Data Structures and Algorithms [1]

Jakub Opršal
Sagnik Mukhopadhyay
Ben McKanna
&
Anis Zarrad (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 and Feedback

- Continuous assessment (20%): Two summative assessments worth 10% each due in Weeks 7 and 11.
- Exam (80%): One 2 hour exam taken in the summer exam period.

Labs

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.

Each student will be assigned a 1-hour slot, and we ask you to come during this time — if you come to a different session we cannot guarantee a space for you, and you might be asked to leave. It is not expected from you that you will spend 6 hours in a lab!

All labs are in the CS building:

- Tuesday afternoon 4–7pm (UG04)
- Wednesday morning 9–11am (UG04) and 11am–12pm (LG04)

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 `int fib(int n)` that computes the *n*-th Fibonacci number!

Example (cont.): Two solutions

```java
int fib_0 (int n) {
  if (n < 2) {
    return n;
  } else {
    return fib_0(n-1) + fib_0(n-2);
  }
}
```

```java
int fib_1 (int n) {
  if (n < 1) return 0;

  int[] fibs = new int[n+1];
  fibs[0] = 0;
  fibs[1] = 1;
  for (int i = 2; i <= n; i++) {
    fibs[i] = fibs[i-1] + fibs[i-2];
  }
  return fibs[n];
}
```

Which one is better??

Time and Space Performance

There are 2 *dimensions* of performance we might be interested in:

- Time: How long it takes to run the algorithm:
  - Measuring this in normal time units makes us dependent on the machine we run it on.
  - Instead measure it in numbers of steps: e.g. the number of additions or multiplications, the number of comparison operations, the number of memory accesses etc.
- Space: How much memory it requires:
  - Different algorithms to accomplish the same result might use different amounts of memory. For example:
    - One takes 1,000,000 steps and require only 2 integer variables
    - Another take 20,000 steps but requires a list of length 1,000

To choose between algorithms, we need to understand both its time and space performance.

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 a list:
  - 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

Time and space analysis of `fib_1`

```java
int fib_1 (int n) {
  if (n < 1) return 0;

  int[] fibs = new int[n+1];
  fibs[0] = 0;
  fibs[1] = 1;
  for (int i = 2; i <= n; i++) {
    fibs[i] = fibs[i-1] + fibs[i-2];
  }
  return fibs[n];
}
```

- 4 operation before the loop.
- $n - 2$ iterations inside the loop.
- 1 to return the value.

Altogether:

$$t_1(n) = 4 + (n - 2) + 1 = n + 3 \leq 4n$$

Space: $s_1(n) = n + 1 \leq 2n$.

Since the upper bound is linear in n, we would say that `fib_1` is computed in linear time and space.

Time analysis of `fib_0`

```
int fib_0 (int n) {
  if (n < 2) {
    return n;
  } else {
    return fib_0(n-1) + fib_0(n-2);
  }
}
```

- assume that computing $\mathit{fib}_0(n)$ takes $t_0(n)$ time, and is monotone, i.e., $t_0(n+1) \geq t_0(n)$ for all n.
- we have $t_0(0) = t_0(1) = 1$, and
- $t_0(n) = t_0(n-1) + t_0(n-2) + 1$ for $n > 1$.

Therefore

$$\begin{aligned} t_0(n) &\leq 2t_0(n-1) + 1 = 2 \cdot (2t_0(n-2) + 1) + 1 = \dots \\ &= 2 \cdot (2 \cdots (2 \cdots (2+1) \cdots + 1) = 2^n + 2^{n-1} + 2^{n-2} + \cdots + 1 \leq 2^{n+1} \end{aligned}$$

Altogether, we get an upper bound on time $t(n) \leq 2 \cdot 2^n$.

We cannot get much better, in fact we can show $t(n) \geq 2^{n/2}$.

In any case $t_0(n) \gg t_1(n)$ for *big-enough* n.

Example (cont.): Two solutions

```
int fib_0 (int n) {
  if (n < 2) {
    return n;
  } else {
    return fib_0(n-1) + fib_0(n-2);
  }
}
```

```
int fib_1 (int n) {
  if (n < 1) return 0;

  int[] fibs = new int[n+1];
  fibs[0] = 0;
  fibs[1] = 1;
  for (int i = 2; i <= n; i++) {
    fibs[i] = fibs[i-1] + fibs[i-2];
  }
  return fibs[n];
}
```

Which one is better?

Time complexity: `fib_0` $t_0(n) \leq 2 \cdot 2^n$ is worse `fib_1` $t_1(n) \leq 4n$
Space complexity: `fib_1` $s_0(n) = ?$ vs. `fib_1` $s_1(n) \leq 2n$.

Challenge! Find a better solution than `fib_1`! Better time and/or better space.

We discussed how to assess quality of code/algorithm.

We introduced two measures of complexity of an algorithms: **time and space complexity**. Time complexity is the number of basic operations (*steps*), while space complexity is the amount of *memory* the algorithm uses.

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

Previously on DS&A. . .

```
int fib_0 (int n) {
  if (n < 2) {
    return n;
  } else {
    return fib_0(n-1) + fib_0(n-2);
  }
}
```

Time: $t(n) \leq C2^n$ for some $C > 0$.

Space: TBD.

```
int fib_1 (int n) {
  if (n < 1) return 0;

  int[] fibs = new int[n+1];
  fibs[0] = 0;
  fibs[1] = 1;
  for (int i = 2; i <= n; i++) {
    fibs[i] = fibs[i-1] + fibs[i-2];
  }
  return fibs[n];
}
```

Time: $t(n) \leq Cn$ for some $C > 0$.

Space: $s(n) \leq Cn$ for some $C > 0$.

Unlike in this example, usually we measure complexity in the length of the input!

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.

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*.

There are other complexity measures, some of which we will discuss later:

- *average case complexity*,
- *amortised complexity*,
- *complexity lower bounds*, etc.

Example: Linear search

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

```
int search (int[] array, int x) {
  int n = array.length;
  int i = 0;
  while (i < n) {
    if (array[i] == x) {
      return i;
    } else {
      i++;
    }
  }
  return -1; // value not found
}
```

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

Worst case time complexity:

when `x` does not appear in `array`

$$t(n) = 2 + n + 1 = n + 3 \leq 4n.$$

Space complexity:

negligible.

Measuring steps/memory 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.

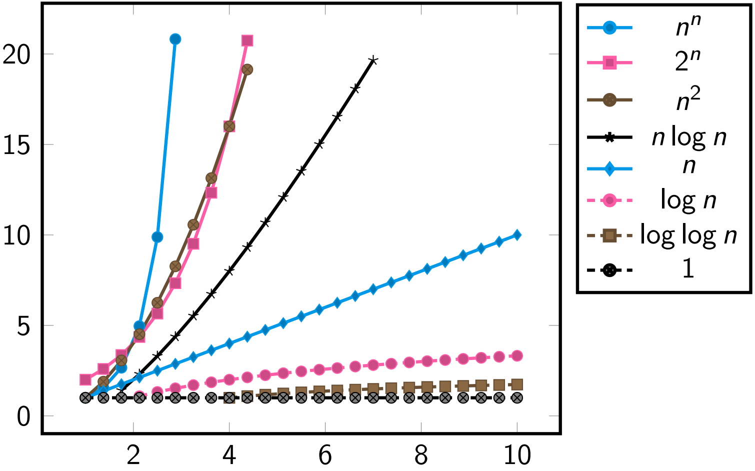
When giving an upper bound, we will give an upper bound of the form:

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

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

  $$\text{“}3n^2 + 4 \leq Cn^2 \text{ for all } n \geq n_0 \text{ for some } n_0, C > 0\text{”}.$$

- "$O(n^2) = 3n^2 + 4$" 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 $3n^2 + n \leq Cn$

Complexity 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:

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

for $C = 3 + 52 + 100 + 454 + 24 + 12 + 43$ and all $n \geq 4$.

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.

Math 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*.

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.

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.

For simplicity assume that the length of `arr` is $n = 2^k$
$\implies$ the number of steps is (at most) $Ck = C \log n$ for some $C > 0$, which is $O(\log_2 n)$.

Challenge! Prove that you cannot do better!

Binary Search Implementation

```
int binary_search(int[] arr, int x) {
  int from = 0;
  int to = arr.length;
  int mid;
  while (from < to) {
    mid = (from + to) / 2;
    if (x == arr[mid]) {
      return mid;
    } else if (x < arr[mid]) {
      to = mid;
    } else {    // x > arr[mid]
      from = mid + 1;
    }
  }
  return -1;    // value not found
}
```

Amortized Complexity

Different kinds of complexities

Worst Case complexity

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

Average Case complexity

= average complexity over all *possible inputs/situations*
(we need to know the likelihood of each of the input!)

Amortized complexity

= average time taken over a sequence of *consecutive* operations

We will often see algorithms where the worst case complexity is much more than the average case complexity. It depends on the usage, if you're processing a lot of data, occasional bad performance might be OK. However, if we are serving a customer and (in the worst case) she has to wait a long time, that's not good for the company's reputation.

In average-, worst- or best-case complexities, we are concerned with the performance of **one** (independent) operation. The amortized complexity is different: we are thinking about the average time complexity among a number of **successive operations**.

The idea is that, sometimes, you deliberately put extra effort in some operations, in order to speed up subsequent operations. For example, you might spend some time cleaning up or reorganizing of your data structure in order to improve the speed of the coming operations.

Whenever you are reorganizing your data structure, it might slow you down now but you'll benefit from this later (and hopefully many times).

Example: Amortized car cost

Oil consumption: 8 litres per 100 miles

Price of 1 litre is £1.20

$\Longrightarrow$ £9.60 per 100 miles

Extra expenses

- new tyres £320 every 70 000 miles
- new brakes £250 every 30 000 miles
- fix gearbox £300 every 130 000 miles
- fix clutch £406 every 100 000 miles

$\Longrightarrow$ extra £1.93 per 100 miles

$\Longrightarrow$ £11.53 is **amortized cost** per 100 miles

($\star$ those numbers are somewhat made up!)

Example: Sequence of searches in an array

Assume that you want to check whether *many* different values are present in an array. For simplicity, we assume that we don't need to know the index — we only need to know whether they are present or not. Say m lookups in an array of size n.

Complexity using linear search m times:

Each lookup takes Cn steps (for some $C > 0$), there are m lookups and therefore:

$$\begin{aligned} \text{total time} &= m \cdot Cn = C \cdot nm \\ \text{average time per lookup} &= Cn = O(n) \end{aligned}$$

Example: Sequence of searches in an array

What if we sort the array first?

Complexity using binary search by sorting the array first:

- Assume we can sort the array in time $t_{\text{sorting}}(n)$.
- First lookup takes $t_{\text{sorting}}(n) + C \log n$ time.
- Every successive lookup takes $C \log n$ time.

$$\begin{aligned} \text{total time} &= t_{\text{sorting}}(n) + m \cdot C \log n = t_{\text{sorting}}(n) + Cm \log n \\ \text{average time per lookup} &= \frac{t_{\text{sorting}}(n) + Cm \log n}{m} = \frac{t_{\text{sorting}}(n)}{m} + C \log n \end{aligned}$$

The first term in the average time vanishes as m increases.
If $m \geq C' \cdot t_{\text{sorting}}(n) / \log n$, it becomes at most $\log n$, and hence the average time will be $(1/C' + C) \log n = C'' \cdot \log n$.

For large-enough m, the **amortized time complexity** is $C'' \cdot \log n = O(\log n)$.

We recalled some basic properties of logarithms and made an agreement to use $\log n$ for the *base 2 logarithm*.

We have learned a more efficient way to lookup a value in a sorted array called *binary search*. The key idea is to halve the search space with every step. It takes $C \log n$ steps as opposed to Cn steps of linear search.

We discussed *amortised complexity*, and gave an example of an amortised algorithm for looking up several values in a fixed array by sorting first, and then using binary search.

Next: Sorting!