Simple algorithms with integer addition and subtraction and arrays, decision problems, other problems, lossless compression

Adam Boult (www.bou.lt)

April 30, 2025

Contents

Preface

This is a live document, and is full of gaps, mistakes, typos etc.

Part I

Integer maths algorithms

Chapter 1

Algorithms for integer multiplication

1.1 Introduction

1.1.1 Introduction

Chapter 2

Algorithms for integer division, modulus and remainders

2.1 Introduction

2.1.1 Introduction

Chapter 3

Calculating natural number square roots

3.1 Introduction

3.1.1 Introduction

We might want an algorithm that returns 4 for $f(17)$. The floor of the square root.

This is useful, for example, for factorising a number.

We can start at 0 and square numbers and see if the result is larget than x, incrementing each time.

```
while i * i <= x:
    x += 1;
return x - 1;
```

Chapter 4

Identifying primes

4.1 Identifying primes

4.1.1 Identifying primes

different to factorising. We don't care what the actual factors are, just see if it's prime

4.1.2 Fermat's primality test

Fermat's little theorem recap

Fermat's primality test

From Fermat's little theorem we know

$a^{n-1} = 1 mod(n)$

Where a is an integer and n is prime.

Chapter 5

Factorising natural numbers

5.1 Integer factorisation

5.1.1 Trial division

We have x

Divide by numbers between 2 and x

Only need to go to sqrt x

Don't need to divide by even numbers other than 2

algorithm for checking if number is a prime

loop up dividing number from 2

if divides, add factor list and divide target number by that

stop when i reaches number

eg for 45

divide 2? no

divide 3? yes :¿ 15

divide 3? yes :¿ 5

divide 4? no

divide 5? yes :¿ 1

6¿1 so stop

number is prime if list just contains target don't have to worry about including non primes in list, as will already have divded by that amount

5.1.2 Fermat's method

Identify the integer as the difference of two squares, and use this.

$x = a.b$

We use the midpoint of the two as $c = \frac{a+b}{2}$

This only works for odd numbers. If we have

The we have:

- $a = c + d$
- $b = c - d$
- $x = (c + d)(c - d)$
- $x = c^2 - d^2$

We can test this by trying a to get $a^2 - x$, and seeing if this is a square number.

Part II

Arrays and simple array algorithms

Chapter 6

Arrays

6.1 Introduction

6.1.1 Defining arrays

A sequence

6.2 Read operations on arrays

6.2.1 The match operation

6.2.2 The read operation

A sequence.

Chapter 7

Reversing arrays

7.1 Introduction

7.1.1 Introduction

Chapter 8

Reductions on arrays

8.1 Getting the max and min

8.1.1 Getting the max and min

Reduction algorithm:

+ Take array. If array is length 0 throw problem

+ If array is length 1 return element

+ If array is length 2 do pairwise comparison on the pair (eg return bigger of two for max)

+ If array is length greater than 2, recursively call reduction on reduction of first two elements and the rest of the array.

Examples of reductions that can be done include:

+ Min

+ Max

+ Sum

+ Count if

+ Sum if

Chapter 9

Sorted arrays and bubble sort

9.1 Sorted lists

9.1.1 Sorted arrays

There can be a total ordering on elements in a array.

We want to return an array such that only the ordering is changed.

$\forall nm[array[n] > array[m] \leftrightarrow n > m]$

9.2 Checking if an array is sorted

9.2.1 Checking a sortable array

9.3 Bubble sort

9.3.1 Bubble sort

Take the first two items. See if they are sorted. If they are not, swap them.

Then move to next pair, and do same.

Keep going until the end.

If the number of swaps was greater than 0, loop around again.

Worst case: $O(n^2)$ comparisons and $O(n^2)$ swaps. Average case: $O(n^2)$ comparisons and $O(n^2)$ swaps.

Best case: $O(n)$ comparisons and $O(1)$ swaps.

This is an in place algorithm.

Chapter 10

Selection sort

10.1 Selection sort

10.1.1 Selection sort

Set up another array of same length. the sorted array.

Go through unsorted array and look for min (can use reduction algorithm).

Put minimum in sorted list to left.

Remove that element from unsorted.

+ if linked list can just remove (but we haven't gotten to those yet) + if array, make new array?

keep going until sorted list exists.

Worst case same as bubble ($O(n^2)$ for comparisons and swaps) but average is only $O(n)$ swaps.

Intuitively because each element only gets moved once.

Chapter 11

Insertion sort

11.1 Insertion sort

11.1.1 Insertion sort on arrays

start by taking the first two elements and either keeping or swapping. This is the sorted part of the list now.

Go to next element If bigger, ok next If smaller, scan across sorted part of list to see where it belongs. Move elements up as necessary and insert the element.

Average $O(n^2)$ for swaps and comparisons.

Chapter 12

Searching sorted and unsorted arrays

12.1 Identifying the location of an element in an array

12.1.1 Identifying the location of an element in an array

12.2 Getting location in sorted array with binary search

12.2.1 Binary search on a sorted array

Get midddle item in array, if less than target number, then can drop lower half of array and iterate.

Chapter 13

Filtering and slicing arrays

13.1 Introduction

13.1.1 Introduction

Chapter 14

Concatenating arrays

14.1 Introduction

14.1.1 Introduction

Chapter 15

Merging sorted arrays

15.1 Introduction

15.1.1 Introduction

Part III

Decision problems and assessing algorithms

Chapter 16

Decision problems

16.1 Introduction

16.2 Introduction

Chapter 17

Correctness of algorithms

17.1 Correctness

17.1.1 Correctness

An algorithm is correct if it produces the expected output for each input.

17.1.2 Partial and total correctness

An algorithm is only partially correct if may not terminate. Otherwise it is totally correct.

17.1.3 Formal verification

17.1.4 Model checking

Model checking allows the formal verification of algorithms with finite inputs. test every possible input.

17.1.5 Deductive verification

Check the parts of the algorithm using theorem provers.

Chapter 18

Measuring algorithmic complexity with big-O notation

18.1 Efficiency

18.1.1 Algorithmic efficiency

An algorithm takes memory and time to run. Analysing these characteristics of algorithms can enable effective choice of algorithms.

Complexity is described using big-O notation. So an algorithm with parameters θ would have a time efficiency of $O(f(\theta))$ where $f(\theta)$ is a function of θ.

Generally we expect $f(\theta)$ to be weakly increasing for all θ. As we add additional inputs, these would not decrease the time or space requirements of the algorithm.

An algorithm which did not change complexity with inputs would have a constant as the largest term. So we would write $O(c)$.

An algorithm which increase linearly with inputs could be written $O(\theta)$.

An algorithm which increase polynomially with inputs could be written $O(\theta^k)$.

An algorithm which increased exponentially could be written $O(e^\theta)$.

Complexity can differ between worst-case scenarios, best-case scenarios and average case scenarios.

We can describe logical systems by completeness (all true statements are theorems) and soundness (all theorems are true). We have similar definitions for algorithms.

An algorithm which returns outputs for all possible inputs is complete. An algorithm which never returns an incorrect output is optimal.

18.1.2 Big-O and little-o recap

18.1.3 Time efficency

18.1.4 Space efficiency

18.1.5 Verifying answers

NP NP-hard NP-complete

18.1.6 Decision problems

Return yes or no.

18.2 Calculating the cost of an algorithm

18.2.1 Instruction costs

18.2.2 Efficiency of loops

number of times each instruction called

18.2.3 Big-O recap (take from maths)

18.2.4 Efficiency of functions with arguments

best case, worst case

Chapter 19

P (PTIME), EXPTIME, DTIME and simulation by Turing-equivalent machines in polynomial time

19.1 Introduction

19.1.1 Introduction

P (aka PTIME): Polynomial in time. $O(poly(n))$

EXPTIME: $O(2^{poly(n)})$

DTIME(f(n)) .ie P is DTIME(poly(n))

Chapter 20

Hardness of problems and completeness of problems in a given complexity class

20.1 Introduction

20.1.1 Hardness

A problem p is hard for a class C if every problem in C can be reduced to p.

That is, p is C-hard if every problem in C can be reduced to p.

20.1.2 Completeness

A problem p is complete for a class C if it is C-hard and in C.

If an "easy" solution is found for a problem p which is C-complete, there is an "easy" solution to all problems in C.

Chapter 21

L (LSPACE), PSPACE, EXPSPACE, DSPACE

21.1 Introduction

21.1.1 Introduction

L (aka LSPACE): Logarithmic in space. $O(log(n)$

PSPACE: Polynomial in space: $O(poly(n)$.

EXPSPACE: $O(2^{poly(n)})$

DSPACE(f(n)) .ie L is DSPACE(log(n))

Chapter 22

The relationships between P, L and PSPACE

22.1 Introduction

22.1.1 Introduction

P is no larger than PSPACE.

P is at least as big as L.

Part IV

Problems reducible to decision problems: Search problems and optimisation problems

Chapter 23

Search problems and reducing them to decision problems

23.1 Introduction

23.2 Introduction

Chapter 24

Optimisation problems and reducing them to decision problems

24.1 Introduction

24.2 Introduction

Part V

Problems not reducible to decision problems: Counting problems and function problems

Chapter 25

Counting problems and their complexity classes (including #P)

25.1 Introduction

25.2 Introduction

Chapter 26

Function problems and their complexity classes (including FP)

26.1 Introduction

26.2 Introduction

Chapter 27

Polynomial-time reductions

27.1 Introduction

27.1.1 Introduction

27.1.2 Polynomial-time Turing reduction (the Cook reduction)

Solve using polynomial number of calls to another problem, and polynomial amount of time outside that.

27.1.3 Many-one reduction

Special case of the Cook reduction. Transform input of one problem to input of another, where answers are the same.

Transformation of inputs must be done in polynomial.

27.1.4 Truth table reduction

Another special case of the Cook reduction.

Transforms inputs into a number of other inputs to a different problem. Result is a function of the outputs of the other problem.

Chapter 28

Log-space reductions

28.1 Introduction

28.1.1 Introduction

Part VI

Simple lossess compression

Chapter 29

Simple lossless compression

29.1 Lossless compression

29.1.1 Compression rates

29.1.2 Run-length encoding: The ND model

eg 12W6RABC4D is WWWWWWWWWWWWRRRRRRABCDDD

or 4444444aaaaaa123 to 447aa6123

ND model. N is number of repeats, D is what to repeat. if bigger than N can take, then split up

eg 111111111111: 9131

29.1.3 RLE with binary/bitstream

thing next on how that works with binary/bitsteam (eg could do 3 bits at a time for 85)

29.1.4 Run-length encoding: The data packet model

If there is something which repeats a lot (eg 0) then can split that out and then do data packets for the rest

eg if we have 000036400000000000006305: 04364090363015

this is RND model?

The strength of RLE with data packets depends on frequency of special character.

29.1.5 Run-length encoding with delta encoding

we can use delta encoding to make repeated characters more likely to be 0 and non zero is present.

do 2 digits to show going to be a run

what about cases like 1111122222

becomes 115225, but how do we know it's not 52 1s, a 2 then a 5? encoding tricks?

29.1.6 LZW compression

A. Lempel and J. Ziv, with later modifications by Terry A. Welch

code table. eg $2^12 = 4096$ codes. first $256(0 - 255)$ are the literal bytes

256-4095 are blocks of bytes

algorithm is how to determine code table

29.1.7 zip, deflate and lzma2

zip

deflate

lzma2