COMP6700/2140 Correctness and Efficiency

Alexei B Khorev and Josh Milthorpe

Research School of Computer Science, ANU

May 2017

Topics

1. Concept of an Algorithm
2. Correctness
3. Cost of Computation:
   - Time, T
   - Space (Memory, P)
   - Energy
4. Efficiency and Complexity of Algorithms
5. $\mathcal{O}$-expression Notations
6. Complexity Classes
7. Linear Search and Selection Sort
8. Performance Measurement
9. Bubble Sort and Gap Sort

Algorithms: Top-Down Design (Again)

Algorithms occupy the intermediate position between a solution and a method, both in terms of generality and detail. The origin — mathematical (algebraic), the essence — transformation of data (input $\rightarrow$ output) in a finite number of steps.

Writing a computer code often involves solving algorithmic problems. Some of these problems can be handled by the *top-down design* approach:

1. Clearly state the problem that you are trying to solve
2. Define the inputs required by the program and the outputs to be produced
3. Decompose the program into classes and their associated methods
4. Design the algorithm that you intend to implement for each method
   - Repeat every step for devising the solution used at the higher level (step-wise refinement)
5. Turn the algorithm into pseudo-code (easily translatable into Java statements)
6. Test the resulting program

The top-down design is often used together with the *bottom-up design* — translating an existing solution (like a mathematical solution into the code and integration separate parts into one working code). The bottom-up design may also involve the *data structure* selection (or construction); this is called the *data-driven design*.

Algorithms: generality

A good algorithm always has a virtue of generality: it can be used on a *family* of data structures with specific characteristics reduced to minimum. An example — sorting an array of N floats:

This problem is a particular case of a more general sorting problem: *Given an array of `N > 0` objects, return an array which contains these `N` objects in ascending (descending) order.*

It is assumed that array elements can be *compared*, using unspecified criterion (in Java, it means that the object class implements the *Comparable* interface). When implemented with such generality, the algorithm is said to be *generic*. Computer algorithms are characterised by:

1. a large number of repetitive operations: large size array, looping constructs, pipelines
2. a particular data model in case of sorting and searching ("data model" here means type of operations which can be performed of the DS)
3. *correctness* and *efficiency*, where correctness means that a right result is obtained for **all possible** problem instances (size of the data structure, state of the input — sorted, unsorted, random, nearly-sorted etc), while efficiency means how the time and computer resources needed to successfully carry out the algorithm application scale with the problem size N (number of elements in the DS).

Algorithms: Complexity

To quantify the algorithm efficiency, one has to establish how $T(N)$ — the number *atomic* operations (they do not depend on N), needed to perform the algorithm — scales with the size of the problem N. The time of execution depends on hardware platform and state of the execution environment; it is not an invariant measure. The computer science uses the *big-$\mathcal{O}$ notations* (pronounced "big-omicron" by eggheads, "big-o" by pub frequenters) to characterise the efficiency (complexity) of an algorithm. It tells what is *the leading term* of the scaling dependency:

$$\text{if } \lim_{N\to\infty} T(N) = \text{const} \cdot f(N), \text{ then } T(N) = \mathcal{O}\left(f(N)\right),$$

when N becomes large enough. The dependence $T(N)$ of an algorithm,

- viewed as a problem, is referred to as *complexity*
- viewed as a solution, is referred as *efficiency*

Complexity classes

The function *f(N)* usually has one of the following types of behaviour:

- the **power** law:

$$f(N) = N^{\beta}, \text{ where } \beta \text{ is a number equals}$$

  to 2 for the Selection Sort, and to 2.376 for "smart" matrix multiplication, *etc*. Such *polynomial* problems are *tractable*.
- the **logarithm** law:

$$f(N) = (\log\ N)^{\beta}, \text{ where } \beta$$

  is 1 for a binary search (in a sorted array, in a binary search tree). These are *easy* problems.
- the **exponential** law:

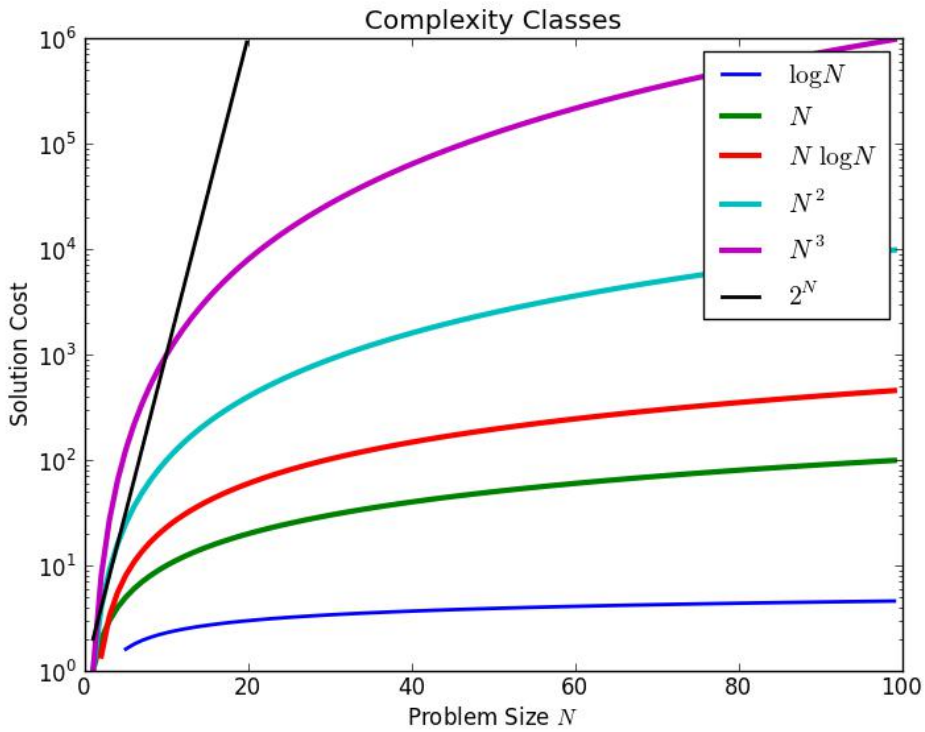
$$f(N) = 10^{\beta\ N}, \text{ where } \beta$$

  is 1/3 for the factorisation problem. Such problems with *exponential* behaviour are *hard* (intractable). Examples: the traveling salesman problem, set partitioning etc.
- A combination (as factors) of the above, *eg*,

$$f(N) = N \cdot \log_2 N, \text{ for the QuickSort algorithm}$$

$$f(N) = N^3 (\log\ N)^k, \text{ for the quantum Shor algorithm}$$

How complexity matters



Logarithmic scale in the ordinate axis!

O, Ω and Θ

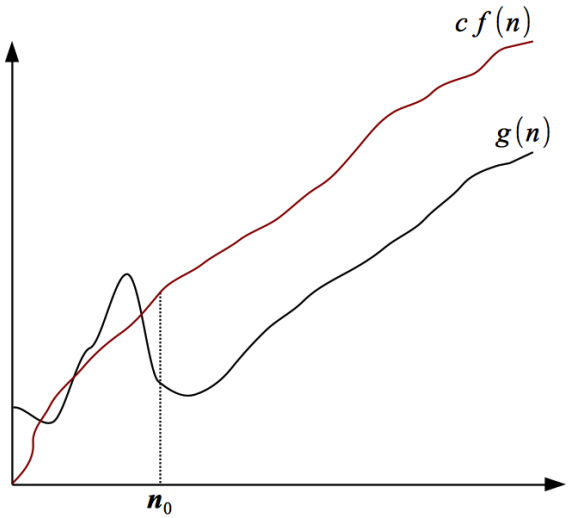
Some rigour

To proper understand the $\mathcal{O}$–notations one should remember that $\mathcal{O}(f(N))$ denotes not a single function, but a *class of functions* with the designated asymptotic behaviour. If we say that a function *g(N)* has an asymptotic of $\mathcal{O}(f(N))$, $g(N) = \mathcal{O}(f(N))$, it means not equality, but rather the inclusion of *g(N)* into the same class.

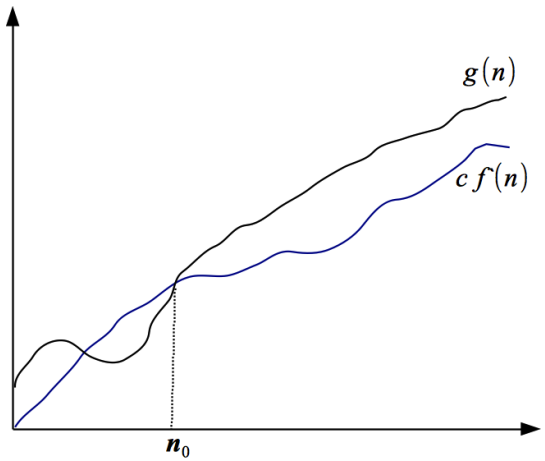
The $g(n) = \mathcal{O}(f(N))$ has meaning of the *upper-bound*, *ie* that its magnitude is **at most** as big as *const*×*f(N)* (when N is large). At most! but it could also be (substantially) smaller. Hence, an algorithm should not be dismissed outright if it is characterised by the efficiency of, say, $\mathcal{O}(N^2)$ (like in sorting), just because its actual performance can be better. Opposite to upper-bound is an lower-bound estimate, *ie* that the function magnitude is **at least** as large as *const*×*f(N)* (N is large). This asymptotic behaviour (different set) is notated with another symbol, *Omega*: $g(N) = \Omega(f(N))$. The Ω-properties of algorithms are harder to establish, but they are more valuable.

Even more valuable (and still harder to come by) are asymptotic with both lower- and upper bounds: that a function *g(N)* is such that for sufficiently large N, there exist two constants, C_1 and C_2 ($C_1 > C_2$), that $C_2 \cdot f(N) < g(N) < C_2 \cdot F(N)$. Such asymptotic sets are notated with the symbol Θ : $g(N) = \Theta(f(N))$.

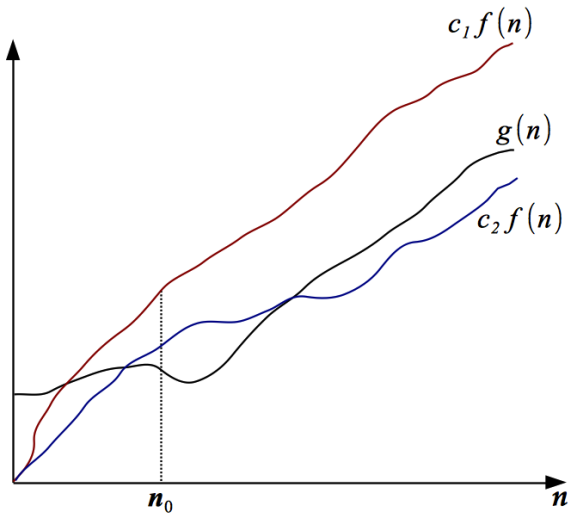
"Greek" Asymptotes



bounded from above
$g(n) = \mathcal{O}(f(n))$



bounded from below
$g(n) = \Omega(f(n))$



bounded in between
$g(n) = \Theta(f(n))$

One can proof that (something which is intuitively obvious):

$$g(n) = \mathcal{O}(f(n)) \;\; \textit{and} \;\; g(n) = \Omega(f(n)) \iff g(n) = \Theta(f(n))$$

The reason (not because "programmers are stupid") of why the $\mathcal{O}$-notations are used predominantly (when Ω would be more accurate) is explained in a classic Donald Knuth's paper "Big Omicron and Big Omega and Big Theta", in *SIGACT NEWS*, Apr–June 1967, pp.18–24.

Linear Search

Problem: Given *unsorted* array of N elements (`double`'s), find the smallest element in a *subarray*.

Algorithm: (trivial, correctness follows from the facts that each element of the array will undergo a comparison with the tentative result and comparisons of integers and real numbers are transitive, *ie* from a$> b$ and $b > c$ follows $a > c$).

1. Assume tentatively that the first number in the array is the smallest element
2. Compare this number with every element in succession
   - If one of these subsequent numbers is smaller, replace the tentative choice by this element
3. When all of the index locations have been exhausted, return the tentative result

```
/** @param data an array of doubles
  * @return value of smallest element  */
public static double smallest(double[] data, int lo, int hi) {
    assert (hi > lo);
    int iSmallest = lo; //Initial value
    for (int i=lo+1; i<=hi; i++)
        if (data[i] < data[iSmallest]) iSmallest = i;
        return data[iSmallest];
}
```

Complexity of this algorithm is $\mathcal{O}(N)$: we have to go from the first element to the very last one in search of the smallest (unsorted nature of the array is essential).

SelectionSort

1. In a loop which iterates a counter, `i`, over every index of an array in ascending order
2. At every step, do the following
   - Compute the `smallestInRange` (the index of the smallest element) between the present array index, `i`, and the maximum array index. Call this index `iSmallest`
   - Interchange the array elements at `i` and `iSmallest`

```
/** @param data an array of doubles */
public static void
        selectionSort(double[] data)
   for (int i=0; i<data.length-1; i++) {
      int iSmallest = SelectionSortTest.
            smallestInRange(data,
                            i,
                            data.length-1);
      // perform a swap if data[i] is smaller
      if (data[i] < data[iSmallest]) {
         double temp = data[i];
         data[i] = data[iSmallest];
         data[iSmallest] = temp;
      }
   }
}
```

How array changes "in-place"

- `step 1: 8.0 5.0 3.0 7.0`
- `step 2: 3.0 5.0 8.0 7.0`
- `step 3: 3.0 5.0 8.0 7.0`
- `step 4: 3.0 5.0 7.0 8.0`

Complexity

- number of iterations, $\sim N$
- `smallestInRange()` is called on average $\sim \frac{N}{2}$ times at each step
- which results into $\mathcal{O}(N^2)$
- for an *almost sorted array* it is $\mathcal{O}(N)$ (as the majority of swaps are not performed)

Algorithm correctness

Correctness and efficiency of algorithm implementations must be tested both theoretically (by examining the code), and experimentally (by running comprehensive *test harnesses* with all possible types of input data, and measuring the time-size dependence). The test harnesses (special testing programs — not necessarily written in Java — and input/output data) represent *assets* of the development as much as the code itself. Correctness is the primary goal of testing, but the efficiency can also be an important (*eg*, to check a theoretical conclusion regarding the algorithm complexity).

To check the **correctness** of our implementation of *SelectionSort* do the following (example SelectionSortTest.java):

1. Create an array and initialise it with N numbers (include cases `0` and `1`!)
2. Print the initial array, or run a routine to check that if it's ordered
3. Run *SelectionSort*
4. Print the sorted array, or run a routine to check that if it's ordered
5. Repeat the above for *many* different sizes and initialisations:
   - sorted arrays
   - unsorted arrays
   - almost sorted, reversed sorted
   - arrays with randomly chosen elements
   - boundary cases — 0 or 1 element long arrays, *very long* arrays

Algorithm efficiency

To check the **efficiency** one has to:

1. Work with large N (the upper limit is determined by the memory constraints)
2. Measure how the time taken by the sorting method scales with N
3. Consider "randomised" arrays (initialised by using a pseudo-random number generator) and limiting cases (fully and nearly sorted arrays, all and almost all equal elements etc). Often, the efficiency changes (deteriorates) in the limiting cases, *eg*, *QuickSort* (discussed later) for sorted array ($N \cdot \log_2 N \rightarrow N^2$), which may force to reconsider the algorithm implementation.

Time measurements

1. **OK:** use the Unix system *utility command* `time -p` when running the Java program (the full path is needed for some shells):

   ```
   /usr/bin/time -p java SelectionSortTest
   ```

   Of three times printed, "user time" gives the time in seconds taken by the program including "paging" interrupts.
2. **Better:** to call the `java.util.Date` class, and to read times in milliseconds. The overhead here is the creation and call to *Date* objects. Also, in a time-sharing system, there is no way to find the time for your particular process alone (In both pervious approaches, the overhead effect flattens out for large N).
3. **Best:** call the system clock via `System.currentTimeMillis()`, or create a utility class similar to `StopWatch.java` to include such calls for calculating elapsed times.

Example of Time Measurements

A measurement example program (written by C. Hortsmann years ago)

- SelectionSortWithTiming.java — the original sorting program with timing based on `System.currentTimeMillis()` calls
- ArrayUtil.java — a utility class to create random array and also perform element's swapping
- StopWatch.java — the helper class to set the time counting on/off

To study the performance of a particular selection sort implementation, one should run this program multiple times, with the length of input arrays changing from `0`, `1`, few and progressively larger and ultimately very large (how large?), using different "filling" for each case: random (several, with different random seed), sorted (both descended and ascended), all equal, and almost sorted. Why? Because, even the most accurate theoretical estimate will not tell you how different parts (search, writing temporary data and reading it etc) of execution will be carried out in a real CPU, its cache(s) and the main memory.

Bubble Sort and *Gap Sort*

These are two elegant algorithms for solving the sorting problem. Neither is as "naïvely intuitive" as the selection sort or the insertion sort in the next section. The gap sort is an amazingly fast variant of the bubble sort The essence of the *BubbleSort* algorithm:

1. Run an inner loop which repeatedly steps through every element in the array from left to right comparing adjacent elements, and if the left element is larger than the right element, then the elements are swapped. The (new) right element then becomes the left element of the next comparison and so on.
2. After one inner loop, the largest element is "floated" to the right hand side. The order of the other elements may have changed.
3. After the second inner loop, the second largest element has floated right to be the `data.length-2` element. The algorithm terminates when it makes one pass with no interchanges.

The gist of this (and other algorithms which try to improve the *SelectionSort* efficiency) is to minimise repetitive swaps. Yet, *BubbleSort* is still $\mathcal{O}(N^2)$ (demonstrate BubbleSortTest.java).

The analogous *GapSort* improves the efficiency, but the reason for this isn't well understood. The algorithm achieves an *intermediate* performance between a standard $\mathcal{O}(N^2)$ and $\mathcal{O}(N \cdot \log_2 N)$ (the actual asymptotic *fluctuates*). *GapSort* compares elements which are separated by a "gap" number of elements (rather than one element as in the bubble sort). The initial *gap* is taken pretty large — approximately 3/4 of the size of the array. It is reduced in size every outer iteration by a "magic" factor (1.3). Intuitively, the gapping is effective in reducing the number of unnecessary swaps (demo in GapSortTest.java).

Insertion Sort

A simplified algorithm is the insertion sort (inspired by card playing):

1. In a loop from left to right (`i = 0..(length-1)`), set a temporary variable `tmp = data[i]`
2. As we go along, the subarray on the left grows from `0` to `lenght-1` and *always* remains sorted
3. Insert `tmp` in to the appropriate place `k` in the left subarray, `k = 0..(i - 1)` in such a way that it remains sorted
4. Shift array items to the right when necessary. The left subarray is enlarged by 1.

The insertion can be done naively, or via *binary search* (because the array on left, in which the insertion is done, is *sorted*). The naïve version is demonstrated in InsertionSortTest.java.

The efficiency of *InsertionSort* is still $\mathcal{O}(N^2)$, for generic (random) inputs. But for *almost sorted* inputs, it becomes almost $\mathcal{O}(N)$. This property of *InsertionSort* is valuable, since the most efficient famous *QuickSort* (see below) suffers deterioration in such cases. By combining the two algorithms, an improved *QuickSort* can be implemented with much more robust performance.

"An algorithm must be seen to be believed" (D. Knuth) — **demo** SortingApplet.

A useful site with major sorting algorithms (animated like the above) including both time and space complexity analyses is Sorting Algorithm Animations.

Note on sorting an important version of sorting is a so called *stable sort*: when you must preserve the order of any pair of elements which are equal for the purpose of sorting. Stable sorting is more complex and *may* have worser performance.

Where to look for this topic in the textbook?

- Hortsmann's Core Java for the Impatient (not discussed)
- Oracle's Java Tutorial chapter Algorithms has only passing reference to $\mathcal{O}$-expression notation (but does discuss sorting and searching algorithms somewhat)
- To study algorithms properly, one should take a whole course and/or use of those bodice-ripper books:
  - "Introduction to Algorithms", by Thomas H. Cormen, Sharles E. Leiserson, Ronald L. Rivest and Clifford Stein, McGraw-Hill, 3d Ed, 2007
  - "Fundamentals of Algorithms", by Gilles Brassard and Paul Bratley, Prentice-Hall, 1996
  - "The Art of Computer Programming" 3rd Edition, by Donald Knuth, Addison-Wesley, 1997–1998; volumes:
    1. "Fundamental Algorithms"
    2. "Seminumerical Algorithms"
    3. "Sorting and Searching"