CSCM12: software concepts and efficiency
Algorithms and their complexity

Arno Pauly (for Cécilia Pradic)
Swansea University, 01/02/2024

Recommended reading after this lecture

- **Chapter 3** "Characterizing Running Times"
  of *Introduction to Algorithms* (4th ed., 2011) by Cormen et. al
- **Chapter 2** "Principles of Algorithm Analysis"
  of *Algorithms in Java* (3rd ed., 2004) by Sedgewick
  No need to look at the "Basic Recurrences" section for now

What is an algorithm?

Definition (Vagueish)

An algorithm is comprised of unambiguous instructions for carrying out a calculation.

What is an algorithm?

Definition (Vagueish)

An algorithm is comprised of unambiguous instructions for carrying out a calculation.

We might deal with numbers, words or more complicated objects.

When are two algorithms the same?

- Asking when two algorithms are the same is a bit distracting.

When are two algorithms the same?

- Asking when two algorithms are the same is a bit distracting.
- But we can certainly write the same algorithm in different programming languages.

When are two algorithms the same?

- Asking when two algorithms are the same is a bit distracting.
- But we can certainly write the same algorithm in different programming languages.
- Figuring out the algorithm you want to use comes before coding.

Algorithm versus specification

The specification states **what** we want to compute, the algorithm states **how** we are computing it.

On pseudo-code

- Programming languages are how we explain algorithms to computers.

On pseudo-code

- Programming languages are how we explain algorithms to computers.
- They are often less convenient when communicating with humans.

On pseudo-code

- Programming languages are how we explain algorithms to computers.
- They are often less convenient when communicating with humans.
- Pseudocode is a way to communicate algorithms to humans in a style similar to programming languages.

On pseudo-code

- Programming languages are how we explain algorithms to computers.
- They are often less convenient when communicating with humans.
- Pseudocode is a way to communicate algorithms to humans in a style similar to programming languages.
- You can get very far in exploring algorithms with pen and paper alone.

One running example

An algorithmic problem

Input: An array A of size n and some integer x

Output: An index i such that $A[i] = x$ or -1 if there is none

Solution #1

```
FindIndex(A, x)
    res ← −1
    n ← size of A
    for i from 0 to n − 1 do
        if A[i] = x then
            res ← i
    return res
```

Running the first solution

Let us try to run this step-by-step!

`FindIndex`(A, x)

- $res \leftarrow -1$
- $n \leftarrow$ size of A
- **for** i ***from*** 0 ***to*** $n-1$ **do**
  - **if** $A[i] = x$ **then**
    - $res \leftarrow i$
- **return** res

- $A = [2, 4, 7, 7, 10, 15]$, $x = 7$

Running the first solution

Let us try to run this step-by-step!

FindIndex(A, x)

```
res ← −1
n ← size of A
for i from 0 to n − 1 do
    if A[i] = x then
        res ← i
return res
```

- $A = [2, 4, 7, 7, 10, 15]$, $x = 7$
- $A = [2, 4, 7, 7, 10, 15]$, $x = 11$

Alternative solution 1

Solution #2

```
FindIndex2(A, x)
    res ← −1
    n ← size of A
    for i from n − 1 down to 0 do
        if A[i] = x then
            res ← i
    return res
```

- Solves the same problem
- Different outputs on our first sample input
- (Roughly the same complexity)

Alternative solution 2

Solution #3

```
FindIndex3(A, x)
    res ← −1
    n ← size of A
    i ← 0
    while res = −1 and i < n do
        if A[i] = x then
            res ← i
        Increment i
    return res
```

- Sometimes more efficient
- But is it significant in practice?

A more precise problem and another solution

A more precise algorithmic problem

Input: A **sorted** array A of size n and some (say, integer) x

Output: An index i such that $A[i] = x$ or -1 if there is none

- The previous solutions work, but...

A more efficient solution for sorted inputs

```
FindIndexDicho(A, x)
    start ← 0
    end ← size of A
    while start < end do
        mid ← ⌈(end+start)/2⌉
        if A[mid] ≤ x then
            start ← mid
        else
            end ← mid

    if A[start] = x then
        return start
    else
        return -1
```

Consideration of efficiency

Given an algorithmic problem:

- Is there an algorithm that solves it? If so is it:
  - feasible? (usable in practice)
  - efficient?
  - optimal?

Given an algorithm:

- How efficient is it?
- Is there a more method of getting the same results?

Rules of thumb for measuring efficiency

- *Typically*, (time) complexity mostly depends on the **size** of the input

$\rightarrow$ we typically express the time complexity as a function "size $\mapsto$ time"

$$x \longmapsto \text{time to compute } \mathcal{A}(x)$$

$$\begin{tikzcd}
\text{Input} \arrow[rr, "c"] \arrow[d, "\text{size}"'] \arrow[drr, phantom, "\leq"] & & \mathbb{N} \\
\mathbb{N} \arrow[urr, bend right, "\underbrace{k \mapsto \max\{c(x) \mid \text{size}(x) = k\}}_{\text{complexity}}"'] & &
\end{tikzcd}$$

Note the $\leq$: typically we want the **worst-case complexity** for inputs of a given size

- best-case: not very interesting
- average: can be interesting, typically harder to compute though :)

Computing time complexity

- Can be roughly be done step-by-step.
- Essentially, each piece of a program can be regarded as a mathematical function

$$\overbrace{\text{State}}^{\text{(initial) value of variables/memory}} \longrightarrow \text{State} \times \underbrace{\mathbb{N}}_{\text{time taken to compute the step}}$$

- Essentially: basic arithmetic operations, assignments: cost ~ 1, array allocation $\sim$ size of the array, loop $\sim$ sum of the complexities, ...

$\rightarrow$ roughly the number of steps in step-wise execution we've done

The notion of space complexity

There is a notion of **space** complexity

- Essentially, assign a size to State and compute the maximal size that occurs in an execution
- Unless you are doing big data or embedded system, this is not typically a limiting factor

  (RAM is cheap)
- In most scenarii, bounded by time complexity

Accurate complexity?

The "time complexity function" we defined might not be *completely* accurate

In practice

- hardware/compiler-dependent behaviors
- not so reliable hardware optimisations

(predictive branching, prefetch, cache misses)

→ We had to make compromises

Accurate complexity?

The "time complexity function" we defined might not be *completely* accurate

In practice

- hardware/compiler-dependent behaviors
- not so reliable hardware optimisations

(predictive branching, prefetch, cache misses)

→ We had to make compromises

However, gives reasonable bounds/estimate

- up to a **constant factor**
- for **large inputs** (and that's we care about!)