5SENG003W - Algorithms, Week 1

Dr. Klaus Draeger

Introduction

Algorithms are everywhere

- Web search
- Computer Graphics
- Cryptography
- Image recognition
- Security
- Recommendations
- ...

Some main topics

- We will see ways of **analysing** and **designing** algorithms
  - Big-O notation
  - Some important complexity classes (logarithmic, linear, quadratic, exponential, ...)
  - How to determine them empirically (doubling hypothesis)
  - Strategies (Greedy, Divide-and-Conquer)
- We will also focus on the relationship between **algorithms** and **data structures**
  - Linear vs non-linear structures
  - Indexed vs linked structures

Some logistics

- In-person lectures
  - Live lecture recordings available on blackboard later
- Tutorials in labs
- One in-class test, one coursework
  - Worth 50% each
  - Need to score at least 30 in each and at least 40 on average

What is an algorithm?

- General idea: a set of **instructions** to solve a **problem**
  - Find a solution (search in a data set, solve equations, . . .)
  - Find an **optimal** solution (shortest path, minimal solution of equations, . . .)
  - Transform data (sort a data set, multiply matrices, . . .)
- Instructions include
  - Atomic operations such as assignments
  - Decisions (branching or loops)
- Can be represented in different ways, including **pseudocode** and **flowcharts**

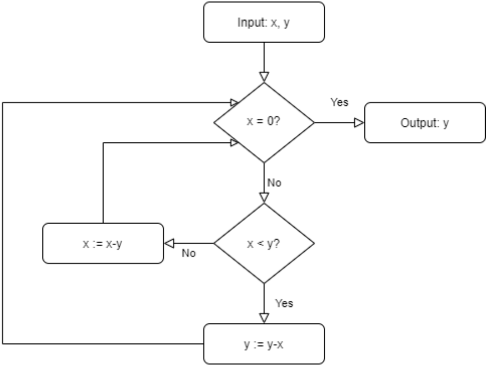
Pseudocode

- Similar to a program, but without irrelevant details
- Example: one of the earliest known algorithms is **Euclid's algorithm** for computing greatest common divisors

```
Input: x, y
Output: gcd of x and y
while x > 0
    if x < y
        y := y - x
    else
        x := x - y
output y
```

Flowcharts

- Diagrams representing the flow of an algorithm
- Boxes represent instructions and decisions
- A flowchart for Euclid's algorithm could look like this:



Analysis of algorithms

- A central question about algorithms is how much **time** they take to solve a problem
  - Can also ask about other resources such as **memory** or **bandwidth** usage
- By "time" we can mean
  - Actual (milli-, micro-, ...) seconds
    - Directly measurable
    - But depends on implementation details, hardware etc
  - Number of atomic operations
    - Can be determined by analysing the **(pseudo)code structure**
    - More advanced tools like **Master's Theorem** for recursive algorithms
- Not straightforward: on which input?
  - Many different ones, of arbitrary sizes

Analysis of algorithms

- Algorithmic analysis
  - Considers the **worst case** (i.e. maximal) time required for any input size n
    - **average case** and **best case** complexity are also sometimes used
  - Focuses on the **order of growth**: for inputs of size n, does the time required grow like $\log n$? n^2? 2^n?
- A related (harder) question is the complexity of the problem itself:
  What is the best we can hope for from **any** algorithm?

Orders of growth

- Suppose we have a mathematical function in the variable n, like $f(n) = 5 \cdot 2^n + 3 \cdot n^4 + n \cdot \log n$
  - This could be the number of steps some algorithm needs on an input of size n
- Only the **fastest-growing** term is relevant
  - Among powers of n: the one with the highest exponent
  - 2^n grows faster than any power, $\log n$ more slowly
  - The hierarchy looks like this:
    $1 < \log n < n < n \cdot \log n < n^2 < \ldots < 2^n < n \cdot 2^n < \ldots$
- Constant factors are irrelevant: for the function f, the relevant term is 2^n, not $5 \cdot 2^n$
- So the order of growth of f is 2^n

Big-O and Theta notation

- In order to describe the complexity of an algorithm or problem, we use the **Big-O** and **Theta**(Θ) notation.
- Suppose f is some function of n.
- An **algorithm**
  - Is in $O(f(n))$ if its (worst case) runtime for inputs of size n grows **no faster than** $f(n)$
  - Is in $\Theta(f(n))$ if its (worst case) runtime for inputs of size n grows **no slower than** $f(n)$
- Similarly, a **problem**
  - Is in $O(f(n))$ if the time needed to solve it for inputs of size n grows **no faster than** $f(n)$
  - Is in $\Theta(f(n))$ if the time needed to solve it for inputs of size n grows **no slower than** $f(n)$

Big-O and Theta notation: example

- One important problem we will encounter in more detail later is **sorting** an array
- Atomic operations are
  - **Comparing** two values (array entries or other variables)
  - **Assigning** a new value (to an array entry or other variable)
- Some other operations can be treated as "essentially atomic", such as **swapping** two values
  - Can be done using three assignments
  - This constant factor 3 is ignored for the order of growth
- It can be proven that the sorting problem is in $\Theta(n \cdot \log n)$
- We will see some algorithms whose complexity is in $O(n \cdot \log n)$, i.e. as good as possible

Examples

- Complexity classes for some typical code samples:
  - Constant ($O(1)$):
    Atomic statement like **`a = 1;`**
  - Linear ($O(n)$):

    ```
    for(int i = 0; i < n; i++)
        a[i] = 1;
    ```

  - Quadratic ($O(n^2)$):

    ```
    for(int i = 0; i < n; i++)
        for(int j = 0; j < n; j++)
            a[i][j] = 1;
    ```

    Or also

    ```
    for(int i = 0; i < n; i++)
        for(int j = 0; j < i; j++)
            a[i][j] = 1;
    ```

Examples

- Generally, e nested loops which all run up to n are in $O(n^e)$
- More involved example:

```
for(int i = 0; i < n; i++)
    for(int j = 0; j < n; j++)
        for(int k = 0; k < 3; k++)
            for(int l = 0; l < i; n++)
                for(int m = 0; m < k; m++)
                    x++;
```

  - **`i`** runs up to **`n`**, so it counts
  - **`j`** runs up to **`n`**, so it counts
  - **`k`** runs up to 3, **independent** of **`n`**, so doesn't count
  - **`l`** runs up to **`i`** which runs up to **`n`**, so it counts
  - **`m`** runs up to **`k`** which is independent of **`n`**, so doesn't count
- So this example takes $O(n^3)$ steps

Examples: logarithmic factors

- Complexity classes containing logarithmic factors, like
  - logarithmic ($O(\log n)$)
  - linearithmic ($O(n \cdot \log n)$)

  typically arise from divide-and-conquer algorithms which we will see later
- The core idea is to solve the problem by
  - cutting the data into halves
  - solving sub-problems on each half

Examples: exponential factors

- Complexity classes containing exponential factors usually arise when
  - We have a number of variables (or array entries, or ...) growing with n
  - We have to go through all combinations of values to find the solution
- Example: in the satisfiability (SAT) problem
  - We are given a logical formula like $(\neg A \vee B \vee C) \wedge (A \vee \neg C \vee \neg D) \wedge (\neg B \vee D \vee E) \wedge \ldots$
  - We want to know if there is some way of assigning *true* or *false* to $A, B, C, \ldots$ which makes the formula true
  - The straightforward algorithm is to just try all combinations
  - For n variables, that is 2^n combinations

The empirical approach

- We can get evidence regarding complexity by sampling runtimes of an implementation on a variety of inputs
- This is an example to the **empirical** approach used throughout science
- Advantage: easy to implement given an implementation of the algorithm and a way to obtain suitable inputs
- Some caveats:
  - Depends on chosen inputs
    - Many problems have easy special cases with lower complexity
    - How to ensure that our inputs are representative?
  - Depends on implementation details
    - You are essentially testing the implementation
    - This can help find implementation errors if you know what the complexity of the algorithm actually should be
  - Depends on hardware (memory, cache) , amount of CPU used by other processes, . . .

The empirical approach

- Basic idea: if the complexity of an implementation is
  - Logarithmic, then repeatedly **doubling** the input size will always increase the runtime by the **same amount**
  - Linear/quadratic/cubic, then repeatedly **doubling** the input size will always **multiply** the runtime by 2/4/8, respectively
  - Exponential, then repeatedly **increasing** the input size by a fixed amount will always **multiply** the runtime by some fixed amount
- These relations will usually be approximate only
- Need enough data points to draw any conclusions

The empirical approach

- Suppose we have measured these runtimes:

Input size	Algorithm 1
100	7
200	13
400	27
800	52

- The input size doubles from row to row
- **Dividing** each runtime by the previous one gives $13/7 = 1.857, 27/13 = 2.077, 52/27 = 1.926$ so they are approximately doubling
- This is evidence that the algorithm's complexity is linear

The empirical approach

- Suppose we have measured these runtimes:

Input size	Algorithm 2
100	17
200	22
400	28
800	33

- The input size doubles from row to row
- Taking the **differences** between successive runtimes gives $22 - 17 = 5, 28 - 22 = 6, 33 - 28 = 5$
  i.e. they don't change except for small fluctuations
- This is evidence that the algorithm's complexity is logarithmic