Data Structures & Algorithms (F28DA)

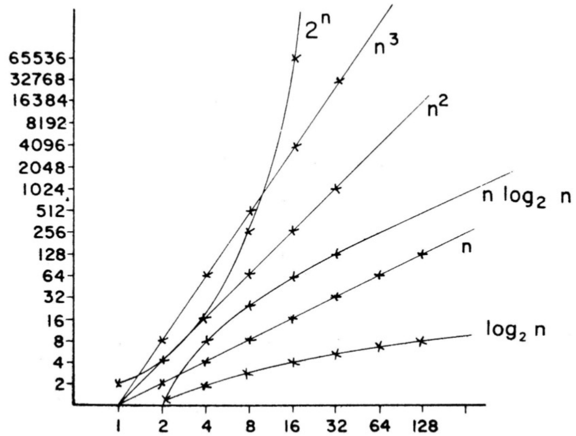
Complexity

Kathrin Stark

Heriot-Watt University, 2024

Reminder: Complexity

Illustration of Growth Rates



Fundamentals of Data Structures, E Horowitz & S Sahni, 1998.

Intro to Data Structures & Algorithms (F28SG)

Last Lecture

Introduction to Complexity

Rob Stewart

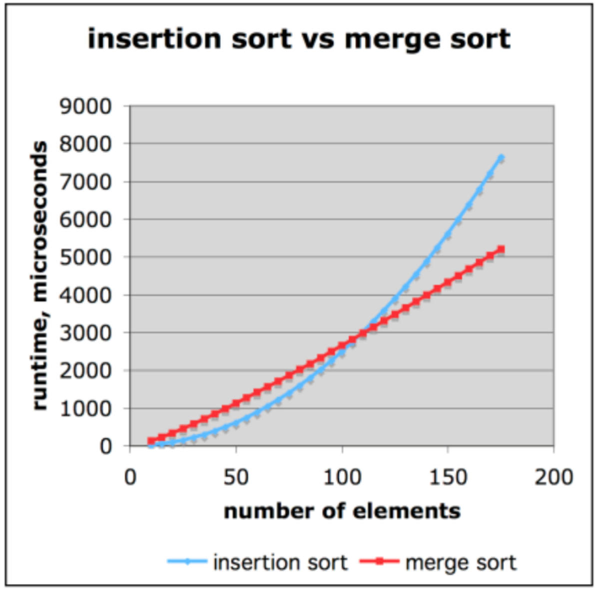
Disclaimer:
We assume you to know what you learned in F28SG.
The next slides are **not** teaching these parts but only a **reminder**.
The pink frame means: You should know this from before!

Today You'll

- Learn the **formal definition of O-notation** for expressing algorithmic complexity.
- Explore how the O-notation definition justifies **simplifying terms** and categorizing into **different complexity classes**.
- Discover methods for inferring complexity classes from actual program runtimes.
- Learn how some common **patterns** translate into complexity.
- Gain insights into the concept of **brute force algorithms** and its implications.

Slide by Matt Stallmann
included with permission.

Comparison of Two Algorithms



insertion sort is
$n^2 / 4$

merge sort is
$2 n \lg n$

sort a million items?

insertion sort takes roughly 70 hours

while

merge sort takes roughly 40 seconds

This is a slow machine, but if 100 x as fast then it's 40 minutes versus less than 0.5 seconds

Reminder: Why Growth Rate Matters

Slide by Matt Stallmann included with permission.

Complexity From a Program's Runtime

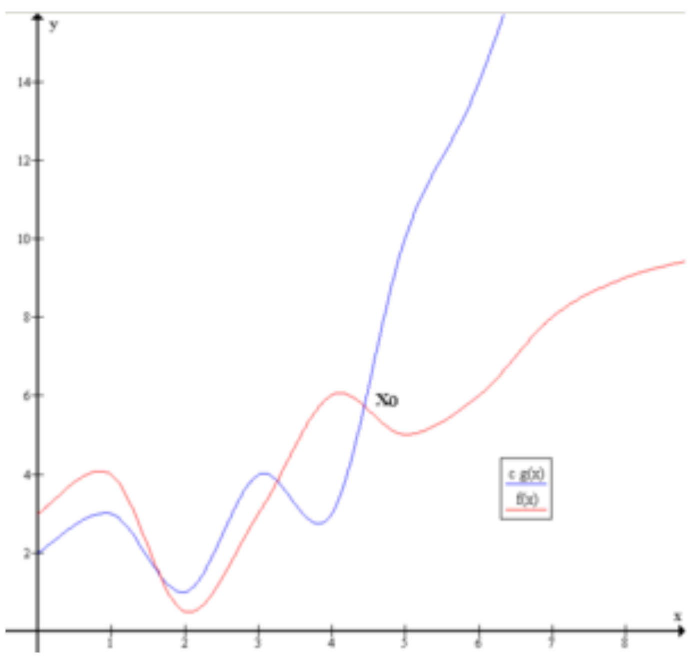
if runtime is...	time for n + 1	time for 2 n	time for 4 n
c lg n	c lg (n + 1)	c (lg n + 1)	c(lg n + 2)
c n	c (n + 1)	2c n	4c n
c n lg n	~ c n lg n + c n	2c n lg n + 2cn	4c n lg n + 4cn
c n^2	~ c n^2 + 2c n	**4c n^2**	16c n^2
c n^3	~ c n^3 + 3c n^2	8c n^3	64c n^3
c 2^n	c 2^{n+1}	c 2^{2n}	c 2^{4n}

runtime quadruples when problem size doubles

Note: This table can also help us if we want to guess the complexity from a program's runtime.

A **Formal** Definition of Big O Notation

Given functions f(n) and g(n), we say that f(n) is O(g(n)) if there are positive constants c and n_0 such that
$f(n) \leq cg(n)$ for $n \geq n_0$.



Why this definition?

- n0 – We care only about the **asymptotic** behaviour (= for big inputs).
- c – With different hardware we cannot influence constant factors.

With this formal definition of O-notation, we can
1. show why the previous **simplifications are valid**.
2. show that the previous **classes** are indeed different.

Simplifying Functions

Last Lecture

- In Big-O try to write functions in the **simplest terms**
- Rules are used to simplify terms
  - **drop lower-order terms**
  - **drop constant factors**
- For example: $6N+5$
  1. we drop the lower-order term 5 (left with 6N)
  2. we drop the constant 6 (left with N)
  3. meaning it is expressed by the linear function **O(N)**
- Another example: $5n^3 + 2n^2 + 3n - 4$
  1. drop the lower terms: $2n^2$, $3n$ and 4
  2. drop the constant 5
  3. meaning it is the cubic function **O(n^3)**

Next lecture: Why are these simplifications justified?

Simplification – Example 1

- *Example*: $2n + 10$ is $O(n)$.

Simplification – Example 2

- *Example*: $2n^2 + 3n$ is $O(n^2)$.

Simplification – Example 3

- *Example*: 3 log n + 5 is O(log n).

General Laws

Let f_1 be a function in $O(g_1)$, f_2 be a function in $O(g_2)$, and k be a nonzero constant.

Let further $f + g$, $\max(f, g)$ and $k * f$ be defined by

$$(f + g)(n) = f(n) + g(n)$$
$$(\max(f, g))(n) = \max(f(n), g(n))$$
$$(k * f)(n) = k * f(n)$$

Then:

1. $f_1 + f_2$ is in $O(\max(g1, g2))$.

2. $k * f_1$ is in $O(g1)$.

You will prove 2. on next week's assignment.

Proof of Sums O-Notation

Let f_1 be a function in $O(g_1)$, f_2 be a function in $O(g_2)$.
Then $f_1 + f_2$ is in $O(\max(g1, g2))$.

Proof.

Examples of Growth Rates

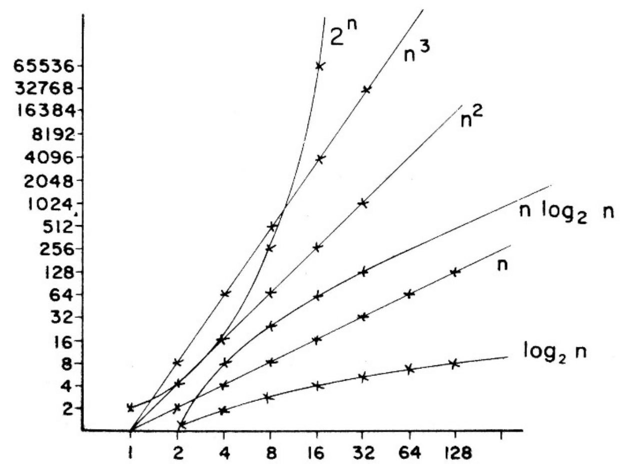
Last Lecture

Common Functions

- We then need to simplify the function
- The following functions are very commonly used:
  - **O(1) - the constant function**
  - **O(log n) - the logarithmic function**
  - **O(n) - the linear function**
  - **O(n log n) - the n-log-n function**
  - **O(n^2) - the quadratic function**
  - O(n^3) - the cubic function
  - O(2^n) - the exponential function
- This are listed in *order of complexity*
  - *O(1)* is the simplest, while *O(2^n)* is the most complex
- *We will come across and introduce the* ***bold*** *functions during this course*

Next lecture: We will see why all these are
- different orders of complexity.
- the order holds.
(This requires the exact definition of O-notation.)

Illustration of Growth Rates



Fundamentals of Data Structures, E Horowitz & S Sahni, 1998.

Example:

The function $f(n) = n^2$ is not $O(n)$.

Big O and Growth Rate

Given functions f(n) and g(n), we say that f(n) is O(g(n)) if there are positive constants c and n_0 such that $f(n) \leq cg(n)$ for all $n \geq n_0$.

=> The Big O notation gives an **upper bound** on the growth rate of a function:
The statement "f(n) is O(g(n))" means that the growth rate of f(n) is no more than the growth rate of g(n).

Let
$f_1(n) = 1$
$f_2(n) = n$
$f_3(n) = n^2$

Are the following statements true or false?

a) $f_2(n)$ is $O(f_2(n))$

b) $f_1(n)$ is $O(f_2(n))$

c) $f_3(n)$ is $O(f_2(n))$

Related Complexity Notations

O(n)

Given functions f(n) and g(n), we say that f(n) is O(g(n)) if there are positive constants c and n_0 such that $f(n) \leq cg(n)$ for $n \geq n_0$.

Intuition: "f(n) is O(g(n))" ~ the growth rate of f(n) is no more than the growth rate of g(n)

Ω(n)

Given functions f(n) and g(n), we say that f(n) is Ω (g(n)) if there are positive constants c and n_0 such that $f(n) \geq cg(n)$ for $n \geq n_0$.

Intuition: "f(n) is Ω (g(n))" ~ the growth rate of f(n) is more than the growth rate of g(n)

Θ(n)

Given functions f(n) and g(n), we say that f(n) is Θ (g(n)) if f(n) = O(g(n)) and f(n) = Ω(g(n)).

Intuition: "f(n) is Θ (g(n))" ~ the growth rate of f(n) is corresponds to the growth rate of g(n)

See an alternative definition for Θ (g(n)) on next week's exercise sheet.

Later (second half): Amortized cost.

Common Patterns
Nested Loops

```
for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
                { ... f(i,j) … }
```

… if a loop contains another loop, we can usually multiply the run times.

What is the asymptotic runtime if

a) `f(i,j)` requires constant time

b) `f(i,j)` requires O(n) time

c) `f(i,j)` requires O(log n) time

d) `f(i,j)` requires O(2^n) time

Common Patterns
Loop Variables Halving

```
for (int i = n; i > 0; i /= 2)
 { ... constant time …  }
```

What's the runtime?

… if the loop variable halves each time, this is **usually** an indication of a log(n) factor.

Note: See also **next lecture**!
This gets more complicated together with recursive calls.

Common Patterns

Recursion 1

```
Algorithm Factorial
Input: A non-negative integer n
Output: The factorial of n (n!)

if n equals 0 then
    return 1   // Base case: 0! = 1

else
    return n * Factorial(n - 1)
```

What is the asymptomatic runtime?

Common Patterns

Recursion 2

```
Algorithm fibonacci
Input: An integer n (position of the Fibonacci number)
Output: The nth Fibonacci number

if n <= 1 then
    return n   // Base case: fibonacci(0) = 0, fibonacci(1) = 1

return fibonacci (n-1) + fibonacci (n-2)
```

What's the runtime?
Attention: Follow the argument and how often recursion is called.
These are relatively easy examples – more next week.

Common Design Techniques and their Runtime

An Outlook

- **Recall**: We have now seen a formal definition of O-notation and how the runtime is determined for simple patterns.
- **Next**:
  - Discuss common design techniques that are applicable to different problems.
  - **Today**: Brute Force
  - **Monday**: Backtracking, Divide & Conquer
  - **Later**: Greedy algorithms, Dynamic Programming
- … an important **factor:**
  - Different algorithms solving the same problem can have vastly different runtimes
  - O-notation will help us to choose an appropriate algorithm for a given problem
  - So all these techniques will include a discussion on what impact the specific technique has on the runtime.

Brute Force

Brute Force Algorithm = A straightforward and exhaustive problem-solving technique that

1. Enumerates all possible solutions
2. Checks whether one of these is actually a solution

without leveraging optimisation or heuristic strategies.

Correctness? – Trivial as all solutions are considered

Efficiency? – Often quite inefficient

Runtime:

- **Number of possible solutions:** g(n)
- **Time to check a solution:** f (n)
- **Total time**: g(n) * f(n)

Example 1: Prime Numbers

To find out whether n is a prime number, calculate whether
n mod i = 0 for i = 2 .. n-1.

```
Algorithm IsPrime
Input: A positive integer n
Output: True if n is prime, False if n
isn't a prime

if n <= 1 then
    return False

for i from 2 to n - 1:
    if n % i == 0 then
        return False

return True
```

1. Why would this be considered brute force?

2. What is the **run time**?

3. Can you think of ways to be more efficient?

Example 2: Linear Search

Linear Search (1)

- In **linear search** we
  - start at the beginning of the array
    - and compare until we find a match
    - which we return
  - Here, is a variant which searches a list of integer and returns
    - **true** if the list contains the element
    - **false** if not

```
public static boolean searchForNumber(int[] arr,int number){
  for(int i = 0; i < arr.length; i++){
    if (arr[i] == number) {
      return true;
     }
  }
  return false;
}
```

1. Why would this be considered brute force?

2. What is the run time?

3. Can you think of ways to be more efficient?

Where Brute Force Algorithms are Used

- … really not a design technique but rather just used if there is no other possibility.
  An example that not only correctness but also efficiency matters!
- Might be used as a benchmark
- Might be feasible when the size of the set is small

Examples:

- Linear search
- For sorting a list `xs`, enlisting all permutations and choosing the sorted one (see next exercise sheet)
- Traveling Salesman Problem**:** Trying all possible orders in which a salesman can visit a set of cities to find the shortest route.

Next Lecture: More efficient design techniques

Today You'll

- Learn the **formal definition of O-notation** for expressing algorithmic complexity.
- Explore how the O-notation definition justifies **simplifying terms** and categorizing into **different complexity classes**.
- Discover methods for inferring complexity classes from actual program runtimes.
- Learn how some common **patterns** translate into complexity.
- Gain insights into the concept of **brute force algorithms** and its implications.

Reading

- Chapter 1.4
- Lecture Slides of Lecture 2 of F28GS (also available on Canvas)

Optional Reading

- Geogebra.org - allows you to plot graphs
- Big-O Complexity Sheet (https://www.bigocheatsheet.com) – Overview of complexity for most standard algorithms
- Base.cs podcast on O-notation