Course introduction

Dario Facchinetti

Welcome

- Course material at:
  - dariofad.github.io/teaching/comp
  - **university e-learning platform**
- About:
  - dariofad.github.io, seclab.unibg.it
  - `dario.facchinetti@unibg.it`
- Overview:
  - Introduction to Competitive Programming (40 hours)
    - review the common data structures and approaches used in coding competitions
    - **improve our ability to prototype efficient solutions**
  - Introduction to Evolutionary Computing (8 hours)
    - prof. Angelo Gargantini

Topic

What are coding competitions?

- solve algorithmic challenges in a limited amount of time
- Many categories, test environments, single player vs team
- ACM-ICPC, Code Jam, CodeChef, Codeforces, Codewars, Hash Code, Kick Start, LeetCode, TopCoder

Figure: Codeforces logo from webpage

Coding problems

How does it work?

1. read the problem statement (and the constraints)
2. try to make the problem more abstract
3. design and implement a solution (i.e., *algorithm*)
4. test and debug locally (optional)
5. submit the solution to the online judge system

Example

- statement: given a sequence of numbers S, return its sum
- input: S is a sequence of space-separated integers
- constraints: $0 \leq S_i \leq 100$
- output: $\sum_i S_i$

```
# sum a sequence of integers
def solve(stdin):
    return sum(map(int, stdin.split()))

return solve('2 18 3 5 10 15')
```

Let's see it in practice!

- LeetCode
- example of *medium* problem: k-closest points to the origin
- https://leetcode.com/problems/k-closest-points-to-origin/

Another example

Let's see another practical demonstration!

```
Given a sequence of non-negative integers, find a matching
pair that is equal to a target sum
```

Demo - pt.1

Solution 1

```
def solution_1(sumT, sequence):
    """All the couples of elements"""

    for i in range(len(sequence) - 1):
        for j in range(i + 1, len(sequence)):
            if sequence[i] + sequence[j] == sumT:
                return sequence[i], sequence[j]

    raise Exception("Result not found")
```

Demo - pt.2

Solution 2

```
def solution_2(sumT, sequence):
    """Sorting + single-sweep scanning strategy"""

    sequence.sort()
    start, end = (0, len(sequence) - 1)
    while start <= end:
        if sequence[start] + sequence[end] < sumT:
            start += 1
        elif sequence[start] + sequence[end] > sumT:
            end -= 1
        else:
            return sequence[start], sequence[end]

    raise Exception("Result not found")
```

Demo - pt.3

Solution 3

```
def solution_3(sumT, sequence):
    """Set + single-sweep"""

    elements = set()
    for elem in sequence:
        if sumT - elem in elements:
            return elem, sumT - elem
        else:
            elements.add(elem)

    raise Exception("Result not found")
```

Demo - pt.4

Testing the solutions

```
def generate_data(size):

    # generate a sequence unique integers
    sequence = [i for i in range(size)]
    # shuffle the sequence (no longer ordered)
    random.shuffle(sequence)

    # generate a valid solution for our test (sumT)
    left_operand = 2 * size
    right_operand = size + 5
    sumT = left_operand + right_operand

    # add the solution at a random position in the sequence
    left_pos, right_pos = None, None
    while left_pos == right_pos:
        left_pos = random.randint(0, size - 1)
        right_pos = random.randint(0, size - 1)
    sequence[left_pos] = left_operand
    sequence[right_pos] = right_operand

    return (sumT, sequence)
```

Demo - Results

- Hardware configuration
  - AMD Ryzen 9 3900X 12-Core Processor
  - Caches (sum of all):
    - L1d: 384 KiB (12 instances)
    - L1i: 384 KiB (12 instances)
    - L2: 6 MiB (12 instances)
    - L3: 64 MiB (4 instances)
  - 64 GB RAM
- Solution 1: 30-second timeout with 100k elements
- Solution 2: 30-second timeout with 50M elements
- Let's comment together Solution 3

Why it matters

- improve your problem solving ability
- improve your programming skills
- improve your knowledge of a programming language
- learn data structures and study common patterns
- challenge yourself on hard problems

Mindset

- participate in weekly contests, test your code with Online Judges, read the solutions from other developers
- focus on new problems, test different approaches

```
The key is deliberative practice: not just doing it again
and again, but challenging yourself with a task that is
just beyond your current ability, trying it, analyzing your
performance while and after doing it, and correcting any
mistakes. Then repeat. And repeat again.
```

from Peter Norvig http://norvig.com/21-days.html

Course program

An introduction to competitive programming:

- `U1` - computational complexity and asymptotic notation *
- `U2` - fundamentals of Python programming *
- `U3` - sorting
- `U4` - data structures and common patterns
- `U5` - dynamic programming and recursion
- `U6` - problems by category
- `Evolutionary Computing`

Units 1* and 2* can be considered the background

Study material

- Slides, notes, and solutions to problems (course website)
- Books:
  - `Introduction to Algorithms, 4th edition, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein, ISBN 9780262046305` (3rd edition available in italian)
  - `Algorithms, 4th edition, Robert Sedgewick and Kevin Wayne, ISBN 9780321573513`
  - `Introduction to Evolutionary Computing, 2nd edition, ISBN 3662448742, Agoston E. Eiben and James E. Smith`

Exam

Written Exam:

- 3 exercises, 10 points for each exercise
  - 2 Competitive Programming exercises
  - 1 Evolutionary Computing exercise
- 1.5 to 2 hours time available
- 18 points required to pass the exam

Project (optional):

- 0, 1 or 2 additional points
- points are added if the exam is passed
- project submission deadline: 24/06, h. 10:30
- submit the project via email

Typical exam exercise

Implement the Selection sort algorithm

```
def selectionSort(sequence: list[int]):
    # ...
    pass
```

Project

- The project is individual
- Send me an email about the topic
- In general, there are two alternatives:
  - implementation and analysis (max 4 pages) of algorithms
  - solution and analysis (max 4 pages) of problems from coding competitions
- Ideas:
  - implement your own hashmap
  - generation of prime numbers
  - flow or optimization problems

PhD students

Attendance required

Weekly calendar

- Please check the official timetable
- Weekly lessons:
  - Wednesday 8.30-10.30, lab Galvani, 1st floor
  - Friday 14.00-16.00, room B005
- Office hours (for students):
  - Wednesday, 10.30-11.30, Dalmine, Building B, room 3.05
  - Please send an email to make an appointment

Computational Complexity (review)

Dario Facchinetti

Unit 1

- computational complexity review

Topic

- How do we measure the runtime of an algorithm?
- Elementary operations
- Architecture

What is computational complexity?

Given an algorithm, the computational complexity is the amount of resources to run it

Given a problem, its complexity is the complexity of the best algorithm that allows solving it

Example 1

Read all the letters in a given sequence

$$A, B, C, D, E, F, G, H, I, L, M, N, O, P, Q, R, S, T$$

Example 2

Read all the letters in a given sequence until you find an A

- case 1:

$$A, B, C, D, E, F, G, H, I, L, M, N, O, P, Q, R, S, T$$

- case 2:

$$L, B, C, D, E, F, G, H, I, A, M, N, O, P, Q, R, S, T$$

- case 3:

$$T, B, C, D, E, F, G, H, I, L, M, N, O, P, Q, R, S, A$$

Example 2

Read all the letters in a given sequence until you find an A

- case 1:

$$A, B, C, D, E, F, G, H, I, L, M, N, O, P, Q, R, S, T$$

- case 2:

$$L, B, C, D, E, F, G, H, I, A, M, N, O, P, Q, R, S, T$$

- case 3:

$$T, B, C, D, E, F, G, H, I, L, M, N, O, P, Q, R, S, A$$

best, average, worst

Growth of functions

When the input size becomes enough large, we can focus on the order or growth of the running time of an algorithm

The extra-precision is not usually worth the effort of computing it. For large enough inputs, the multiplicative constants and lower-order terms of an exact running time are dominated by the effects of the input size itself.

$\rightarrow$ **asymptotic** efficiency of algorithms

Θ notation

For a given function $g(n)$, we denote by $\Theta(g(n))$ the set of functions:

$$\Theta(g(n)) =$$

$$\{f(n) : \exists\ c_1, c_2, n_0 \mid 0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n),\ \forall n \geq n_0\}$$

c_1, c_2, n_0 constants ≥ 0

Θ notation - sandwich

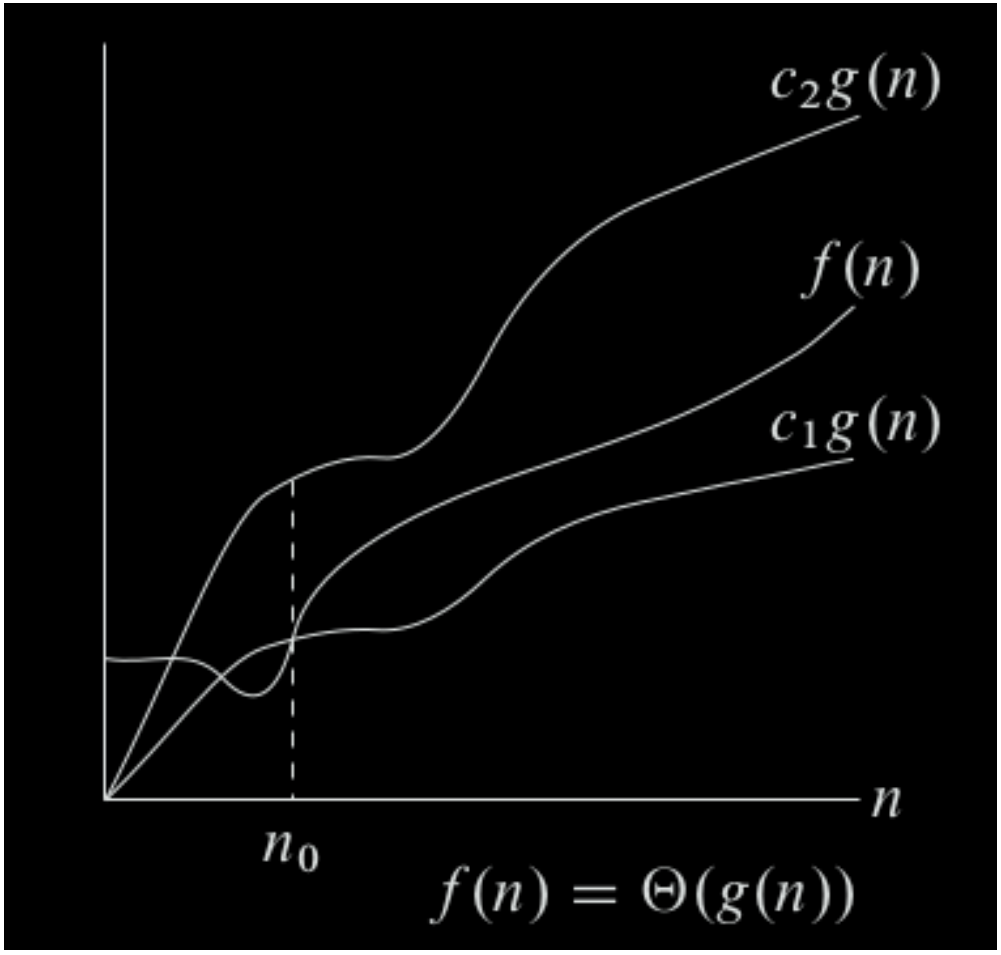


Figure: visualization of the big-theta notation, from Introduction to Algorithms, by T. H. Cormen, C. E. Leiserson, R. L. Rivest and C. Stein

Notation

$\Theta(g(n))$ is a set so we should write $f(n) \in \Theta(g(n))$, however we abuse of notation and express the same notion with

$$f(n) = \Theta(g(n))$$

The same applies for the following concepts

O notation

For a given function $g(n)$, we denote by $O(g(n))$ the set of functions:

$$O(g(n)) =$$

$$\{f(n) : \exists\ c, n_0 \mid 0 \leq f(n) \leq c \cdot g(n),\ \forall n \geq n_0\}$$

c, n_0 constants ≥ 0

O notation - upper

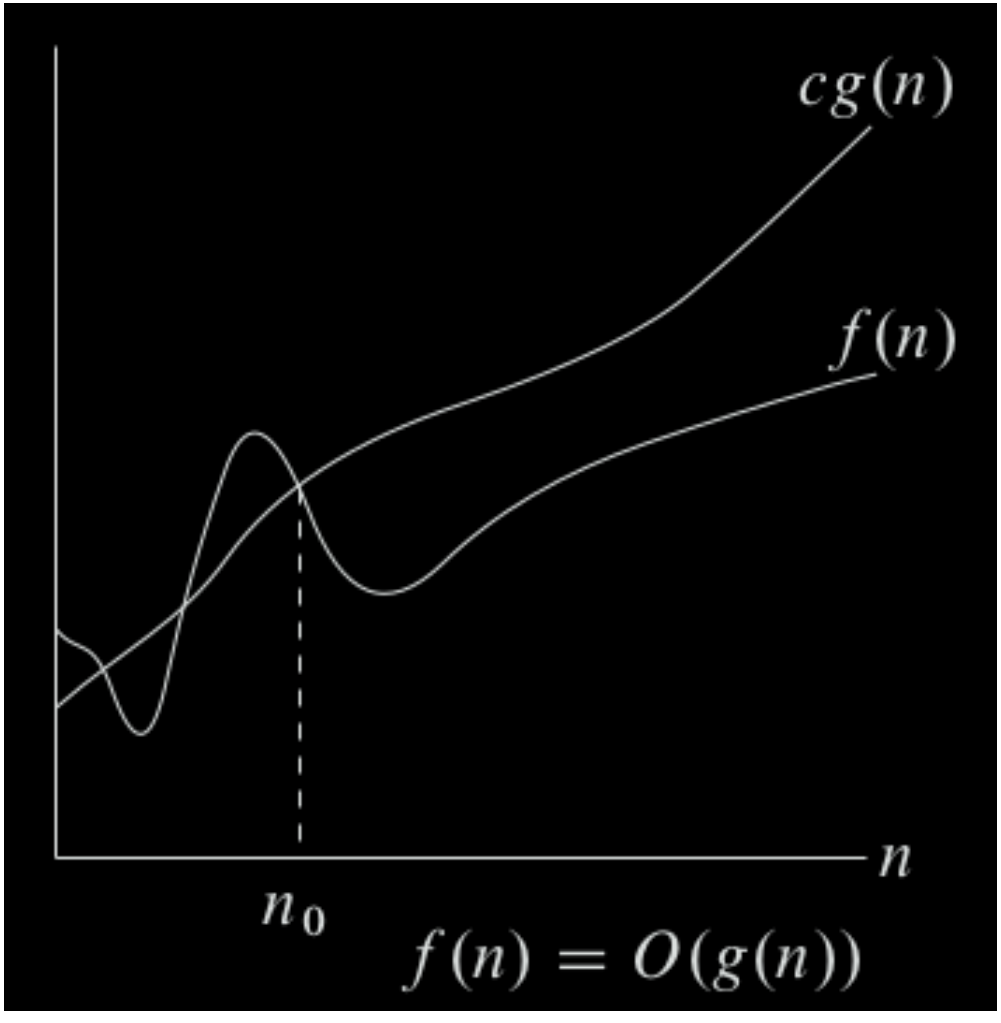


Figure: visualization of the big-oh notation, from Introduction to Algorithms, by T. H. Cormen, C. E. Leiserson, R. L. Rivest and C. Stein

Ω notation

For a given function $g(n)$, we denote by $\Omega(g(n))$ the set of functions:

$$\Omega(g(n)) =$$

$$\{f(n) : \exists\ c, n_0 \mid 0 \leq c \cdot g(n) \leq f(n),\ \forall n \geq n_0\}$$

c, n_0 constants ≥ 0

Ω notation - lower bound

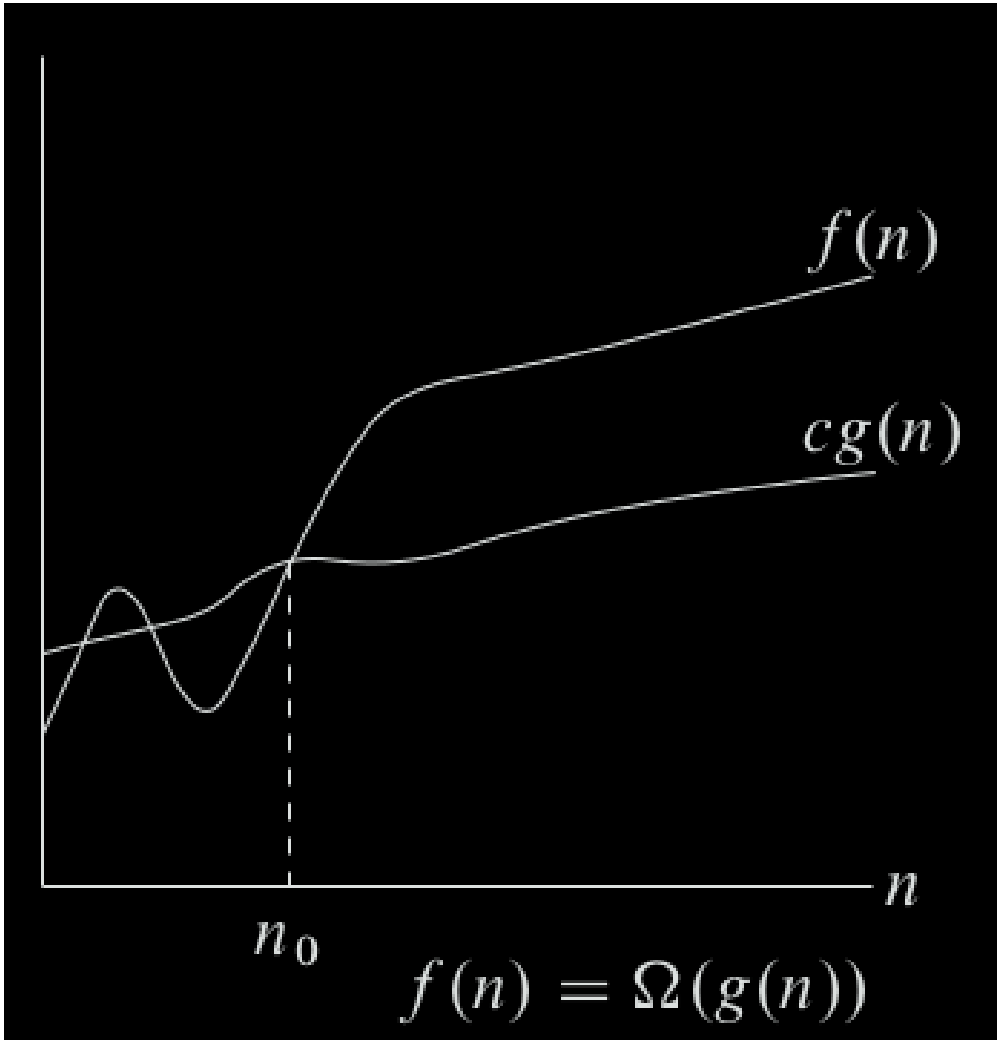


Figure: visualization of the big-omega notation, from Introduction to Algorithms, by T. H. Cormen, C. E. Leiserson, R. L. Rivest and C. Stein

Comments

Even though $\Theta(\cdot)$ can describe with more precision the complexity of an algorithm, $O(\cdot)$ is very important in practice, as it describes the upper bound

With $O(\cdot)$ one can describe well the worst case scenario

With $\Omega(\cdot)$ one can describe well the best solution to a problem

Complexity Table

Table: big-oh complexity table

Complexity	Description
$O(1)$	constant
$O(\log(n))$	logarithmic
$O(n)$	linear
$O(n \log(n))$	log-linear
$O(n^2)$	quadratic
$O(n^3)$	cubic
$O(n^k)$	polynomial
$O(k^n)$	exponential
$O(n!)$	factorial

$k > 1$

What is feasible in practice? I

```
import math

max_order = 10
n = [10**i for i in range(1, max_order + 1)]

def log(n):
    return math.log(n)

def linear(n):
    return n

def loglinear(n):
    return n*math.log(n)

def quadratic(n):
    return n*n

def cubic(n):
    return n**3

def exponential(n):
    return math.pow(2, n)
```

What is feasible in practice? II

```
def factorial(n):
    # approximation
    return math.sqrt(2*math.pi*n)  *\
        math.pow(n/2.71, n) * \
        math.pow(2.71, 1/(12*n))

complexities = [log,
                linear,
                loglinear,
                quadratic,
                cubic,
                exponential,
                factorial,
 ]

def stringify(ops):

    ops_per_musec = 1e3
    ops_per_msec = 1e6
    ops_per_sec = 1e9
    ops_per_day = ops_per_sec * 60 * 60 * 24
```

What is feasible in practice? III

```
    if ops < ops_per_musec:
        return "{} ns".format(round(ops))
    if ops < ops_per_msec:
        return "{} mus".format(round(ops/ops_per_musec))
    if ops < ops_per_sec:
        return "{} ms".format(round(ops/ops_per_msec))
    if ops < ops_per_day:
        return "{} s".format(round(ops/ops_per_sec))
    else:
        return "-"

def compute(n, func):
    try:
        ops = func(n)
        return  stringify(ops)
    except:
        return "-"
```

What is feasible in practice? IV

```
# runtime table
runtime = [[0] * \
           (len(complexities)+1) for _ in range(len(n)+1)]
# insert complexity name
for c in range(len(complexities)):
    runtime[0][1+c] = complexities[c].__name__
runtime[0][0] = "n"

for i in range(len(n)):
    # input size
    runtime[i+1][0] = "10^{}".format(i)
    # expected runtime
    for j in range(len(complexities)):
        runtime[i+1][j+1] = compute(n[i], complexities[j])

return runtime
```

Approximate runtime

approximate runtime based on the input size

$\sim 10^9$ ops/sec architecture, $-$ stands for practically not-feasible

n	log.	lin.	loglin.	quad.	cub.	exp.	fact.
10^1	$2ns$	$10ns$	$23ns$	$100ns$	$1\mu s$	$1\mu s$	$4ms$
10^2	$5ns$	$100ns$	$461ns$	$10\mu s$	$1ms$	-	-
10^3	$7ns$	$1\mu s$	$7\mu s$	$1ms$	$1s$	-	-
10^4	$9ns$	$10\mu s$	$92\mu s$	$100ms$	$1000s$	-	-
10^5	$12ns$	$100\mu s$	$1ms$	$10s$	-	-	-
10^6	$14ns$	$1ms$	$14ms$	$1000s$	-	-	-
10^7	$16ns$	$10ms$	$161ms$	-	-	-	-
10^8	$18ns$	$100ms$	$2s$	-	-	-	-
10^9	$21ns$	$1s$	$21s$	-	-	-	-
10^{10}	$23ns$	$10s$	$230s$	-	-	-	-

Space complexity

Time is not the only thing that matters

We can use what we have seen so far to measure the memory or *space* to run an algorithm too

Space is used to allocate variables and objects, but also when a sequence of function calls is performed (e.g., recursive call stack)

Before going into details let's review some math

A bit of math - floor and ceil

For all real x

$$x - 1 < \lfloor x \rfloor \leq x \leq \lceil x \rceil < x + 1$$

For any integer x

$$\lfloor x/2 \rfloor + \lceil x/2 \rceil = x$$

For any integer a and any positive integer n

$$a \bmod n = a - n\lfloor a/n \rfloor$$

A bit of math - exponentials

For all real $a > 0$, m and n

$$a^0 = 1$$

$$a^1 = a$$

$$a^{-1} = 1/a$$

$$(a^m)^n = a^{mn},\ (a^m)^n = (a^n)^m$$

A bit of math - exponential vs polynomial

For all real constants a and b such that $a > 1$

$$\lim_{n \to \infty} \frac{n^b}{a^n} = 0$$

Any exponential function with a base strictly greater than 1 grows faster than any polynomial function

A bit of math - logarithms I

$$\ln n = \log_e n$$

$$\log^k n = (\log n)^k$$

$$\log \log n = \log(\log n)$$

A bit of math - logarithms II

For all real $a > 0$, $b > 0$, $c > 0$, and n (in each equation logarithm bases $\neq 1$)

$$a = b^{\log_b a}$$

$$\log_c ab = \log_c a + \log_c b$$

$$\log_b a^n = n \log_b a$$

$$\log_b a = \frac{\log_c a}{\log_c b}, \quad \log_b \frac{1}{a} = -\log_b a, \quad \log_b a = \frac{1}{\log_a b}$$

$$a^{\log_b c} = c^{\log_b a}$$

A bit of math - factorial

$n! = 1 \cdot 2 \cdot 3 \cdot 4 \cdot \ldots \cdot n$

Weak upper bound $n! \leq n^n$

Stirling's approxmation

$$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n e^{\frac{1}{12n}} \to \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \left(1 + \Theta\left(\frac{1}{n}\right)\right)$$

Let's practice (1)

You are given the following functions, the goal is to order them based on the asymptotic behavior

- $f_1(n) = n$
- $f_2(n) = \sqrt{n}$
- $f_3(n) = n^2$
- $f_4(n) = n + \sqrt{n}$

Let's use the blackboard!

Let's practice (1)

Answer: $(f_2, \{f_1, f_4\}, f_3)$

Let's practice (2)

You are given the following functions, the goal is to order them based on the asymptotic behavior

- $f_1(n) = 2^n$
- $f_2(n) = \binom{n}{n/2}$
- $f_3(n) = \binom{n}{4}$
- $f_4(n) = n!$
- $f_5(n) = n^{100}$

Let's use the blackboard!

Let's practice (2)

- $f_1(n) = 2^n$
- $f_4(n) = n!$
- $f_5(n) = n^{100}$

Partial answer: (f_5, f_1, f_4)

What about binomial coefficients?

Let's practice (3)

- $f_1(n) = 2^n$
- $f_2(n) = \binom{n}{n/2}$
- $f_3(n) = \binom{n}{4}$
- $f_4(n) = n!$
- $f_5(n) = n^{100}$

Answer: $(f_3, f_5, f_2, f_1, f_4)$

Can you guess the complexity?

- we have reviewed the concept of computational complexity
- we have also seen how to compare different runtimes
- let's study the complexity of some functions written in Python
- for each of them let's also **discuss a better implementation**

Code questions (1)

What is the time and space complexity of the following function?

```
def minimum(sequence):
    minv = 0
    if len(sequence) > 0:
        minv = sequence[0]
    else:
        return Exception("empty sequence")
    for elem in sequence[1:]:
        if elem < minv:
            minv = elem
    return minv
```

space $O(1)$, time $O(n)$, where n is the length of the `sequence`

Code questions (2) - constants

What is the time and space complexity of the following function?

```
def sums(sequence):
    if len(sequence) == 0:
        return Exception("empty sequence")
    sumv = 0
    square_sum = 0
    for elem in sequence:
        sumv = sumv + elem
        square_sum = square_sum + elem * elem
    return (sumv, square_sum)
```

space $O(1)$, time $O(n)$, where n is the length of the `sequence`

Code questions (3) - recursion I

What is the time and space complexity of the following function?

```
def sumf(n):
    if n == 0:
        return n
    return n + sumf(n - 1)
```

space $O(n)$, time $O(n)$

What can we say about the compiler? Some languages can leverage tail call optimization, in that case the space becomes $O(1)$

Code questions (4) - quad

What is the time and space complexity of the following function?

```
def cartesian_sum(sequence):
    if len(sequence) == 0:
        return Exception("empty sequence")
    if len(sequence) == 1:
        return sequence[0]
    sumv = 0
    idl = 0
    idr = 1
    while idl < len(sequence) - 1:
        while idr < len(sequence):
            sumv = sequence[idl] + sequence[idr]
            idr = idr + 1
        idl = idl + 1
        idr = idl + 1
    return sumv
```

space $O(1)$, time $O(n^2)$, where n is the length of the `sequence`

Code questions (5) - log

What is the time and space complexity of the following function? (`sequence` is ordered)

```
def search(sequence, key):
    if len(sequence) == 0:
        return false
    left = 0
    right = len(sequence) - 1
    while left <= right:
        middle = (left + right) // 2
        if sequence[middle] == key:
            return True
        if sequence[middle] < key:
            left = middle + 1
        else:
            right = middle - 1
    return False
```

space $O(1)$, time $O(\log n)$, where n is the length of the `sequence`

Code questions (6) - exp

What is the time and space complexity of the following function?

```
def nodes(h):
    if h <= 1:
        return 1
    return nodes(h - 1) + nodes(h - 1)
```

space $O(h)$, time $O(2^h)$

Code questions (7) - functions

What is the time and space complexity of the following function?

```
def logger(strings):
    for s in strings:
        print(s)
```

space $O(1)$, time $O(n \cdot m)$, where n is the number of `strings` and m is the length of the longest `s`

Code questions (8) - multiple params

What is the time and space complexity of the following function?

```
def odd_sum(data, weights):
    sumv = 0
    for w in weights:
        if w % 2 == 1:
            return 0
    for d in data:
        sumv = sumv + d
    return sumv
```

space $O(1)$, time $O(n+m)$, where n is the length of `data` and m is the length of `weights`

Code questions (9) - useless allocations

What is the time and space complexity of the following function?

```
def reverse(values):
    if len(values) <= 1:
        return values
    r = [0]*len(values)
    for p, v in enumerate(values):
        r[len(values) - 1 - p] = v
    return r
```

space $O(n)$, time $O(n)$, where n is the length of `values`

Code questions (10) - termination

What can we say about the following function?

```
import random

def do_some():
    while random.choice([True, False]) != True:
        continue
    return
```

Code questions (11)

What is the time and space complexity of the following function?

```
def is_prime(n):
    if n < 1:
        return Exception("undefined behavior");
    if n == 1 or n == 2:
        return True
    x = 2
    while x * x <= n:
        if n % x == 0:
            return False
        x = x + 1
    return True
```

space $O(1)$, time $O(\sqrt{n})$

Code questions (12) - recursion II

What is the time and space complexity of the following function?

```
def fibonacci(n):
    if n <= 0:
        return 0
    if n == 1:
        return 1
    else:
        return fibonacci(n - 1) + fibonacci(n - 2)
```

space $O(n)$, time $O(2^n)$ (not very 'tight' bound)

Recap

- computational complexity
- *practical* solutions (not only for coding competitions)
- asymptotic behavior
- time vs space
- constants vs asymptotic behavior
- multiple params with no established relation vs asymptotic behavior
- allocations vs function calls

What's next

- advantages and drawbacks of compiled vs interpreted programming languages
- choice of the programming language for this course
- basic data structures

Choice of the programming language

Dario Facchinetti

Unit 2

- Programming language choice: advantages and drawbacks
- Python fundamentals
  - memory model
  - string, list, tuple, hashmap, set
  - import system, namespaces
  - stdin/stdout
- Practice with simple exercises

Topic

- Programming language choice: advantages and drawbacks

Programming language choice

Which language should we choose...

- to code the most efficient solution?
- to study an algorithmic problem?
- to solve a coding interview?

There are advantages and drawbacks

Efficiency - 1

Compiled and statically typed languages are associated with the **lowest runtime**

Also, zero-cost abstractions can help us **minimize memory consumption**

Examples are C, C++ and Rust

Efficiency - 2

. . . but we also must explicitly handle the memory and deal with aspects that are not necessarily associated with the problem we are trying to solve

E.g., object de-allocation, memory safety

Readability and Simplicity - 1

When we implement solutions in coding contests we tend not to focus on readability, an aspect which is very important in practice

Some languages such as Python or JavaScript permit to implement solutions concisely, with a syntax that is closer to pseudocode

Readability and Simplicity - 2

These languages are usually interpreted and dynamically typed

This permits to reduce the effort we spend to code the solution, as we can focus only on the problem

However there is a price to pay at runtime (more time and space)

Libraries

When selecting a language we should also consider the libraries it provides, and some of them come with powerful libraries that can be used out of the box

To give some examples, Python and C++ provide a powerful standard library compared to a language such as C, this permits to avoid re-implementing everything from scratch

An incremental approach to the solution

When learning algorithms it is often useful to test small functions interactively, for example to understand how to use a new data structure

Read-eval-print loops are very useful to this end

Choice

Let's summarize what we have discussed so far!

Which is the best answer for you?

Python probably offers the best trade-off for this course, but keep in mind that the most popular choice among competitive programmers is C++

Python fundamentals - pt.1

Dario Facchinetti

Topic

Let's review Python programming fundamentals

- `Python is a high-level, general-purpose programming language that emphasizes code readability`
- Get started: https://www.python.org/

Python intro

Properties of Python:

- general purpose
- dynamically typed
- garbage-collected
- object-oriented
- support for functional programming

There are many implementations (CPython, Jython, Python for .NET, IronPython, PyPy), we will focus on the most widely used (CPython)

Syntax and keywords

Python is characterized by an extremely simple syntax, with very few keywords

False	await*	else	import	pass
None	break	except	in	raise
True	class	finally	is	return
and	continue	for	lambda	try
as	def	from	nonlocal	while
assert	del	global	not	with
async*	elif	if	or	yield

There are also some *soft keywords*, or rather identifiers that are reserved only for specific contexts (e.g., `match`, `case`)

Source code format

- default is UTF-8 unless otherwise specified
- logical vs physical lines
- explicit (\) vs implicit line joining
- statements
- whitespace to separate tokens in a statement
- tokens are read from left to right
- grouping of statements based on indent level (tabs or spaces)
- comments with # and """

Demo:

- source code format

How does it work exactly?

1. write a source file `program.py`
2. run in the console `python program.py`
    - lexer (**tokens**), parser (**ast**), compiler (**bytecode**)
    - interpreter executes the program

Demo:

- tokenize, parse, compile (then disassemble)

What about the REPL?

Data model

Definition of **objects**, **values** and **types** (and identity)

`Objects are Python's abstractions for data. All data in a Python program are represented by objects or by relations between objects. Code is also represented by objects`

Demo:

1. `==`, `is`, `id()`, `type()`
2. reference count
3. `del`

Value update

- There are mutable and immutable objects
- Object mutability is determined by the type:
  - numbers, strings and tuples are immutable
  - dictionaries and lists are mutable
- How do you mutate an immutable object? When a new number is assigned to a variable (i.e., a *name*), a new object is created into memory. New object ref count is set to 1, old is set to 0. Python memory manager performs garbage collection periodically, removing objects with ref count equal to 0

Demo:

- mutable containers

Dynamic typing

The type is not "embedded in the name", so a name can reference different object types during its lifetime. This is called dynamic typing

Remember: objects have type, names don't

Demo:

- dynamic typing
- inner values for containers

Object destruction

What happens to objects that are no longer reachable?

- Objects are never explicitly destroyed
- When they become unreachable they may be garbage-collected
- An implementation is allowed to postpone garbage collection or omit it (as long as objects not collected are not reachable)

Execution model - blocks

- Python programs are constructed from code blocks
- **Block**: a piece of program executed as a unit
  - module
  - function body
  - class definition
- each command typed interactively is a block
- a script file is a code block
- a module run as a top level script is a code block
- a string argument passed to the built-in functions `eval()` and `exec()` is a block

Each block is executed in an execution **frame**, which contains some administrative (debug) information and determines how execution will continue (e.g., next block)

Execution model - binding of names

The concept of block is important because it affects how names are resolved

If a name is bound in a block, it is a local name to that block (unless `nonlocal` or `global` are used)

Execution model - resolution of names

The **scope** defines the visibility of a name

A name is resolved using the nearest enclosing scope

E.g., the scope of a local name is the defining block

Demo:

- built-in vs global vs local namespace
- examples of name resolution
- `global` vs `nonlocal`

Data types - numbers

- Python provides several types to memorize numbers
  - some are built-in: integers (with unlimited precision!), floating point numbers, complex numbers, boolean numbers
  - others are defined by the standard library (e.g., fraction)
- Let's review how to initialize some of them

Integer literals

```
# binary
b = 0b110

# octal
o = 0o1237

# hexadecimal
x = 0xaef
```

Floating point literals

```
f = 3.14 # 3.14
f = 10. # 10.0
f = .001 # 0.001
f = 1e100 # 1e+100
f = 3.14e-10 # 3.14e-10
f = 0e0 # 0.0
f = 3.14_15_93 # 3.141593
```

Data types - containers

- Some data types are very important, because they can store collections of values
- They are called **containers**
- Containers can be built-in types or defined in the `collections` module in the Python Standard Library
- We start with `list` and `tuple` (more coming in the next lessons)

Sequences - supported operations

Sequences (such as `list` and `tuple`) support a number of operations:

Operation	Result
`x in s`	True if an item in s equals x
`x not in s`	...
`s + t`	concatenation of sequences s and t
`s * n`	equivalent to adding s to itself n times
`s[i]`	i-th item of s (0 is the origin)
`s[i:j]`	slice of s from i to j
`s[i:j:k]`	slice of s from i to j with step k
`len(s)`	length of s
`min(s)`	min item in s
`max(s)`	max item in s
`s.index(x)`	index of the first occurence of x in s
`s.count(x)`	total number of occurences of x in s

List

Very useful when we need to store a sequence of values, and don't know whether that sequence will grow in the future

- Mutable
- Expandable
- Sortable
- E.g., `l = [1,2...]`

Let's test some of the operators with the REPL

List - some functions

```
l = [1, 2, 3]
# append
l.append(6) # [1, 2, 3, 6]
# extend
l.extend([10, 11, 12, 13, 14]) # [1, 2, 3, 6, 10, 11, 12,
    13, 14]
# remove last item
l.pop() # [1, 2, 3, 6, 10, 11, 12, 13]
# get element at position
l[1] # 2
# slicing
l[2:6] # [3, 6, 10, 11]


# pop element at specific index (*bad usage*)
l.pop(2) # [1, 2, 6, 10, 11, 12, 13]
# insert at position (*bad usage*)
l.insert(2, 80) # [1, 2, 80, 6, 10, 11, 12, 13]
```

List - dynamic resizing

- We can extend a list with new items, but how can we keep accessing any position in constant time?
- Elements must be allocated in contiguous memory locations
- The answer is dynamic resizing
- Let's see how it works on the blackboard

References:
github.com/python/cpython/blob/main/Objects/listobject.c#L60

```
newsize = ((oldsize + oldsize // 8) + 6) & (~3)
```

- the bitwise operation sets `newsize` to a multiple of 4
- approximately 12.5% increase in size

List - dynamic resizing

When an append is performed, there are two cases:

- the list has still capacity
  - a single append is performed, so $O(1)$ operations
- the list no longer has capacity
  - approx. 12.5% more space is allocated
  - all previous elements are copied, so $O(n)$ operations
  - the append is performed, so $O(1)$ operations

To determine the complexity of this operation we need to perform an **amortized analysis**

List - dynamic resizing

- We perform a sequence of n append operations
- Dynamic array in Python grows approximately by $k = 9/8$
- Let's assume $k = 2 > 9/8$ (size doubles, but less frequently and at the end we will waste a bit more memory)
- Extra-precision is not that useful with asymptotic behavior

With n calls to the `append()` function, we have performed:

- n appends each taking $O(1)$, so $O(n)$
- $1 + 2 + 4 + \ldots + n/2 + n$ operations to copy elements, so $O(2n) = O(n)$

Summing up $O(n) + O(n)$ for n calls to the `append()` function, each taking $O(1)$ according to the analysis

List - dynamic resizing

There is another approach to demonstrate it

Assume you have performed the $n\text{-}th$ reallocation, that means you have performed about k^n append operations

Let's determine the number of copies performed C

$$C = k^{n-1} + k^{n-2} + \ldots + 1$$

Now, let's multiply by k and subtract C

$$kC - C = k^n - 1$$

List - dynamic resizing

$$C(k-1) = k^n - 1$$

$$C = \frac{k^n - 1}{k - 1}$$

Amortized number of copies:

$$\frac{C}{k^n} = \frac{k^n - 1}{k - 1} \cdot \frac{1}{k^n}$$

Result is still $O(1)$

List - problems with `pop()` and `insert()`

- In both cases, we may shift up to n elements to perform the operation
- This is not ideal from a performance perspective
- We will study other structures to solve the problem

List - complexity chart

Operation	Amortized complexity
`append()`	$O(1)$
`extend([1, ..., k])`	$O(k)$
`pop()`	$O(1)$
`pop(idx)` with idx != last	$O(n)$
`insert(idx, elem)` with idx != last	$O(n)$
`in` operator	$O(n)$
`index()`	$O(n)$

Tuple

- Immutable, memory efficient
- No extra-space allocated compared to `list` (less properties)
- Best type to return values from a function

```
# creation
t = (1,2,3)

# positional access
t[1] # 2

# slicing
t[0:2] # (1, 2)

# error, tuple object does not support value update
t[0] = 5
```

Tuple - memory tip

- Python can reuse the space allocated for previously created and deleted tuples to reduce memory consumption
- The following *should* print the same identity twice

```
t1 = (1,2,3)
print(id(t1))

t1 = "new value assignment"

t2 = (4,5,6)
print(id(t2))
```

Range

- An immutable sequence of numbers commonly used for looping
- Integer input:
  - start (optional, default is 0)
  - stop
  - step (optional)

```
list(range(10)) # [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

list(range(0, 10, 3)) # [0, 3, 6, 9]

for i in range(3):
    print(i)
# 0
# 1
# 2
```

Enumerate

Useful to simplify loops

```
letters = ("a", "b", "c",)

for idx, l in enumerate(letters):
    print((idx, l), end=" ") # (0, 'a') (1, 'b') (2, 'c')
```

List comprehensions

- Not to be confused with the `list` structure
- A concise way to create lists
- Useful to:
  - filter data
  - quickly modify the format of the entries in a container

```
letters = ("a", "b", "c",)
unicode_letters = [ord(l) for l in letters if l != "c"]
unicode_letters # [97, 98]
```

Demo:

- efficiency of list comprehension vs append

Functions

Demo:

- arguments separated by comma
- default arguments
- keyword arguments
- variable number of arguments
  - keyword
  - non-keyword
- Call-by-Object

Exercises

- Ex. 1: `bits_to_flip`
  - You are given two positive integer numbers a and b. Find the number of bits to flip to convert int a to int b.
- Ex. 2: `gray_code`
  - Gray code is a binary numeral system where two successive values differ only in one bit. You are given a number, compute its Gray code.
- Ex. 3: `condensed_ranges`
  - You are given a list of tuples representing ranges. Condense the ranges.
- Ex. 4: `max_subarray`
  - You are given a list of integer numbers. Find the sub-list with the largest sum and return the sum.
- Ex. 5: `list_of_logs`
  - You are given a list of integer numbers `l1`. Return a new list `l2` containing the base 2 logarithm only for even numbers in `l1`.

Python fundamentals - pt.2

Dario Facchinetti

Recap

In the last lesson we have studied the fundamentals of Python programming:

- execution environment
- memory model
- basic types
- containers (`list` and `tuple`)

Topic

Today we will study:

- `str`
- `set` vs `dictionary`
- set/dictionary comprehension
- stdin and stdout
- other functions useful in competitive programming

String

A sequence of characters (Unicode Points) enclosed in single or double quotes

- `"this is a string"`
- `'this is another'`
- a double quoted string literal can contain single quotes (`"I'm fine"`) and likewise single quoted string can contain double quotes
- multi-line strings: `"""` or `"yet \` (next line) `another one"`
- escape with \ (eg. `"\n"`, `"\t"`)

Demo

String immutability

Strings are immutable: they cannot be changed after they are created

This means that if we try to concatenate two strings, a third one will be created

String characters

- String can be accessed with the `[]` notation
- zero-based indexing
- `IndexError` is raised when we go out of range
- buil-in function `len()` is used to extract the string length
- similarly to lists, slices can be used

Demo:

- slice

Print objects

- Python provides functionality to "print objects"
- `f""` or the method `str.format(...)` can be used

Demo

String methods

See docs.python.org/3/library/stdtypes.html#string-methods

`str.capitalize()`, `str.count(sub)`,
`str.encode(encoding='utf-8')`, `str.endswith(suffix)`,
`str.find(sub)`, `str.index(index)`, `str.isalnum()`,
`str.isalpha()`, `str.isdigit()`, `str.isdecimal()`,
`str.islower()`, `str.isupper()`, `str.join(iterable)`,
`str.lower()`, `str.lstrip(chars)`, `str.split(sep)`,
`str.startswith(prefix)`, `str.strip(chars)`,
`str.zfill(width)`

Demo:

- how to find info from the doc, `find()` vs `index()`
- `join()`

String exercise

- Ex. 1: `sum_strings`
  - You are given two non-negative integers, a and b represented as string. Return the sum a + b as string.
    Do not use numbers to handle large integers, also, do not convert strings to integers directly.
    Test the code with different inputs.
  - E.g.: `a="888"`, `b="111"`, `r="999"`

Sets

A **set** object is an unordered collection of distinct *hashable* objects

What does *hashable* mean?

- the object has a hash value which never changes during its lifetime (a `__hash__()` method is defined)
- generally, mutable objects are not hashable, immutable objects are

Demo:

- `hash()`
- `__hash__()`

Sets

There are two built-in set types, `frozenset` and `set`

`set`:

- mutable with `add()` and `remove()`
- no hash value is defined

`frozenset`:

- immutable, contents cannot be altered after it is created
- hashable

Demo:

- `frozenset` vs `set`
- set operations

Set comprehension

Returns a set based on iterables

Similarly to list comprehensions, we have a name, an iterable-based expression and a predicate

```
{i for i in [1,2,3,4]} # {1, 2, 3, 4}
type({i for i in [1,2,3,4]}) # <class 'set'>

{i**2 for i in range(10) if i % 2 != 0} # {1, 9, 81, 49, 25}
```

Dictionary - `dict`

- A mapping between hashable values and arbitrary objects
- Mutable

```
d = {"ferrari": 2, "red bull": 1, "mercedes": 3}
type(d) # <class 'dict'>
d # {'ferrari': 2, 'red bull': 1, 'mercedes': 3}
```

Dictionary operations

Operations:

- Store a value with a given key
- Extract a value given the key
- Delete a key:value pair

Remember:

- If you store using a key that is already in the dictionary, the old value associated with that key is replaced
- Extracting a value using a non-existent key raises an error

Demo:

- basic operations
- `len()`, `get()`, `list()`, `copy()`, `clear()`

Dictionary - iteration

There are several ways to iterate over the elements in a dictionary

```
for key in d:
    print(d[key])

for key in d.keys():
    print(d[key])

for value in d.values():
    print(value)

for (k, v) in d.items(): # parenthesis are optional
    print(k, v)
```

Dictionary - update

Some useful functions to update the values in a dictionary

```
# todo: read documentation with help(d.setdefault)
d.setdefault("williams", 10)
d # {'ferrari': 2, 'red bull': 1, 'mercedes': 3, 'williams':
    10}

# update with new values (overwriting existing keys)
d.update([("ferrari", 12), ("mclaren", 5)])
d # {'ferrari': 12, 'red bull': 1, 'mercedes': 3,
#    'williams': 10, 'mclaren': 5}

# merge two dictionaries
d1, d2 = dict([("a", 1), ("b", 2)]), dict([("c", 3)])
d1 | d2 # {'a': 1, 'b': 2, 'c': 3}

# merge and update
d1, d2 = dict([("a", 1), ("b", 2)]), dict([("c", 3)])
d1 |= d2
d1 # {'a': 1, 'b': 2, 'c': 3}
```

Dictionary - pop

There are two ways to pop an element from the dictionary

```
# pop by key, raises KeyError when there is no key and
# default value is not specified
d = dict([("ferrari", 2), ("red bull", 1), ("mercedes", 3)])
d.pop("alpha tauri") # KeyError
d.pop("alpha tauri", -1) # -1
d.pop("ferrari") # 2

# remove and return a (key, value) pair from the dictionary
# return value is in LIFO order
d.popitem() # ('mercedes', 3)
d.popitem() # ('red bull', 1)
d.popitem() # KeyError, empty dictionary
```

Dictionary comprehension

There is a quicker way to instantiate a dictionary

```
tracks_km = {"singapore":5.063,
             "monza":5.793,
             "suzuka":5.807,}
tracks_km
# {'singapore': 5.063, 'monza': 5.793, 'suzuka': 5.807}
type(tracks_km) # <class 'dict'>

yards_per_km = 1093.61
tracks_yd = {t:round(l*yards_per_km)
             for t,l in tracks_km.items()}
tracks_yd
# {'singapore': 5537, 'monza': 6335, 'suzuka': 6351}
```

Dictionary - complexity

What makes a dictionary so special? Insertion and lookup happen in almost constant time

Dictionary exercise

- Ex. 1: `counter`
  - Implement a Counter, or rather count the occurrences of each character in a string. Note the ordering in the following example.

```
from collections import Counter
s = "hollywood"
Counter(s)
# Counter({'o': 3, 'l': 2, 'h': 1, 'y': 1, 'w': 1, 'd': 1})
```

Zip vs map

There are two functions that come handy in competitions:

- `zip()`, returns a list of tuples, where the i-th tuple contains the i-th element from each of the argument sequences or iterables
- `map()`, applies a function to every item of an iterable and returns a list of the results

Demo

Reading cmd line arguments

Python provides built-in support for reading arguments passed to a script from the command line.

```
python args.py 3 a b c
```

```
import sys
print(sys.argv)
# ['args.py', '3', 'a', 'b', 'c']
```

Demo

Reading from stdin

`sys.stdin` is the file object used by the interpreter for interactive input, it is easy to use it to read data from file

```
python stdin.py < input.txt
```

```
import sys

for i, line in enumerate(sys.stdin):
    arr = [int(x) for x in line.split()]
    print("line {}, sum {}".format(i, sum(arr)))
```

Demo

Writing to stdout

Similarly, `sys.stdout` is the file object used by the interpreter for interactive output

```
python solver.py < input.txt > output.txt
```

```
import sys

for i, line in enumerate(sys.stdin):
    arr = [int(x) for x in line.split()]
    sys.stdout.write(str(i) + " ")
    sys.stdout.write(str(sum(arr)) + "\n")
```

Demo

Where to learn more

- The Python language reference:
  https://docs.python.org/3/reference/
- The Python standard library:
  https://docs.python.org/3/library/
- Exploring the internals:
  https://devguide.python.org/internals/exploring/

What's next

Efficient approaches to sort numbers

Coding strategies

Dario Facchinetti

Unit 2

We learned Python's syntax and memory model!

Let's practice with some coding strategies:

- sliding windows
- two pointers
- fast and slow pointers

We will focus on strings and lists

Sliding windows

A technique that aims to reduce the use of nested loops

It permits to traverse a sequence in a ‘clever way’, without performing unnecessary (extra) work on portions of it

Sliding windows - example

The most famous problem that can be solved efficiently with sliding windows is probably the following:

```
Find the longest substring without repeating characters
in a given string
```

Demo

Two pointers

Sometimes it is convenient to work on a data structure with two pointers that move in opposite directions (e.g., one pointer traverses the sequence from the beginning, the other one from the end)

Termination happens when the two pointers swap positions (e.g., the right pointer is located to the left of the left pointer)

This strategy is particularly useful when a property is set for a sequence (e.g, the sequence is sorted), and it usually permits to solve a problem with a single iteration

Two pointers - example

The following is another popular problem

```
You are given an integer sequence of length n. Each i-th
element in the sequence represents the vertical height of
a wall. Find the maximum quantity of water that can be
stored between two walls (the two walls can be at any
distance). Assume that walls have no volume (i.e., they
are just vertical lines).
```

Demo

Fast and slow pointers

When we are solving a problem we may find natural to look for the best data structure, filter some items, measure a distance, compute a score, cumulate a counter, sort the data, and so on

But sometimes what is needed is just **creativity** (and you will save a lot of memory)

Imagine you are stuck in a labyrinth and the only way out is to keep walking following a red thread. You have forgotten your camera and you cannot take any picture of the environment around you, how can you tell if you have already crossed a certain area of the labyrinth? Well, if you walk by yourself is difficult, but maybe if there is a friend who can run...

Fast and slow pointers - example

A popular algorithm that uses this strategy to detect cycles in a list or a graph is the Hare-Tortoise algorithm (or Floyd's cycle finding algorithm)

```
You are given a sequence 'l' of integer numbers. Each number
in the sequence stores the next position you can jump to.
Position 'None' terminates the sequence. Write a program to
detect cycles in the sequence. There is a cycle if you can
jump to the same position twice or more.

Sequences are built so that you will not trigger an out of
bound exception.
```

Demo

Sorting

Dario Facchinetti

Unit 3

- Sorting
  - Selection sort
  - Insertion sort
  - Merge sort
  - Quick sort
  - Heap sort
  - Radix sort
- Hybrid approaches

Topic

Arrange the items in a sequence ordered by some criterion

We will study and implement the most used approaches

Bubble sort

The most trivial algorithm:

1. loop through the sequence element by element
2. compare each element with the following
3. swap if needed

Bubble sort - complexity

Time:

- worst $O(n^2)$
- best $O(n^2)$
- average $O(n^2)$

Space: $O(1)$

Selection sort

Selection sort divides the input sequence into two parts:

- a sorted subsequence built from left to right
- the subsequence of the remaining unsorted elements

Iterate until the unsorted sequence is not empty:

1. select the smallest element
2. add such element as the last one in the sorted subsequence

Selection sort - complexity

Time:

- worst $O(n^2)$
- best $O(n^2)$
- average $O(n^2)$

Space: $O(1)$

Selection sort - pros and cons

Pros:

- no extra memory needed

Cons:

- at each iteration it has to find the smallest element (this takes $O(n)$ time)

Insertion sort

Selection sort builds the sorted array one element at a time:

1. remove one element from the input sequence
2. find the location where it belongs in the sorted list (by swapping to the right elements that are greater than it)
3. insert the element

Insertion sort - complexity

Time:

- worst $O(n^2)$, when the sequence is sorted decreasingly, it requires to swap all the elements, at each iteration
- best $O(n)$, no swaps performed if the sequence is already ordered
- average $O(n^2)$

Space: $O(1)$

Insertion sort - pros vs cons

Pros:

- very simple implementation
- very efficient for sequences that are *substantially* sorted
- stable, as it does not change the relative order of elements with the same value
- no extra memory needed
- *online*, it means that it can sort a sequence as it receives it

Cons:

- too many swaps in the average case

Merge sort

Merge sort applies the *divide and conquer* approach:

1. an unsorted sequence is split into two unsorted subsequences
2. merge sort is applied to both the subsequences
3. the two sorted subsequences are merged into a sorted sequence

Tips:

- a sequence with one element is intrinsically sorted
- step 3 is not computationally expensive

Merge sort - complexity pt.1

Space: $O(n)$

What about time?

Merge sort - complexity pt.2

Recurrence relation:

$$T(n) = 2T(n/2) + n$$

We need to use the **Master theorem**

Master theorem

The recurrence relation

$$\begin{cases} T(n) = aT(n/b) + cn^k \\ T(1) = c \end{cases}$$

where a, b, c and k are constants solves to:

1. $T(n) \in \Theta(n^k)$ if $a < b^k$
2. $T(n) \in \Theta(n^k \log n)$ if $a = b^k$
3. $T(n) \in \Theta(n^{\log_b a})$ if $a > b^k$

Merge sort - complexity pt.3

Recurrence relation: $T(n) = 2T(n/2) + n$

$\rightarrow a = 2,\ b = 2,\ c = 1$ and $k = 1$ so $\Theta(n^k \log n)$

Time:

- worst $O(n \log n)$
- best $O(n \log n)$
- average $O(n \log n)$

Merge sort - pros vs cons

Pros:

- stable
- it reduces the number of comparisons

Cons:

- requires extra space to store the intermediate results
- the entire sequence goes through the whole process

Quick sort

Quick sort takes a very interesting approach:

- select a pivot from a sequence
- move all the elemets smaller than the pivot to its left
- apply Quick sort to the subsequence of elements to the left and to the right of the pivot

Quick sort - complexity pt.1

- We have used two methods to implement the algorithm: `partition` and `quicksort`
- Both methods don't perform allocations, but how many calls to `partition` does `quicksort` perform?
- It depends on the position of the pivot, and in the worst case, each call to `partition` could return two subsequences long 1 and $n - 1$, respectively

Quick sort - complexity pt.2

Thus, in the **worst case**, the algorithm performs n calls to `partition`, each taking $O(1)$ space and $O(n)$ time

Worst case complexity:

- time $O(n^2)$
- space $O(n)$

Quick sort - complexity pt.3

In the **best case**, the algorithm determines partitions which are $n/2$ elements long

Best case complexity:

- time $O(n \log n)$
- space $O(\log n)$

Quick sort - complexity pt.4

Average case complexity:

- time $O(n \log n)$
- space $O(\log n)$

The proof is not immediate, and is reported on the textbooks

Quick sort - complexity pt.5

- Remember that space complexity depend on the implementation
- There are strategies like in-place partitioning and tail recursion that can be used to reduce the amount of memory required

Reference:


```
Robert Sedgewick.
Implementing Quick sort programs.
Commun. ACM 21, 10 (Oct. 1978), 847  857
```

https://doi.org/10.1145/359619.359631

Quick sort - pros and cons

Pros:

- slightly faster than Merge sort and Heap sort for randomized data

Cons:

- requires extra memory

Heap sort

A (binary) heap is a binary tree with two additional properties:

- *shape property*, a heap is a complete binary tree in which all the levels (except possibly the deepest) are filled
- *heap property*, the key stored in each node is either greater than or equal to (in the case of max heap) or less than or equal to (in the case of min heap) the keys in the node's children

Heap sort:

1. Transform the unsorted sequence into a max heap
2. Key extraction (iterative step):
    - move the leftmost key (max key) to the rightmost position
    - shorten the max heap by one key and *sift*

Heap sort - complexity

Building the max heap takes $O(n \log n)$ time (we will study more efficient ways for building the heap later on), extracting all the keys also takes $O(n \log n)$ time

Time:

- worst $O(n \log n)$
- average $O(n \log n)$
- Heap sort takes $O(n)$ when all the keys are the same, when instead the keys are distinct, the best time complexity is still $O(n \log n)$

Space: $O(1)$

Heap sort - pros vs cons

Pros:

- no extra space
- performs well in the best, average and worst scenarios

Cons:

- unstable
- a bit slower than a good Quick sort implementation

Radix sort

A non-comparative sorting algorithm

- It distributes the elements into buckets according to their radix
- The process is repeated for each of the digits

Radix sort - complexity

(Implemented version)

Time: $O(nd)$

Space: $O(n + d)$

Where d is the number of digits

Radix sort - pros vs cons

Pros:

- it can be applied to any data that can be sorted lexicographically (integers, words, etc.)
- useful when we can match a key to a string

Cons:

- negative numbers must be handled explicity
- it takes more memory compared to algorithms that perfom in-place sorting

Summary

We have implemented the most known sorting approaches, do standard libraries use them?

Yes, but sorting functions from standard libraries adopt more sophisticated approaches

To improve performance, a common strategy is two leverage two or more algorithms that solve the same problem, and combine them into a **hybrid algorithm** (taking advantage of their best features)

Hybrid approach - example

Quick sort is efficient if the size of the input is very large, but Insertion sort is more efficient than Quick sort for short, partially sorted sequences (and it also consumes less memory)

Let's combine them into a hybrid algorithm!

Additional exercise: try to implement a naive version of Timsort, a hybrid sorting algorithm derived from Merge sort and Insertion sort

Architecture and other optimizations

There are many other aspects that play an important role

Architectures are not ideal, and simple **optimization rules** can greatly reduce the overall number of operations that need to be performed to sort a sequence

We also must consider that often sequences contain consecutive ordered elements, or **runs**

Let's have look at Python's approach!
https://github.com/python/cpython/blob/main/Objects/listsort.txt

Summary

We have implemented some of the most popular sorting techniques

Some of them minimize memory consumption, other reduce the number of swaps and comparisons

We have also discussed their limits, and combined them to get better implementations

However, there are many other sophisticated and powerful approaches to explore!

Linked list

Dario Facchinetti

Unit 4

- Data structures:
  - linked list

Topic - Linked list

A linear collection of data elements

Implementations usually refer to the elements as *nodes*

Each node contains:

- the data item (sometimes called *key*)
- a reference to the next node

Singly linked list

We will focus on the **singly linked list**, but consider there are also other variants like:

- doubly linked list
- circular linked list

Properties

A linked list is characterized by **linear time access**, since we need to traverse the structure following the chain of references to get a particular key

Although access time is not constant, it permits to insert and remove elements without re-allocating (part of) the structure

Also notice that storing the nodes in non-contiguous memory permits to save space, but implies worse locality (cache)

Methods

We are going to implement the following methods:

- `append(key)`, to append a key to the end of the list
- `prepend(key)`, to add a key at the beginning of the list
- `get(index)`, to return the key at a given index (or position)
- `set_at(key, index)`, to add a key at a given index
- `delete_at(index)`, to delete the key at a given index
- `reverse()`, to reverse the list in-place (no new allocations)
- `detect_cycle()`, to detect a cycle (fast and slow strategy)

Home practice

Implement `delete_last_k(k)`, which removes the last `k` nodes from the sequence

Requirement:

- perfom only a single scan on the list

Home practice - hint

Use two pointers kept `k` positions distant from each other

Queue and Stack

Dario Facchinetti

Unit 4

- Data structures:
  - Queue
  - Stack
  - Python's `deque`

Topic - Queue and Stack

Both structures are a collection of entities that are maintained in a sequence

Based on the implementation of the `push` and `pop` methods:

- FIFO $\rightarrow$ queue
- LIFO $\rightarrow$ stack

Keep in mind that internal ordering may be different than that of the sequence of `push` operations, as we can implement queues based on priorities (e.g., pop the entity with the highest priority)

Queue

The simplest variant:

- push a new element to the beginning of the sequence
- pop the element at the end of the sequence

Let's implement it

Time complexity for our solution:

- `push()`, $O(1)$
- `pop()`, $O(1)$
- `len()`, $O(1)$

Stack

We will:

- push a new element to the beginning of the sequence
- pop the element at the beginning of the sequence

Let's implement it

Time complexity for our solution:

- `push()`, $O(1)$
- `pop()`, $O(1)$
- `len()`, $O(1)$

deque

There is no need to re-implement everything from scratch when prototyping a new application

The `collections` module provides `deque`, a list-like container with fast push and pop on either end

A `deque` is a generalization of stack and queue (the name stands for 'double-ended queue'), and `append` and `pop` operations take approximately $O(1)$ time on both ends

It is also thread-safe

Demo

deque complexity

Operation	Average Case	Amortized Worst Case
`copy`	$O(n)$	$O(n)$
`append`	$O(1)$	$O(1)$
`appendleft`	$O(1)$	$O(1)$
`pop`	$O(1)$	$O(1)$
`popleft`	$O(1)$	$O(1)$
`extend`	$O(k)$	$O(k)$
`extendleft`	$O(k)$	$O(k)$
`rotate`	$O(k)$	$O(k)$
`remove`	$O(n)$	$O(n)$
`len`	$O(1)$	$O(1)$

Home practice

Implement a Queue and a Stack using `deque`

Heap

Dario Facchinetti

Unit 4

- Data structures:
  - Heap

Topic - Heap

We have already studied this structure in the Heap sort

It is an *almost-complete* tree that satisfies the **heap propriety**:

- the key stored in each node is either greater than or equal to (in the case of max heap) or less than or equal to (in the case of min heap) the keys in the node's children

A backing array is used to store the tree

Heap

There are two variants:

- min heap
- max heap

We will implement just one of them, as in general the other variant can be constructed negating the keys before they are inserted and after they are extracted

Interface

Interface:

- `heapify(keys)`, to construct in-place the heap in $O(n)$ time
- `validate()`, to test the heap propriety in $O(n)$ time
- `push(key)`, to insert a new key in $O(\log_2 n)$ time
- `pop()`, to extract the max key in $O(\log_2 n)$ time

Let's implement a max heap

Usage

A heap is particularly useful in the following situations:

- we have to work with k entities (e.g., numbers, sets, lists) over n
- we have *priorities*
- we need to repeatedly insert/extract elements from a pool

heapq

There is no need to implement a heap from scratch when solving a problem

Python provides this function within the `heapq` module

Let's explore the API:

- `heapify`
- `heappush`
- `heappop`
- `merge`
- `nsmallest`
- `nlargest`

Priority queue

A data structure similar to a queue or a stack, however:

- a *priority* is associated with each element in the queue
- the element associated with the highest priority is always extracted first

Let's implement a priority queue using a heap

Home practice

Exercise 1

```
You are given a list of integers. Return the k most frequent
elements.
```

Exercise 2

```
You are given a list of N integers and an integer k. Return
the k-th largest integer in the list. You must solve the
problem in O(N) time.
```

Binary tree

Dario Facchinetti

Unit 4

- Data structures:
  - Binary tree

Topic - Binary Tree

A tree structure in which each node has at most two children

Trees are useful to:

- perform search operations
- implement hierarchical structures such as filesystems
- take decisions
- compress resources

An extension called k-tree also exists

Structure

Each node stores:

- a key
- a pointer to the left child
- a pointer to the right child
- optional information (e.g., number of occurences for the key, node positioning info)

An ordering property defines the position of each key:

- to the left of the current key (i.e., subtree rooted at the left child) there are values preceding it
- to the right of the current key there are values succeeding it

Key lookup

Key lookup is straightforward:

1. is the key stored in the current node?
   $\rightarrow$ success
2. is the key greater than the value stored in the current node?
   $\rightarrow$ keep searching the subtree rooted at right child (step 1)
3. is the key lower than the value stored in the current node?
   $\rightarrow$ keep searching the subtree rooted at left child (step 1)
4. is the current node a leaf?
   $\rightarrow$ fail

In this examples we assumed the keys were integers

Interface

Interface:

- `insert(key)`, to insert a new key
- `search(key)`, to test the presence of a key
- `in_order()`, `pre_order()`, and `post_order()` visits

Let's implement a binary tree

Height

Height is defined as the length of the longest chain that connects the root node to a leaf

Implementation

Complexity

Operations are efficient only when the tree is **balanced**

In a balanced tree, for each node:

- the height of the left and right subtrees differ at most by 1
- the left and right subtrees are balanced

When the tree is balanced we can reach each leaf in $O(\log_2 n)$, where n is the number of nodes in the tree

When the tree collapses to a sequence a linear scan is required

Full binary tree

In a **full** tree each node has either no child or 2 children

Implementation

Exercise - reverse

```
Write a function to reverse the keys in the binary tree.
The function must not allocate new nodes.
```

Exercise - LCA

```
Given two keys, write a function that returns the Lowest
Common Ancestor (LCA). The two keys are guaranteed to be
stored in the tree.
```

What's next

In the next lesson we will implement a self-balancing search tree

We will also implement the `delete(key)` operation

AVL tree

Dario Facchinetti

Unit 4

- Data structures:
  - AVL tree

Topic - Adelson-Velsky and Landis Tree

A **self-balancing** binary search tree (proposed in 1962)

For each node, the heights of the left and right child subtrees differ at most by one

How? If at any time the heights of two subtrees differ by more than one, **rebalancing** is done

Complexity

In a balanced tree `insert`, `delete` and `search` take $O(\log_2 n)$ time in both the average and worst cases

Remember that n is the number of nodes in the tree

Rotations

One or more **rotations** may be required after `insert` or `delete`

A rotation alters the shape of the tree without compromising the order of the keys

Demo:

- examples of rotation on the blackboard

Interface

Interface:

- `insert(key)`, to insert a new key
- `delete(key)`, to delete a key
- `search(key)`, to test the presence of a key

Auxiliary functions:

- `height(tree)`, `successor(root)`, `balance_factor()`
- `rot_right(root)`, `rot_left(root)`

Let's implement an AVL tree

Graph

Dario Facchinetti

Unit 4

- Data structures:
  - Graph

Topic - Graph

A finite set of **vertices**, together with a finite set of **edges** to connect them

- Vertices are often called *nodes* or *points*
- Edges are often called *links* or *lines*
- Notation:
  - V denotes the (number of) vertices
  - E denotes the (number of) edges
- Edges can be represented as pairs of vertices:
  - ordered pairs are used in a **directed graph**
  - unordered pairs are used in an **undirected graph**
- Values can be associated to each edge:
  - symbolic label
  - cost, weight, capacity, length, etc.

Visualization

Let's draw examples of graph on the blackboard

- vertices vs edges
- directed vs undirected
- example of values

Applications

Graph is a very important structure, its applications include:

- computer networks
- geolocalization and navigation
- logistics
- social networks
- queries and computation
- and many more

Common representations - pt.1

Adjacency list: vertices are stored as records, and every vertex stores a list of adjacent vertices

- $A \to B \to C \to$ `None`
- $B \to A \to$ `None`
- $C \to$ `None`

Slow to remove vertices and edges (linear scans)

Common representations - pt.2

Adjacency matrix: 2D matrix in which each row represents the source vertex and the column the destination

	A	B	C
A	0	1	1
B	1	0	0
C	0	0	0

Slow to add or remove vertices (matrix re-allocation)

Common representations - pt.3

Incidence matrix: 2D matrix in which each row represents a vertex and the colum represents an edge; each entry indicates the incidence relation between a vertex and and edge

	E_{1A}	E_{1B}	E_2
A	1	-1	1
B	-1	1	0
C	0	0	-1

Again, slow to add or remove vertices (matrix re-allocation)

Exploring the graph

There are two ways to explore or visit a graph:

- Breadth-First Search (BFS)
- Depth-First Search (DFS)

BFS first explores the vertices at a given depth (i.e., the nearest to the source), while DFS takes the opposite approach

Implementation (with `deque`) and complexity analysis

Home exercise - Number of islands

```
You are given an R x C grid which represents a map of '1's
(land) and '0's (water).
Return the number of islands in the map.

A valid island is formed by connecting adjacent lands
horizontally or vertically. Also, assume that all four
edges of the grid are all surrounded by water.

Example of map with 2 islands:

|0|1|0|1|
|1|1|1|0|
|0|0|1|0|
```

Acyclicity

A **cycle** is a non-empty trail in which only the first and last vertices are equal

Let's implement a program that checks whether there are cycles in a directed graph. If a cycle is detected, we will print to stdout all its vertices

Shortest path

Dijkstra's algorithm (1956) permits to find the shortest path between two nodes in a weighted graph

We are going to implement a solution that finds the shortest path to each node from a given source

Remember that you can use a Fibonacci heap to further optimize the algorithm

Home exercise - Fibonacci heap

```
Find information about the Fibonacci heap.
```

Connected components

A **connected component** (or simply *component*) of an undirected graph G is a subgraph in which each pair of vertices is connected with a path

Bridges

You are given an undirected graph G

A **bridge** is defined as an edge which, when removed, makes the graph disconnected (i.e., it increases the number of connected components)

Let's implement a program to find all the bridges in an undirected graph

Home exercise - Bridges, iterative solution

```
Try to translate the recursive version studied in class to
an iterative version.

After, compare your solution with the teacher's.

Question: is there a systematic way to translate a
recursive method into an iterative one?
```

Strongly connected components

A directed graph G is **strongly connected** if there is a path between all pairs of vertices

A **strongly connected component** of G is a subgraph in which no additional edges or vertices (from G) can be included without breaking its property of being strongly connected

Home exercise - Strongly connected components

Try to implement an algorithm to find all the strongly connected components in a directed graph.

If you cannot find the solution in 20 minutes, don't worry, just implement the Kosaraju-Sharir's algorithm.

Home exercise - Spanning tree

```
Learn the concept of Spanning Tree.

Learn the concept of Minimum Spanning Tree.

Implement the Kruskal's algorithm to find the Minimum
Spanning Tree given a list of weighted edges.
```

Home exercise - Topological sort

```
Learn the concept of  topological sort or topological
ordering of a directed graph.

Implement the Kahn's algorithm.
```

Sparse table

Dario Facchinetti

Unit 4

- Data structures:
  - Sparse table

Motivation

Problem statement:

1. You are given a sequence of numbers $S = s_1, s_2, \ldots, s_{n-1}$
2. You are also given an interval $[i_l, i_r]$

Question:

- find the minimum number in S within the interval $[i_l, i_r]$, in other words answer minimum queries given a range

Let's discuss the problem in detail

Brute force solution

The simplest solution

```
min(S[left:right+1])
```

but given Q queries, this approach takes $O(QN)$ time

Idea

What else can we do?

Precompute some answers and reuse them at query time

But which ones? This looks like a combinatorial problem

Demo

Intuition

Precompute all answers for ranges with power of two lengths

Why? Each range can be splitted into ranges with power of two lengths, and if we already knew the answer for such ranges, then we could just combine the answers to solve the query

Demo

Sparse table

Sparse table implementation

Complexity

Precomputation takes $O(N \log N)$ time and $O(N \log N)$ space (i.e., there is trade-off between time and space used)

However, subsequent queries can be answered in almost $O(1)$

Problems

The approach we studied works well with idempotent functions (e.g., min, max)

But we need to be careful with other kinds of query (e.g., sum)

The reason is that the precomputed ranges we use to answer a query may be **overlapping**

Immutability

A drawback of the sparse table is that it can only be used with **immutable sequences** (e.g., no update between two queries)

After an update, the entire table has to be recomputed

Home exercise - Segment tree

```
Learn the concept of segment tree.

Use a segment tree to answer sum queries.
```

Home exercise - Binary lifting

```
Learn the concept of binary lifting.

You are given a tree T. Given queries of the form (n, k)
where n is a node in T and k is an integer, find the
k-th ancestor of n in T.
```

Trie

Dario Facchinetti

Unit 4

- Data structures:
  - Trie

Topic - Trie

A **Trie** (or prefix tree) is a k-ary search tree used to locate keys from within a set

It is typically used with string keys, in that case, the links between nodes are created based on the sequence of characters

Example: plane, planisphere, planet, planimetry

```
                |-e-t-r-[y]
            |-m
            |-s
        |-i   |-p-h-e-r-[e]
        |-[e]
p-l-a-n   |-[t]
```

Search

A DFS visit is used to search a key

When N is the number of keys stored in the Trie, and L is the length of the longest key (i.e., each key has at most L characters), each key lookup takes $O(L)$ time

Usage

A Trie can be used to:

- implement a vocabulary
- generate efficiently completion lists
- implement lexicographic sort

Construction

We are given a vocabulary and a prefix string

Let's find all the words in the vocabulary that start with the given prefix

Implementation

Home exercise - break words

```
You are given a string S and a dictionary D. Return True if
S can be divided into a sequence of space-separated
words from the dictionary.

Example 1:
  - D: {base, ball}
  - S: baseball
  - returned value: True

Example 2:
  - D: {fish, chips, tartar, sauce}
  - S: chipsauce
  - returned value: False
```

Home exercise - update and delete

```
Start from the implementation of Trie studied in class.
Implement the update and delete methods.
```

Disjoint-set

Dario Facchinetti

Unit 4

- Data structures:
  - Disjoint-set

Motivation

Problem statement:

1. You are given a set of elements $E = \{e_1, e_2, \ldots, e_n\}$
2. You are also given a set of connections (e_i, e_j), each representing a link between two elements
3. Given an element e_k, the partition P_{e_k} is a subset of E that stores all the elements linked to e_k

Question:

- given two elements e_1 and e_2, find whether they belong to the same partition (i.e., a link exists)

Let's discuss the problem in detail

A different perspective

Instead of using partitions (that are expensive to find and merge), why don't we use a representative for each element?

Idea:

1. use nodes instead of elements
2. use trees instead of sets to group the elements
3. the representative of each partition (or the partition identifier) is the root of the tree

$\rightarrow$ merge operations can be be performed rearranging pointers

Demo

Topic - Disjoint-set

Also called **union-find**, or **merge-find set**

A data structure that stores a collection of disjoint sets

It provides three operations:

- `make-set`, to create/add new sets
- `union`, to merge two sets
- `find`, to find the set representative given an element

Let's implement a basic version

Merge strategy - pt.1

What is the best strategy to merge large partitions/trees?

$\rightarrow$ **always merge partition representatives** (i.e., root nodes)

Demo

Merge strategy - pt.2

What is the best strategy to merge large partitions/trees?

obs: `find`-ing the partition representative is a non-costant operation (i.e., it depends on the height of the tree)

Demo

Union by rank

Rank

- an upper bound of the height of a tree ^^

Union-by-rank

- always merge the shorter tree into the taller tree

```
^^ rank is an upper boud of the height of the tree when
union by rank is the only optimization introduced
```

Demo

Union by rank - implications

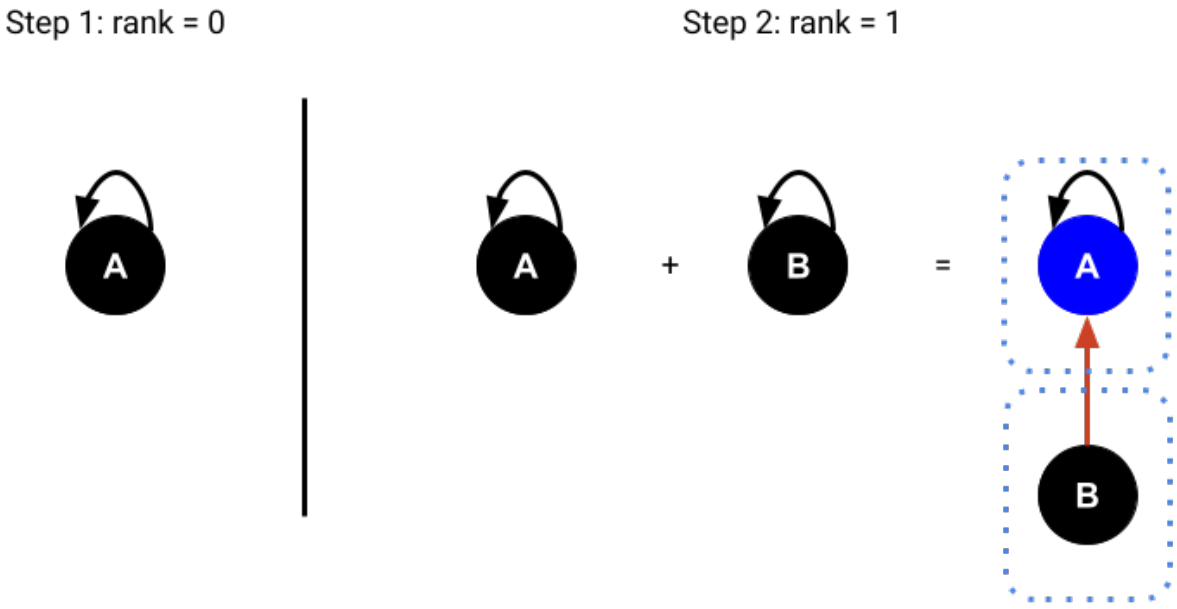
claim: the number of nodes in a tree of rank r is at least 2^r

Intuitively, to get a tree of rank $r+1$ we have to *(at least) double the size* of a tree with rank r

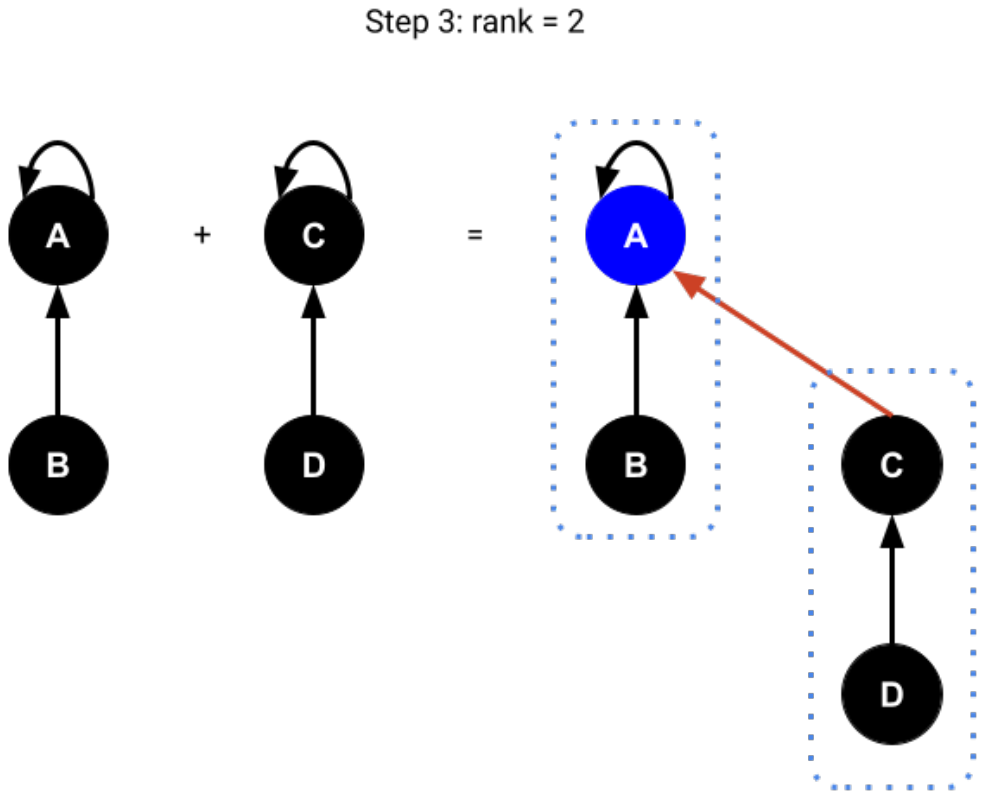
Demo (minimum tree given a rank)

$\rightarrow$ when union by rank is performed, `find` takes $O(log(N))$ time, where N is the total number of nodes in the tree

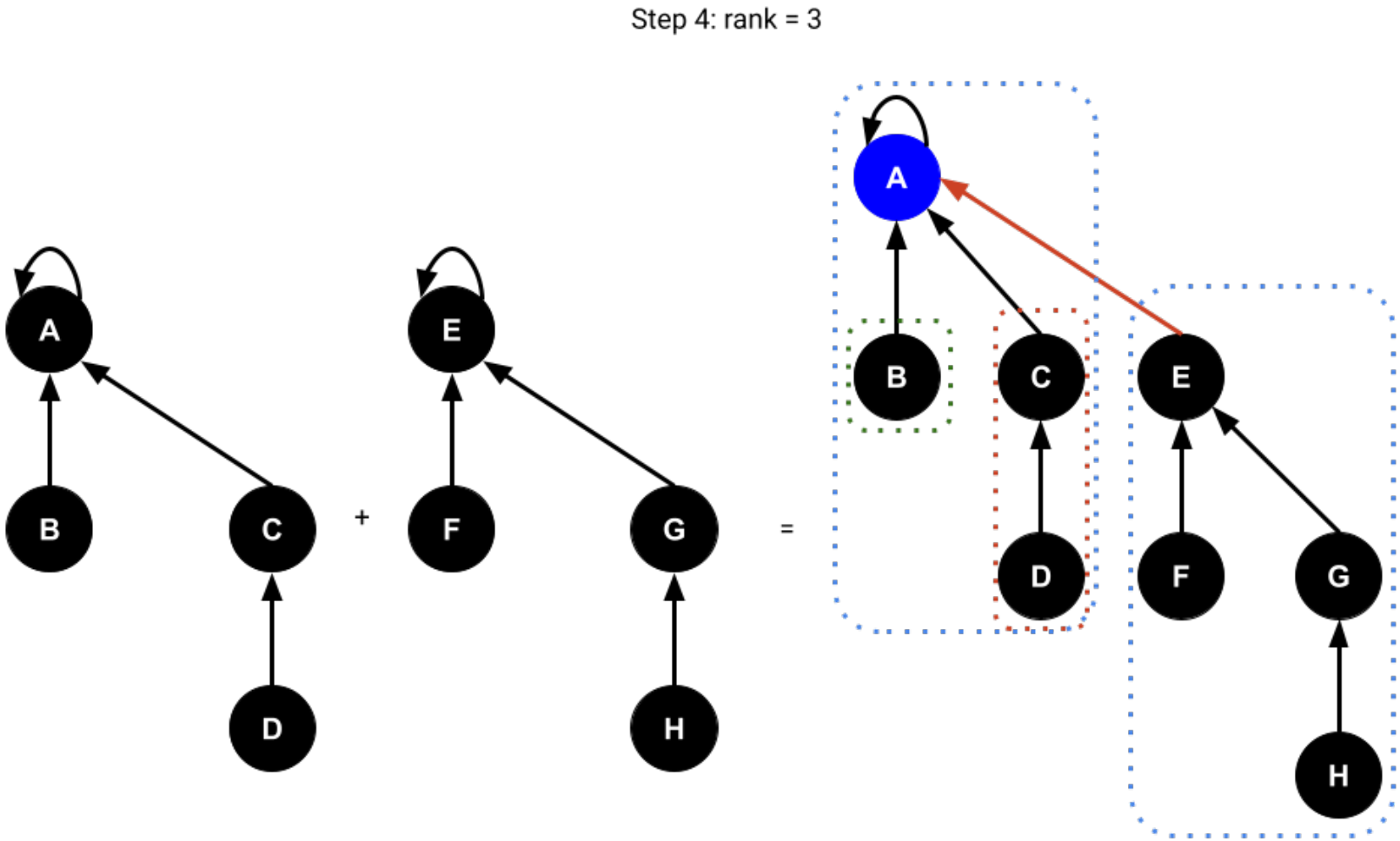
Union by rank - pt.1



Union by rank - pt.2



Union by rank - pt.3



Path compression

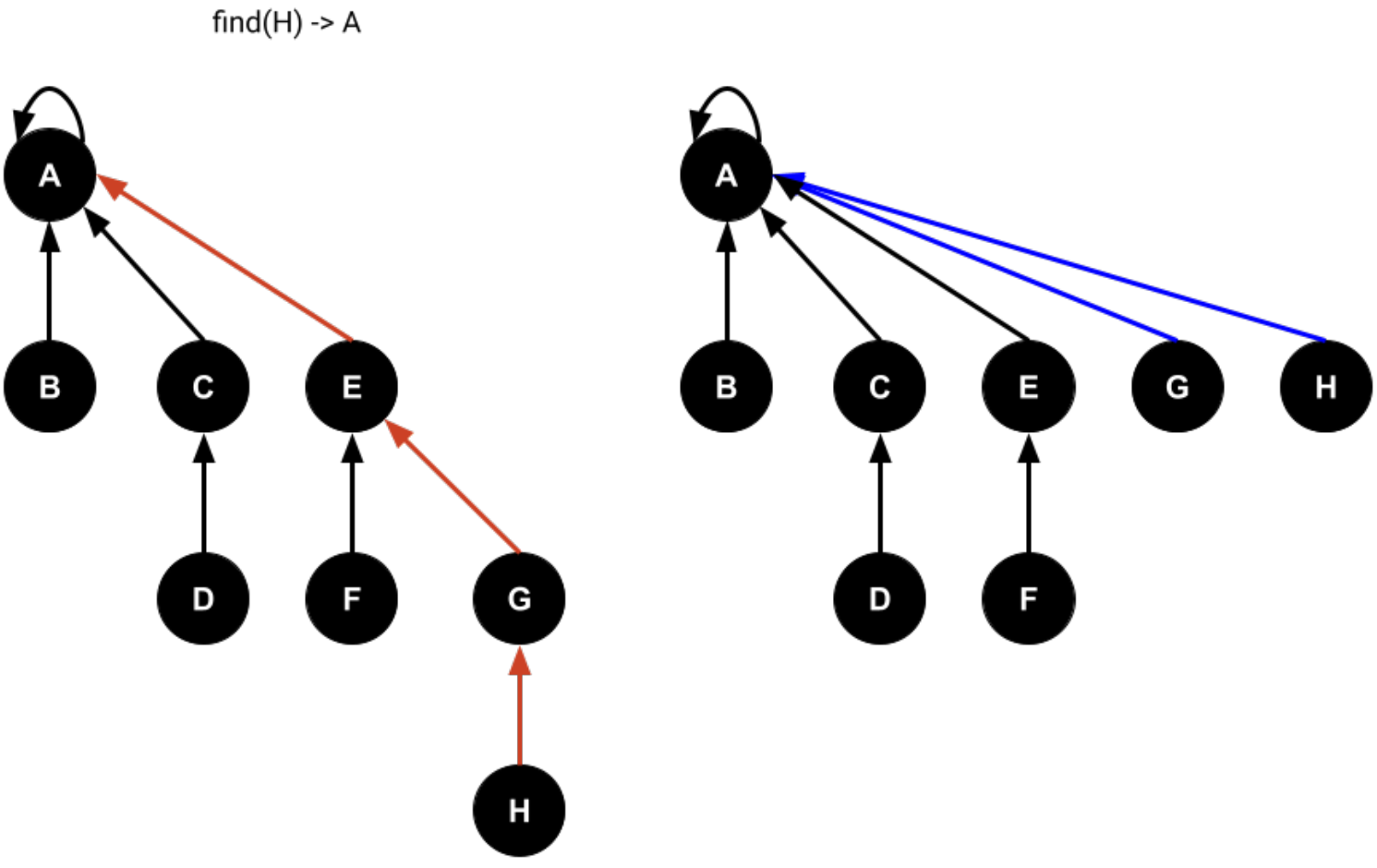
We may want to shorten the path to the representative for some elements, it's a simple (constant-time) operation that can speed up subsequent calls to `find`

Path compression

- every time we find the representative of an element, we can reassign the representative of each traversed element to flatten the tree

Demo

Path compression



Complexity

What happens when we combine union by rank and path compression? The data structure becomes particularly efficient

Amortized analysis is required to demonstrate complexity, however, the runtime of performing M `union` and `find` operations on a disjoint-set forest with N nodes is $O(M\alpha(N))$

α is the inverse Ackermann function, which is **extremely slowly-growing**

Ackermann function - A

There are inequivalent definitions of "the Ackermann function" in textbooks

What you need to remember is that it grows faster than an exponential function, or even a multiple of exponential function

To give an idea, the inverse Ackermann function $\alpha(N)$ never exceeds 4 when $N \leq 10^{80}$

Home exercise

Exercise 1

```
Use disjoint-set to detect cycles in an undirected graph
```

Exercise 2

```
Implement the Kruskal MST algorithm with disjoint-set
```

References

Chapter 21 of textbook Introduction to Algorithms (Cormen, Leiserson, Rivest, Stein)

Robert E. Tarjan and Jan van Leeuwen. 1984.
Worst-case Analysis of Set Union Algorithms.
J. ACM 31, 2 (April 1984), 245 281
https://doi.org/10.1145/62.2160

Dynamic Programming

Dario Facchinetti

Unit 5

A different approach to solve problems:

- Dynamic Programming (DP)

Idea

Sometimes a complex problem can be solved combining the solutions of simpler (or smaller) versions of the original problem

The answer to each sub-problem is usually saved in a table, this avoids the work of recomputing the answer every time a certain sub-problem must be solved (again)

Disclaimer

Dynamic Programming is often used to solve optimization problems

These problems can have *many* solutions

We aim to find *an* optimal solution, not *the* optimal one

Steps

To develop a DP solution:

- identify the structure of an optimal solution
- recursively define the value of the optimal solution
- compute the optimal solution (bottom-up)
- (optionally) reconstruct the optimal solution with the computed information

We will learn the process by example

Edit distance

```
The minimum edit distance between two strings is the minimum
number of editing operations needed to change the first
string into the other. Valid edit operations consist of
insertions, deletions and substitutions.

Example:
  s1 = "rose"
  s2 = "frost"
  result = 2
```

Minimum number of coins

```
You are given an array of coins having different
denominations and an integer amount representing the total
money. Return the fewest number of coins that are needed
to make up that sum, or +Inf if it is impossible. Assume
that there is an infinite number of coins for each
denomination.

Example:
  amount = 7
  coins = [1,2,5,10,20]
  result = 2 coins, [2, 5]
```

Knapsack

```
You are given a list of items. Each item has a weight and a
value. You are also given a knapsack with a maximum
capacity, or rather  the maximum weight it can carry.
Determine which of the items to pack in the knapsack so that
the value is maximum.

Example:
  values = [6, 10, 12]
  weights = [10, 20, 30]
  knapsack capacity = 50
  result = 22
```