Data Structures & Algorithms [1]

Jakub Opršal
Sagnik Mukhopadhyay
Rajesh Chitnis
&
Ahmad Kamel (Dubai)

[1]Thanks to *Alan Sexton* and many staff from the School of Computer Science, Uni of Birmingham, UK

Programs = Algorithms + Data Structures

Data Structures efficiently organise data in computer memory.

Algorithms manipulate data structures to achieve a given goal.

In order for a program to terminate fast (or in time), it has to use *appropriate* data structures and *efficient* algorithms.

In this module we focus on:

- various data structures
- basic algorithms
- understanding the strengths and weaknesses of those, in terms of their time and space complexities

Learning Outcomes

After completing this module, you should be able to:

- Design and implement data structures and algorithms
- Argue that algorithms are correct, and derive time and space complexity measures for them
- Explain and apply data structures in solving programming problems
- Make informed choices between alternative data structures, algorithms and implementations, justifying choices on grounds such as computational efficiency

At the end of the term you should be able to decide which algorithms and data structures to choose, so that they are the best for the given task.

Assessment

Continuous assessment (20%)

Three summative assessments:

1. Theory assessment due in Week 3 (8%)
2. Coding assessment 1 due in Week 7 (6%)
3. Coding assessment 2 due in Week 10 (6%)

CA is designed to prepare you for the exam!

Exam (80%)

- 2 hour pen and paper exam taken in the summer exam period.
- Pen and paper. Calculators may be used provided they are not capable of being used to store alphabetical information
- Write algorithms using *pseudocode*, although you might be asked to write a few lines of *Java code*

Feedback and Support

Help sessions

The purpose of labs is to provide space and support for you to implement and practice algorithms and data structures you will learn about at the lecture.

All labs are in the CS building (UG04):

- Tuesday morning 11am–1pm and afternoon 2–4pm
- Friday afternoon 4–6pm.

Canvas discussions and FAQ

Feel free to ask questions on Canvas Discussions.

- We are monitoring Canvas Monday–Friday during working hours.
- Expect answers within the next working day.

Office Hours — see Canvas.

Complexity and Efficiency

Example: Compute Fibonacci numbers

Fibonacci numbers is a sequence of integers $f_0, f_1, f_2, \ldots$ starting with $f_0 = 0$, $f_1 = 1$, where the following terms are computed as the sum of previous two, i.e.,

$$f_{n+2} = f_{n+1} + f_n$$

The sequence begins:

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, \ldots$$

Goal. Write a function `long fib(int n)` that computes the n-th Fibonacci number!

Example (cont.): Two solutions

```
1  long fib_0 (int n) {
2    if (n < 2) {
3      return n;
4    } else {
5      return fib_0(n-1) + fib_0(n-2);
6    }
7  }
```

```
1  long fib_1 (int n) {
2    if (n < 1) return 0;
3
4    long[] fibs = new long[n+1];
5    fibs[0] = 0;
6    fibs[1] = 1;
7    for (int i = 2; i <= n; i++) {
8      fibs[i] = fibs[i-1] + fibs[i-2];
9    }
10   return fibs[n];
11 }
```

Which one is better??

Test the time performance of the two functions.

Run the code on inputs $n = 46$, $n = 47$, and $n = 92$ (93rd Fibonacci number does not fit in Java's long anymore), and measure the time required for execution.

How can we estimate the time it takes to execute a function?

- Measuring this in normal time units makes us dependent on the machine we run it on.
- We could measure in processor cycles, but we do not know optimisations made by the compiler, e.g., how many steps does the following code take?

```
1        int tmp = i;
2        i = j;
3        j = tmp;
```

  Some compilers will optimise this by swapping pointers.
- Instead we measure it in numbers of 'basic steps' (e.g. the number of additions or multiplications, the number of comparison operations, the number of memory accesses etc.) and accept that we will be *off by a constant factor*.

Time analysis of `fib_1`

How well does the code scale?

```
1  long fib_1 (int n) {
2    if (n < 1) return 0;
3
4    long[] fibs = new long[n+1];
5    fibs[0] = 0;
6    fibs[1] = 1;
7    for (int i = 2; i <= n; i++) {
8      fibs[i] = fibs[i-1] + fibs[i-2];
9    }
10   return fibs[n];
11 }
```

We can ignore what happens before and after the loop since the length of execution of this code does not depend on the input.

The loop is executed $n - 2$ times.

The actual time taken by the computation is $(An + B)$ seconds for some constants A and B:

- A is the time required for one execution of the loop,
- B is the time required for setup and the operations before/after the loop.

Since the the time taken is a linear function in n, we would say that `fib_1` is computed in linear time.

Time analysis of `fib_0`

How well does this code scale?

```
1  long fib_0 (int n) {
2    if (n < 2) {
3      return n;
4    } else {
5      return fib_0(n-1) + fib_0(n-2);
6    }
7  }
```

Assume that computing $fib_0(n)$ takes $t_0(n)$ nanoseconds. Then

$$t_0(n) = t_0(n-1) + t_0(n-2) + C$$

for all $n > 2$ for some constant C.

In particular, $t_0(n) \geq t_0(n-1) + t_0(n-2)$ which means that the time scales as quickly as Fibonacci numbers!

In fact, with a bit of math, you can show that $t_0(n) \geq 2^{n/2}$ for all big-enough n:

$$t_0(n) \geq t_0(n-1) + t_0(n-2) \geq t_0(n-2) + t_0(n-3) + t_0(n-2) \geq 2t_0(n-2)$$

The time more than doubles every two steps!

$t_0(n)$ is approximately $C \cdot (\frac{1+\sqrt{5}}{2})^n$, this is called *exponential time*.

Example (cont.): Two solutions

```
1  long fib_0 (int n) {
2    if (n < 2) {
3      return n;
4    } else {
5      return fib_0(n-1) + fib_0(n-2);
6    }
7  }
```

```
1  long fib_1 (int n) {
2    if (n < 1) return 0;
3
4    long[] fibs = new long[n+1];
5    fibs[0] = 0;
6    fibs[1] = 1;
7    for (int i = 2; i <= n; i++) {
8      fibs[i] = fibs[i-1] + fibs[i-2];
9    }
10   return fibs[n];
11 }
```

Which one is better?

Time complexity: `fib_0`'s exponential time is much worse than `fib_1`'s linear time.

Challenge! Find a better solution than `fib_1`! Better in any measure you like.

We discussed how to assess quality of algorithm based on its **time complexity**.

Time complexity is the number of basic operations (*steps*), which is a good approximation of the time required for execution of the algorithm. These computations are always approximate – we cannot compute the exact number of processor cycles/seconds required.

We have shown that two functions computing the same values can have vastly different complexities (2^n vs n).

big-O notation

Assymptotics

The precise number of steps is often too detailed to get a clear understanding of the performance of an algorithm:

- What if an algorithm does a comparison and a multiplication on every element of an array
  - Complexity is n steps, where a step is a comparison **AND** a multiplication
  - Complexity is $2n$ steps, where a step is a comparison **OR** a multiplication

The difference between an algorithm that has time complexity n and one that has $2n$ is small in relation to the difference between two algorithms that have respective complexities of n and n^2.

Similarly in an algorithm that has complexity $n^2 + n$, the n part only makes a small contribution.

Solution: simplify to headline complexity

Complexity in Terms of Input Size

Even if we know an algorithm's time performance (in units of steps) and space performance (in units of words of memory), we still do not yet have a way of capturing how that performance changes with different sizes of problems.

Solution: parametrize the performance by the size of the input:

- This algorithm takes $3N + 2$ steps on an input of size N
- This algorithm takes 2^N steps on an input of size N
- This algorithm uses N^3 words of memory on an input of size N

We want to measure how well does the algorithm scale with the input size.

Measuring steps up to a constant factor

The number of *steps* is measured up to a constant factor. This is again to have a measure that is independent on the machine: A different processor architecture can have different basic *operations* = steps.

- E.g., some processor have a single operation $\mathrm{ma}(x, y, z) = x \cdot y + z$ which performs floating point multiplication and addition in *one step* instead of 2.

It also simplifies the complexity calculations.

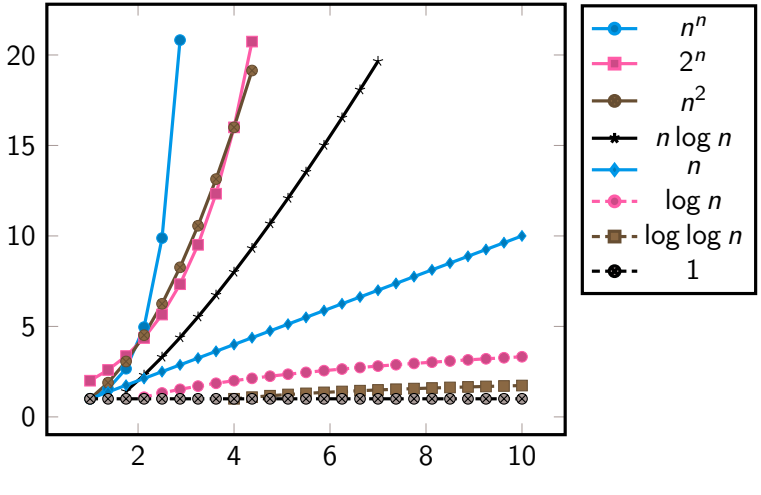
When giving an upper bound, we will give an upper bound of the form:

$$f(n) \leq Cg(n) \quad \text{for some constant } C > 0.$$

For example,

- $4n + 3 \leq Cn$ for some $C > 0$ (take $C = 7$),
- $n^3 + 2n^2 + n + 20 \leq Cn^3$ for some $C > 0$ (take $C = 24$),
- $5n^4 + 20n^3 - 19n^2 + 1001 \leq Cn^4$ for some $C > 0$ (take $C = 5 + 20 + 19 + 1001$),
- $2^{n+12} = 2^{12} \cdot 2^n \leq C2^n$ for some $C > 0$ (take $C = 2^{12}$).

Comparing Functions



https://www.geogebra.org/m/zxwctk7y

Big O Notation

Usually, instead of writing $f(n) \leq Cg(n)$ for some $C > 0$, we use the big O notation.

Definition
We write $f(n) \in O(g(n))$ if

$$\text{there exist constants } C, n_0 > 0 \text{ such that for all } n \geq n_0, \quad |f(n)| \leq |Cg(n)|$$

- Formally, $O(g(n))$ is a *set of functions*. The set contains all functions that do not grow at a faster rate than g.
- Informally, we may treat the expression $f(n) \in O(g(n))$ as comparison of functions '$f(n) \preccurlyeq g(n)$' meaning 'f does not grow at a quicker rate than g'.

Idea: f does not grow at a faster rate than g as n increases. It might grow at the same rate or it might grow at a slower rate.

Big O and Abuse of Notation

It is common to write $f(n) = O(g(n))$ instead of $f \in O(g(n))$ which is an abuse of notation!

- "$3n^2 + 4 = O(n^2)$" is just a shorthand for

  "$3n^2 + 4 \leq Cn^2$ for all $n \geq n_0$ for some $n_0, C > 0$".

- "$O(n^2) = 3n^2 + 4$" does not have any meaning!
  - If it did, we could do nasty things like: $3 = O(1) = 2$ hence $3 = 2$ **WRONG!!**

Big O Notation

Examples:

- $2n \in O(n)$?
  - *TRUE*: choose C to be 3
- $2n + 100 \in O(n)$?
  - *TRUE*: choose C to be 1000
- $n \in O(1)$?
  - *FALSE*: no value of C is large enough so that $n \leq C$
- $3n^2 + n \in O(n^2)$?
  - *TRUE*: $3n^2 + n \leq 3n^2 + n^2 \leq 4n^2$ choose C to be 5
- $3n^2 + n \in O(n^3)$?
  - *TRUE*: $3n^2 + n \leq 4n^2 \leq 4n^3$ choose C to be 5
- $3n^2 + n \in O(n)$?
  - *FALSE*: no value of C is large enough so that $3n^2 + n \leq Cn$

Intermezzo: Logarithms

Recall, the definition of the logarithmic function:

$$\log_b a = x \iff b^x = a.$$

The $\log_b$ function is called *logarithm of base b*, e.g., $\log_2 8 = 3$

Recall:

$$\log_b x + \log_b y = \log_b(x \cdot y) \qquad \log_a x = \frac{\log_b x}{\log_b a} = (\log_a b) \cdot (\log_b x) \qquad y^x = 2^{x \log_2 y}$$

We will use base $b = 2$. **In everything that follows,** $\log n$ **means** $\log_2 n$**.**

You can think of this logarithm as follows:

- The length of n in binary is $\approx \log n$;
- To store a value between $0 \ldots n-1$ you need $\approx \log n$ bits.

Computations

- if $f \leq Cg$ for some $C > 0$ then $f + g \leq (C + 1)g$ for some $C > 0$, e.g., $100n^2 + 454n \log n \leq Cn^2$ for some $C > 0$.
- for all $n \geq 4$, we have

$$1 \leq \log n \leq n \leq n \log n \leq n^2 \leq 2^n \leq n^n$$

Therefore

$$\begin{aligned} 1 &\in O(\log n) \\ \log(n) &\in O(n) \\ &\vdots \\ 2^n &\in O(n^n). \end{aligned}$$

Thus, for example:

$$3n^n + 2^n + 52n^3 + 100n^2 + 454n \log n + 24n + 12 \log n + 43 \leq C \cdot n^n \in O(n^n)$$

Which is true?

$f(n) \in O(g(n))$ or $g(n) \in O(f(n))$ or neither?

Which is true? Question 1

Option A

$$200n + 3 \in O\left(2^n\right)$$

or

Option B

$$2^n \in O\left(200n + 3\right)$$

Which is true? Question 2

Option A

$$n^3 + 20n^2 - n + 1000 \in O\left(200n^2 + n - 1\right)$$

or

Option B

$$200n^2 + n - 1 \in O\left(n^3 + 20n^2 - n + 1000\right)$$

Which is true? Question 3

Option A

$$n^2 \in O\left(2^n\right)$$

or

Option B

$$2^n \in O\left(n^2\right)$$

Which is true? Question 4

Option A

$$(\log_2 n)^2 \in O(n)$$

or

Option B

$$n \in O\left((\log_2 n)^2\right)$$

Which is true? Question 5

Option A

$$\log_2 n \in O\left(\log_{10} n\right)$$

or

Option B

$$\log_{10} n \in O\left(\log_2 n\right)$$

Which is true? Question 6

Option A

$$2^n \in O\left(2^{n/2}\right)$$

or

Option B

$$2^{n/2} \in O\left(2^n\right)$$

Which is true? Question 7

Option A

$$1729n^2 - 13n + 3 \in O\left(n^2 + 1\right)$$

or

Option B

$$n^2 + 1 \in O\left(1729n^2 - 13n + 3\right)$$

Which is true? Question 8

Option A

$$m \log n + n \in O\left(m \log n\right)$$

or

Option B

$$m \log n \in O\left(m \log n + n\right)$$

Which is true? Question 9

Option A

$$m + n \in O(m \log n)$$

or

Option B

$$m \log n \in O(m + n)$$

Which is true?

$f(n) \in O(g(n))$ or $g(n) \in O(f(n))$ or neither?

$f(n) = 200n + 3$	$g(n) = 2^n$	(1)
$f(n) = n^3 + 20n^2 - n + 1000$	$g(n) = 200n^2 + n - 1$	(2)
$f(n) = n^2$	$g(n) = 2^n$	(3)
$f(n) = \log_2^2 n$	$g(n) = n$	(4)
$f(n) = \log_2 n$	$g(n) = \log_{10} n$	(5)
$f(n) = 2^n$	$g(n) = 2^{n/2}$	(6)
$f(n) = 1729n^2 - 13n + 3$	$g(n) = n^2 + 1$	(7)
$f(m, n) = m \log n + n$	$g(m, n) = m \log n$	(8)
$f(m, n) = m + n$	$g(m, n) = m \log n$	(9)

Big O and Friends

So far we have looked at Big O as a way to identify an upper bound on complexity of an algorithm, and that is what we will be most concerned with. But there are others:

- **Big O** $f(n) = O(g(n))$: g is an upper bound on how fast f grows as n increases.
- **Omega** $f(n) = \Omega(g(n))$: a lower bound on how fast f grows as n increases (the lower bound equivalent of big O).
- **Theta** $f(n) = \Theta(g(n))$: Both upper and lower bounds, which are given by the same function, except with different constant factors, i.e., f and g grow at the same rate.

Definitions:

- $f \in O(g)$ if there exist $C, n_0 > 0$ such that for all $n > n_0$:

$$|f(n)| \leq C|g(n)|$$

- $f \in \Omega(g)$ if there exist $C, n_0 > 0$ such that for all $n > n_0$:

$$C|g(n)| \leq |f(n)|$$

- $f \in \Theta(g)$ if there exist $C_0, C_1, n_0 > 0$ such that for all $n > n_0$:

$$C_0|g(n)| \leq |f(n)| \leq C_1|g(n)|$$

Big-O, Omega, and Theta

Remember:

- $f \in \Omega(g)$ if and only if $g \in O(f)$.
- to show that $f \in \Theta(g)$ show both $f \in \Omega(g)$ and $f \in O(g)$!

Examples:

- $n \log n \in O(n^3)$ and $n^3 \in \Omega(n \log n)$.
- $2n^2 + n = O(n^2)$.
- $2n^2 + n = \Omega(n^2)$.
- the two above imply that $2n^2 + n = \Theta(n^2)$.
- $n^2 + 2n + 1 = \Theta(n^2)$, but $n^2 + 2n + 1 \notin \Theta(n^3)$!

We made measuring *time and space complexity* more formal by measuring it as a function of the input size with precision up to a constant factor.

Today, we were concerned with *worst case complexity* which a *complexity upper bound*, and we have introduced *the big O notation* for such bounds. Warning: big O is an abuse of notation!

Let's play a game!

I am thinking of a number $n \in \mathbb{N}$ between 0 and 127:

$$0 \leq n \leq 127.$$

Your goal is to guess the number!

To make it easier, every time you guess incorrectly, I will tell you whether my number is smaller or bigger than your guess.

Worst case complexity

Complexity Upper Bounds = Worst Case Complexity

An algorithm can take different amount of time (steps) on two inputs of the same size n.

We are usually interested in an upper bound on the complexity of an algorithm, i.e., we want to make sure that the algorithm *runs in a specified time no matter which input is given*.

This is also called worst case complexity since the maximum time is equal to the time the algorithm takes on the *worst case input*.

Example: Linear search

To find a position of a value in an array, we can run this code:

```
1  int search (int[] array, int x) {
2    int n = array.length;
3    int i = 0;
4    while (i < n) {
5      if (array[i] == x) {
6        return i;
7      } else {
8        i++;
9      }
10   }
11   return -1; // value not found
12 }
```

Input length: $n + 1$ (the length of the array and `x`) $\approx n$.

Worst case time complexity:
when `x` does not appear in `array`

$$t(n) = 2 + n + 1 = n + 3 \in O(n).$$

Note that the time is in $O(n)$ for all inputs of length n!

It is equally valid to say that the time complexity is $O(n^2)$, but this is a worse upper bound than $O(n)$. We always look for the best upper bound!

Search in a Sorted Array

Previously, you have seen how to lookup a value `x` in an array `array` in time $O(n)$, where $n =$ `arr.length`.

Can we do better in case the array is sorted?

(The values in `array` are ascending.)

Binary Search

Searching `x` in a sorted array `arr`:

1. Compare `x` and `arr[arr.length / 2]`.
2. If `x` is bigger, recursively search `arr` on positions `(arr.length / 2) + 1`, ..., `arr.length - 1`.
3. Otherwise, recursively search `arr` on positions `0`, ..., `arr.length / 2`
4. We continue like this until we are left with only one element in the array. Then, return whether this element equals `x`.

The length of the array we search through reduces by one half in every step and we continue until the length is 1.

Binary Search

```
1: function BinarySearch(array a, value x)
2:     l := 0, r := length(a) − 1
3:     mid := (r + l)/2
4:     while r > l do
5:         if a[mid] > x then
6:             l := mid + 1
7:         else
8:             r := mid
9:     return a[l]
```

For simplicity assume that the length of a is $n = 2^k$

$\Longrightarrow$ the number of steps is (at most) $Ck = C \log_2 n$ for some $C > 0$.

Hence the time complexity is $O(\log n)$.

Challenge! Prove that you cannot do better!

Pseudocode

Pseudocode (like the one on the previous slide) is a more precise description of an algorithm than english, but less precise than code.

It can be used as a baseline for implementing the algorithm in many programming languages, e.g., *Java*, *Python*, *C++*, *Rust*, *Javascript*.

There are no strict rules on how to write pseudocode, but here are some guidelines:

1. Include all loops, if-statements, and generally keep the same structure as code.
2. Clearly state inputs and outputs (`return` statement).
3. State the type of variables.
4. Omit irrelevant implementation details.
5. Pseudocode should give anyone reading it a good idea of how to implement it, and should be precise-enough so that we can, for example, judge the complexity of the code.

During the term, we will see many examples with various level of detail.

Different kinds of complexities

Worst Case complexity

= the worst complexity over all *possible inputs/situations* (complexity upper bound)

Average complexity

= average complexity over all *random choices*
(in case of *randomised algorithms*, or if we know the likelihood of each of the inputs!)

Amortized complexity

= average time taken over a sequence of *consecutive* operations
(useful for measuring performance of data structures)