Algorithms and Data Structures

Introduction

dr Szymon Murawski

Comarch SA

March 1, 2019

Table of contents I

Course plan

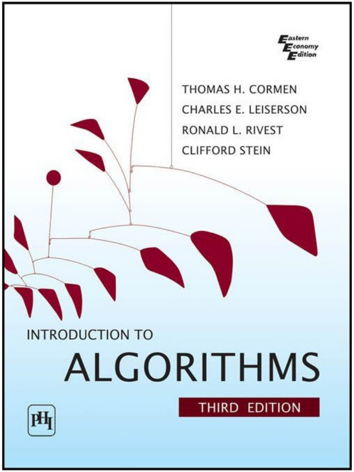
Course plan

Syllabus

Refer to .docx document on Moodle.

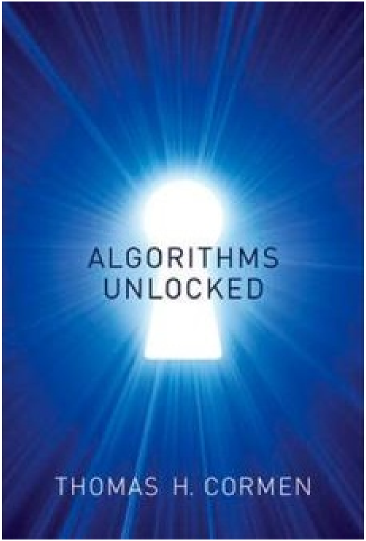
Course plan

Introduction to Algorithms



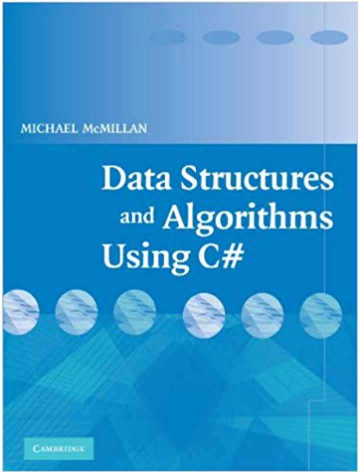
- Thomas Cormen, Charles Leiserson, Ronald Rivest, Clifford Stein
- Exhaustive (1000+ pages) study of modern computer algorithms and data structures
- Don't let the word "introduction" fool you!

Algorithms Unlocked



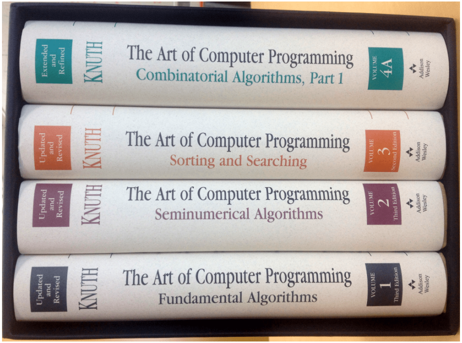
- Thomas Cormen
- Great introductory book
- Understandable even by non-programmers
- Still packs a lot of knowledge

Data structures and Algorithms Using C#



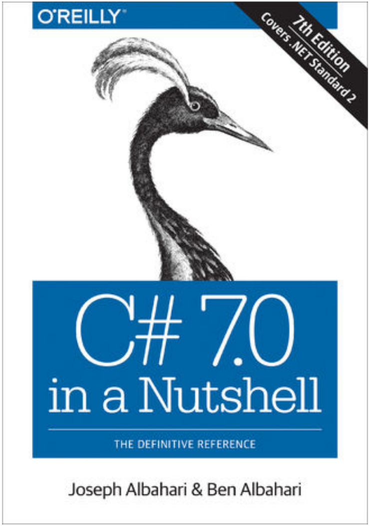
- Michael McMillan
- C# implementations of algorithms and data structures
- Sadly full of bugs
- Awful brackets placement

The Art of Computer Programming



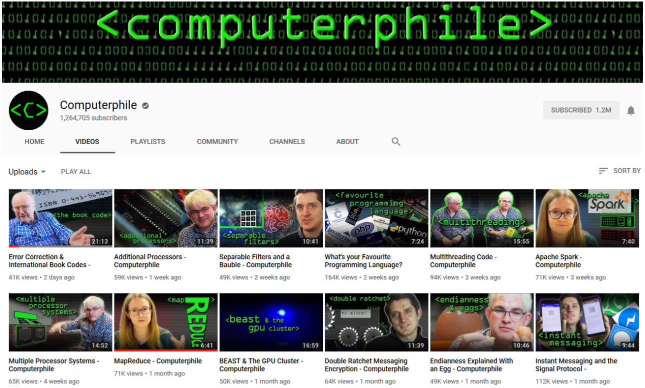
- Donald E. Knuth
- The ultimate reference
- Virtually bug-free
- Very challenging!

C# in a nutshell



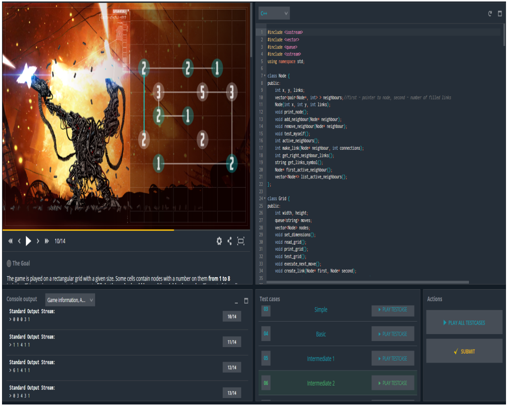
- Joseph & Ben Albahari
- Very good book on C#
- Updated to recent version of .Net standard
- Covers basics as well as advanced topics
- Probably the only book on C# that you will ever need

Computerphile



- YouTube channel
- Hundreds of videos on different skill level
- A little bit of math with a lot of hand drawing
- Also check out "Numberphile"

Codingame



- http://www.codingame.com
- Great source of practice tasks on all skill levels
- A lot of programming languages supported
- After solving a puzzle you can check other solutions in a given language

Course plan

Course plan

What is an algorithm

Definition

- An algorithm is a set of explicit finite steps which, when carried out for a given set of initial conditions (input), produce the corresponding output and terminate in finite time.
- An algorithm should be generic, universally applicable to a class of problems

Origin of name

Term 'Algorithm' comes from 9th century Persian mathematician Abū Abdallāh Muhammad ibn Mūsā al-Khwārizmī from Khorasan (modern Uzbekistan).
al-Khwārizmī → Al-Khwarithmi → Algoritmi

Most important algorithms in the world

- PageRank
- RSA
- Diffie-Hellman key exchange
- Fourier and Fast Fourier Transform
- Auto-Tune
- Dijkstra algorithm
- NewsFeed
- "You might also like"
- MP3 compression
- High-Frequency Trading

Why study algorithms and data structures

- It let's you look differently at some problems, often finding not obvious solutions.
- In your career as a programmer you will not be constrained by specific language or paradigm.
- You will understand, that faster computers are not always the answer
- You can understand what's behind algorithms you use daily, like Facebook's NewsFeed, Shazam, Google's PageRank etc.
- Many interview questions in IT are about algorithms and data structures.
- You will accept that majority of security issues arise from human error, not due to algorithms used
- They are just beautiful!

Course plan

Natural language

Searching for maximum value in a set

1. If there are no numbers in the set then there is no highest number.
2. Assume the first number in the set is the largest number in the set.
3. For each remaining number in the set: if this number is larger than the current largest number, consider this number to be the largest number in the set.
4. When there are no numbers left in the set to iterate over, consider the current largest number to be the largest number of the set.

1. Low level of abstraction
2. Challenging transformation to computer code
3. Rarely used for complex algorithms

Formal definition

Factorial

$$n! = \begin{cases} 1, & \text{if } n = 0 \\ n(n-1)!, & \text{if } n > 1 \end{cases}$$

- Mathematical expression of algorithm
- Requires at least basic knowledge of mathematical operators
- Very high level of abstraction
- Can be difficult to understand
- Transformation to computer code can be hard

Computer code

```
public static void SieveOfEratosthenes(int n) {
  bool[] prime = new bool[n+1];
  for(int i = 0; i < n; i++)
    prime[i] = true;
  for(int p = 2; p*p <= n; p++) {
    if(prime[p] == true)
      for(int i = p*p; i <= n; i += p)
        prime[i] = false;
  for(int i = 2; i <= n; i++)
    if(prime[i] == true)
      Console.Write(i + " ");
}
```

- Ready to run out of a box
- Can be hard to transform into another programming language
- Implementation details could overshadow the algorithmic core

Pseudocode

Euclidean algorithm for greatest common divisor

- Input: Two positive integers m, n
- Output: greatest common divisor of m and n

```
If m<n then swap(m,n)
while n != 0
   let r = m mod n
  m = n
  n = r
return m
```

- Blend of natural language and computer code
- Ease of transforming into computer code
- Not implementation specific
- Medium level of abstraction

Course plan

Course plan

Comparing algorithms

Problem

Given several algorithms able to solve specific problem, which one do you choose?

Criteria for choosing an algorithm

- Speed
- Memory footprint
- Difficulty of implementation
- Difficulty of understanding the algorithm
- Flexibility

Algorithm efficiency

- Our measurements of algorithm efficiency should be independent of computer architecture, programming language, current state of the system etc.
- We search for abstract unit, expressing the character of the dependency while omitting unimportant details

- Algorithm efficiency is described as a function of cost (resource) vs the size of input data.
- We are looking for efficiency for very large input - an asymptotic analysis.
- Key to finding proper function is identification of dominant operations in a given algorithm i.e. comparison of elements, swapping.
- Algorithm can depend on the distribution of input data, our analysis should identify best and worst-case scenarios

Analyzing algorithms

- While analyzing algorithms we can measure the following:
  - **Time** - while measuring time we need to make sure, that some other processes are not interfering with our program
  - **Space** - memory used by our algorithm
  - **Operations performed** - we select one (or more) dominant operations and calculate how many of them are executed for input of given size
- Usually Time and Operations performed complexity is the same, but space complexity can be very different!
- For some specialized algorithms other metrics can be taken into account like count of database queries or file access operations.

Types of algorithm complexity

- Pessimistic - the upper bound of execution time (or memory usage) for all input data. For some classes of algorithms pessimistic case occurs quite common.
- Average - the average time an algorithm should run for typical case (problem: what is a typical case).
- Optimistic - the best case scenario, occurring quite rare, most of the times should not be taken into account. Important exemptions include cryptography - we don't want our passwords to be cracked fast!

Calculating algorithm efficiency

Assumptions

- Instead of calculating the real time of every instruction we say they take an abstract 1 unit of time each.
- We concentrate on fastest growing term, as in the limit of big input all other terms would be much smaller
- We omit all constants
- We assume that better algorithm is described by a function of lower order.
- Sometimes we take into account smaller terms, especially when comparing algorithms of the same class.

Course plan

Big-O notation

- To describe algorithms we use Big-O notation
- Assume we have function f(n), that scales with input size
- We say that $f(n) = O(g(n))$, if there are constants c, n_0 such that $f(n) \leq c * g(n)$ for $n \geq n_0$.
- Using $O(n)$ we omit constants and lower order functions
- $O(n)$ is an upper bound on a function
- Similarly $\Omega(n)$ is a lower bound - $f(n) = \Omega(g(n))$, if there are constants c, n_0 such that $0 \leq cg(n) \leq f(n)$ for $n \geq n_0$.
- $\Theta(n)$ is tight bound - $f(n) = \Theta(g(n))$, if there are constants c_1, c_2, n_0 such that $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ for $n \geq n_0$.
- In our analysis we will concentrate on $O(n)$

Asymptotic notation graphs

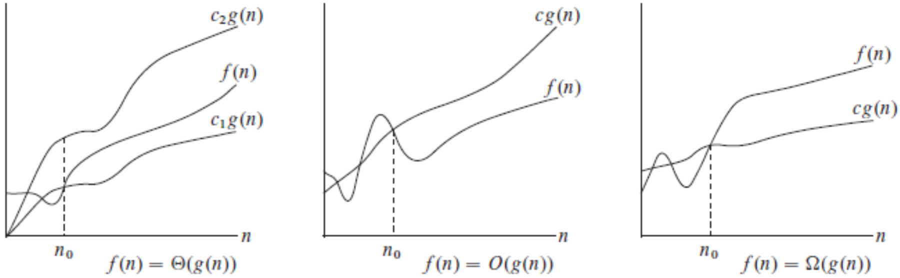


Figure: Taken from: Cormen, Leiserson, Rivest, Stein - *Introduction to algorithms*

Classes of algorithms

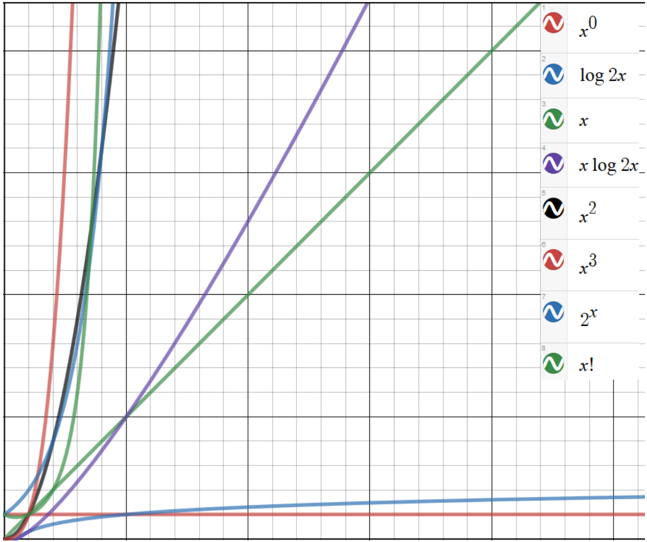
Polynomial classes

- $O(1)$ - determining if a number is odd or even
- $O(lgn)$ - finding a word in a dictionary
- $O(n)$ - reading a book
- $O(nlgn)$ - sorting a deck of cards
- $O(n^2)$ - checking if you have everything from your shopping list

Non-polynomial classes

- $O(x^n)$ - brute-force attempt at guessing a password of size x
- $O(n!)$ - factorial, i.e. traveling salesman
- $O(\infty)$ - algorithms that can potentially never end, i.e. tossing a coin on its edge

Classes of complexity



Example

Given input data $n = 10^6$ and assuming computer performs 10^6 operations per second how long would it take to obtain results for different classes of algorithms?

Complexity	Operations performed	Real time
$O(1)$	1	$1\mu s$
$O(n)$	10^6	$1s$
$O(n^2)$	10^{12}	10 days
$O(n^3)$	10^{18}	27 years
$O(2^n)$	10^{301030}	10^{301016} years

- Computer speed doubles every 18 months
- This can make problems described with polynomial algorithms solvable in finite time
- Exponential complexity however is not solvable by improving our computers
- That is why data security and cryptography is based on algorithms being solvable in exponential time!

Course plan

Linear search

Problem

- Given an array A with size n, filled with random data, how can we check if an element x is in array?
- We assume, that we know nothing how the data is organized
- We search for best, average and worst case scenario

Course plan

Simple linear search

Pseudocode

- Inputs:
  - A - an array
  - n - number of elements in the array
  - x - searched value
- Output: Index i for which A[i] = x, or -1 if the value x is not found in array A.

```
1 1 let res = −1
2 2 for i in (0..n−1)
3 3   if A[i] = x then res = i
4 4 return res
5
```

- Dominant operation is comparison (line 3)
- Each time we are traversing whole array, we could do better!

Course plan

Improved linear search

Pseudocode

- Inputs:
  - A - an array
  - n - number of elements in the array
  - x - searched value
- Output: Index i for which A[i] = x, or -1 if the value x is not found in array A.

```
for i in (0..n−1)
   if A[i] = x then return i
return −1
```

- Should run faster than the previous one
- In fact, in every loop we are making two comparisons, `A[i]==x` and `i==n-1`
- We can get rid of second comparison!

Course plan

Improved linear search with sentinel

Pseudocode

- Inputs:
  - A - an array
  - n - number of elements in the array
  - x - searched value
- Output: Index i for which A[i] = x, or -1 if the value x is not found in array A.

```
let last = A[n-1]
A[n-1] = x
while A[i] != x
   i++
if i < n-1 or last = x then return i
return -1
```

- Best running time of all algorithms

Calculating complexity of linear search

Best case scenario

Element is at the beginning of the array, so complexity is $O(1)$

Worst case scenario

Element is at the end of the array, we need to traverse it all, so complexity is $O(n)$

Average case

Element could be in array at position k with probability $p_k = 1/n$. We are then making on average

$$T(n) = \sum_{k=1}^{n} kp_k = \frac{1}{n}\sum_{k=1}^{n} k = \frac{1}{n}\frac{n(n+1)}{2} = \frac{n+1}{2}$$

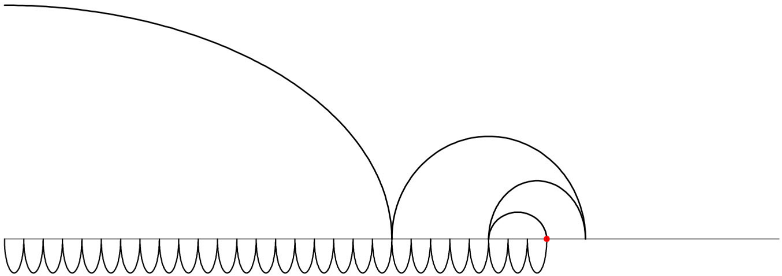
comparisons. Algorithm is still of class $O(n)$.

Course plan

Binary search

- We cannot further improve linear search
- However if we know something about the data we are going through, we can take advantage of that
- Simple case - for sorted data we can use binary search
- As before we are looking for best, average and worst case scenarios

Linear and binary search comparison



Binary search pseudocode

Pseudocode

- Inputs:
  - A - an array
  - n - number of elements in the array
  - x - searched value
- Output: Index i for which A[i] = x, or -1 if the value x is not found in array A.

```
Set p = 0, r = n−1
While p <= r do
  Set q = floor((p+r)/2)
  If A[q] = x then return q
  Else If A[q] != x and A[q] > x, then set r = q−1
  Else if A[q] < x then set p = q+1
Return −1
```

Visualization

Searched value $x = 38$

	0	1	2	3	4	5	6	7	8	9
	2	5	8	13	16	23	38	56	72	91
$p = 0, q = 4, r = 9$	2	5	8	13	16	23	38	56	72	91
$p = 5, q = 7, r = 9$	2	5	8	13	16	23	38	56	72	91
$p = 5, q = 5, r = 6$	2	5	8	13	16	23	38	56	72	91
$p = 6, q = 6, r = 6$	2	5	8	13	16	23	38	56	72	91

Calculating complexity of binary search

Best case scenario

Element is exactly in the middle, so complexity is $O(1)$

Worst case scenario

Assuming we can divide exactly by two every time we get

$$T(n) = T(n/2) + 1 = T(n/4) + 1 + 1 = T(n/2^k) + \underbrace{(1 + \cdots + 1)}_{k}$$

In worst case scenario $2^k = n \rightarrow k = lgn$, so we are left with:

$$T(n) = T(n/n) + lg\ n$$

In worst case complexity is then $O(lg\ n)$

Calculating complexity of binary search

Average case

- First step checks one place in the middle, so probability $p_1 = 1/n$
- In second step we check the middle of 2 subarrays, so $p_2 = 2/n$
- In third step we check positions in a total of 4 subarrays, so $p_3 = 4/n$
- In xth step the probability is $p_x = 2^{x-1}/n$

Then the number of steps in average case is

$$T(n) = \sum_{k=1}^{lg\ n} \frac{k2^{k-1}}{n} = \frac{1}{n}\sum_{k=1}^{lg\ n} k2^{k-1} < \frac{1}{n}\sum_{k=1}^{lg\ n} lg\ n2^{k-1} = \frac{lg\ n}{n}\sum_{k=1}^{lg\ n} 2^{k-1}$$
$$= \frac{lg\ n}{n}(2^{lg\ n} - 1)1 = \frac{lg\ n}{n}(n-1) < lg\ n$$

So the complexity is $O(lg\ n)$, same as in worst case!