g : \mathbb{N} \longrightarrow \mathbb{R}^+$ be functions. Define the set

$$O(g(n)) \;:=\; \{\, f : \mathbb{N} \longrightarrow \mathbb{R}^+ : \exists n_0 \in \mathbb{N}^+ \,.\, \exists c \in \mathbb{R}^+ \,.\, \forall n \,.\; n \geq n_0 \;\rightarrow\; f(n) \leq c \cdot g(n) \,\}$$

In words, $f \in O(g)$ just if there exist a positive integer n_0 and a positive real c such that for all $n \geq n_0$, we have $f(n) \leq c \cdot g(n)$.

Informally $O(g)$ is the set of functions that are bounded above by g, ignoring constant factors, and ignoring a finite number of exceptions.

Equivalently, $f \in O(g)$ just if there exist positive reals a and b such that for all n, we have $f(n) \leq a \cdot g(n) + b$. (Proof: exercise)

Examples

$$O(g(n)) \;:=\; \{\, f : \mathbb{N} \longrightarrow \mathbb{R}^+ : \exists n_0 \in \mathbb{N}^+ \,.\, \exists c \in \mathbb{R}^+ \,.\, \forall n \,.$$
$$n \geq n_0 \;\rightarrow\; f(n) \leq c \cdot g(n) \,\}$$

1. $3^{98} \in O(1)$. Just take $n_0 = 1$ and $c = 3^{98}$.
2. $5n^2 + 9 \in O(n^2)$.
   Take $n_0 = 3$ and $c = 6$. Then for for all $n \geq n_0$, we have $9 \leq n^2$, and so $5n^2 + 9 \leq 5n^2 + n^2 = 6n^2 = cn^2$.
3. Take $g(n) = n^2$ and $f(n) = 7n^2 + 3n + 11$. Then $f \in O(g)$.
4. Some more functions in $O(n^2)$:
   - $1000n^2$, n, $n^{1.9999}$, $n^2 / \lg\lg\lg n$ and 6.

A shorthand for "Big O"

We say "g is an ***asymptotic upper bound*** for f" just if $f \in O(g)$.

Instead of writing $f \in O(g)$ we usually write

$$f(n) \;=\; O(g(n))$$

(read "f is Big-O of g").

Informally $f(n) = O(g(n))$ means

> "$f(n)$ grows no faster than (a constant factor of) $g(n)$, for sufficiently large n".

Big-O on the RHS of an equation

It is convenient to write

$$f_1(n) \;=\; f_2(n) + O(g(n))$$

instead of $f_1(n) - f_2(n) = O(g(n))$ (which is the proper way).

Pitfalls of "Big O" notation

When writing

$$f(n) \;=\; O(g(n))$$

bear in mind that it is a shorthand for $f(n) \in O(g(n))$.
So "=" does not have all the usual properties of equality. In particular it is not symmetric!

Examples

- $n = O(n^2)$ but $n^2 \neq O(n)$.
- $n = O(n^3)$ and $n^2 = O(n^3)$ but $n \neq n^2$.

So why use the big-O notation?

- More than 100 years old (so very standard mathematical notation), it is too well established to change now.
- We can and should read "=" as "is". Note that "is" does not imply equality.
- We already abuse the "=" symbol in computer science...

Properties of Big-O

Notation. $\log_2 n$ is sometimes written $\lg n$; and $\log_e n$ is sometimes written $\ln n$ ("n" for natural logarithm).

Properties of Big-O

Lemma 1. *Let* $f, g, h : \mathbb{N} \longrightarrow \mathbb{R}^+$. *Then:*

1. *For every constant* $c > 0$, *if* $f \in O(g)$ *then* $c\,f \in O(g)$.
2. *For every constant* $c > 0$, *if* $f \in O(g)$ *then* $f \in O(c\,g)$.
   (E.g. $f(n) = O(\ln n)$ *iff* $f(n) = O(\log_{10} n)$. *Thus it is preferable to be "neutral" and write* $f(n) = O(\log n)$.*)*
3. *If* $f_1 \in O(g_1)$ *and* $f_2 \in O(g_2)$ *then* $f_1 + f_2 \in O(g_1 + g_2)$.
4. *If* $f_1 \in O(g_1)$ *and* $f_2 \in O(g_2)$ *then* $f_1 + f_2 \in O(\max(g_1, g_2))$.
5. *If* $f_1 \in O(g_1)$ *and* $f_2 \in O(g_2)$ *then* $f_1 \cdot f_2 \in O(g_1 \cdot g_2)$.
6. *If* $f \in O(g)$ *and* $g \in O(h)$ *then* $f \in O(h)$.
7. *Every polynomial of degree* $l \geq 0$ *is in* $O(n^l)$.
8. *For any* $c > 0$ *in* $\mathbb{R}$, *we have* $\lg(n^c) \in O(\lg(n))$.
9. *For every constant* $c, d > 0$, *we have* $\lg^c(n) \in O(n^d)$.
10. *For every constant* $c > 0$ *and* $d > 1$, *we have* $n^c \in O(d^n)$.

Question and Example

We have seen that $O(\lg n) = O(\ln n)$.
Question. Is $O(2^{\lg n})$ the same as $O(2^{\ln n})$?

No. $O(2^{\ln n}) = O(n^{0.6931\ldots})$. Recall: $a^{\log_b c} = c^{\log_b a}$.

Question and Example

Example. We show

$$57n^3 + 4n^2 \cdot \lg^5(n) + 17n + 498 \ \in \ O(n^3)$$

by appealing to Lemma 1:

$$\begin{aligned}
\lg^5(n) &\in O(n) && \because 9 \\
4n^2 \cdot \lg^5(n) &\in O(4n^3) && \because 5 \\
57n^3 + 4n^2 \cdot \lg^5(n) + 17n + 498 &\in O(57n^3 + 4n^3 + 17n + 498) && \because 3 \\
57n^3 + 4n^3 + 17n + 498 &\in O(n^3) && \because 7 \\
57n^3 + 4n^2 \cdot \lg^5(n) + 17n + 498 &\in O(n^3) && \because 6
\end{aligned}$$

Big-Θ

We write $f = \Theta(g)$, read "g is an ***asymptotic tight bound*** of f", to mean $f = O(g)$ and $g = O(f)$.

I.e. $f = \Theta(g)$ just if there are positive reals c_1 and c_2 and a positive number n_0 such that for all $n \geq n_0$, we have

$$c_1 g(n) \ \leq \ f(n) \ \leq \ c_2 g(n).$$

We think of f and g as having the "same order of magnitude".

Examples

1. $5n^3 + 88n = \Theta(n^3)$
2. $2 + \sin(\log n) = \Theta(1)$
3. $n! = \Theta(n^{n+1/2}e^{-n})$. (A consequence of Stirling's Approximation.)

Big-Ω

The Big-O notation is useful for upper bounds. There is an analogous notation for lower bounds. We write

$$f = \Omega(g)$$

to mean "there are a positive number n_0 and a positive real c such that for all $n \geq n_0$, we have $f(n) \geq cg(n)$."
We say "g is an ***asymptotic low bound*** of f" just if $f \in \Omega(g)$.

Example

1. $n^n = \Omega(n!)$
2. $\exp(n) = \Omega(n^{10})$.

Exercise

Prove that $f = \Omega(g)$ iff $g = O(f)$.

Some useful results

Logarithms.

Recall the following useful facts. Let $a, b, c > 0$.

$$\begin{aligned} a &= b^{\log_b a} \\ \log_b a &= \frac{\log_c a}{\log_c b} \\ \log_b a &= \frac{1}{\log_a b} \\ a^{\log_b c} &= c^{\log_b a} \end{aligned}$$

A form of ***Stirling's approximation***:

$$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \left(1 + O\left(\frac{1}{n}\right)\right)$$