Scientific Programming: Algorithms and Data Structures

Introduction

Luca Bianco - Academic Year 2020-21
luca.bianco@fmach.it
[credits: thanks to Prof. Alberto Montresor]

About me

Computer Science

Ph.D. at the University of Verona, Italy, with thesis on Simulation of Biological Systems

Research Fellow at Cranfield University - UK

Three years at Cranfield University working at proteomics projects (GAPP, MRMaid, X-Tracker...)

Module manager and lecturer in several courses of the MSc in Bioinformatics

Bioinformatician at IASMA – FEM

Currently bioinformatician in the Computational Biology Group at Istituto Agrario di San Michele all'Adige – Fondazione Edmund Mach, Trento, Italy

Collaborator uniTN - CiBio

I ran the Scientific Programming Lab for QCB for the last couple of years and this course last year

Organization

145540 Scientific Programming (12 ECTS, LM QCB)
145685 Scientific Programming (12 ECTS, LM Data Science)

Part A - Programming (22/9-29/10)

Introduction to the Python language and to a collection of programming libraries for data analysis.

- Mutuated as 145912 Scientific Programming (LM Math, 6 credits)

Part B - Algorithms (3/11-15/12)

Design and analysis of algorithmic solutions. Presentation of the most important classes of algorithms and evaluation of their performance.

Topics

- Introduction
  - Recursion
  - Algorithm analysis
  - Asymptotic notations
- Data structures
  - High level overview
  - Sequences, maps (ordered/unordered), sets
  - Data structure implementations in Python
- Trees
  - Data structure definition
  - Visits
- Graphs
  - Data structure definition
  - Visits
  - Algorithms on graphs
- Algorithmic techniques
  - Divide-et-impera
  - Dynamic programming
  - Greedy
  - Backtrack
  - NP class: brief overview

Learning outcomes

At the end of the module, students are expected to:

- evaluate algorithmic choices and select the ones that best suit their problems;
- analyze the complexity of existing algorithms and algorithms created on their own;
- design simple algorithmic solutions to solve basic problems.

Teaching team

- Instructor: Dr. Luca Bianco
  - Theory lectures, algorithmic exercises
  - luca.bianco [AT] fmach.it
- Teaching assistant: Dr. Erik Dassi
  - Lab sessions on algorithms (QCB)
  - erik.dassi [AT] unitn.it
- Teaching assistant: Dr. David Leoni
  - Lab sessions on algorithms (data science)
  - david.leoni [AT] unitn.it

Tutors:

Gabriele Masina (QCB)
Andrea Ferigo (Data Science)

Schedule

Week day	Time	Room	Description
Monday	14.30-16.30	online	Lab
Tuesday	15.30-17.30	online	Lecture
Wednesday	11.30-13.30	online	Lab
Thursday	15.30-17.30	online	Lecture

midterms:

Part A on Friday, November 6th 11:30-13:30 online

— for QCB no lab tomorrow... Study time!

Part B (tentatively ~ December, 16th... more on this closer to the date)

Mark registration

145540,145685 Scientific Programming (12 credits)

- If you pass **both** midterm exams, you can register the mark
- The mark is computed as the average of the marks of the midterm exams, rounded up (e.g. $(25+26)/2 = 26$)
- To register your mark you need to enroll to one of the regular sessions (not the midterm ones).
- If you passed both midterm exams, enroll to a session and do not show up, we assume you want to register your mark

Mark registration

continued

- If you passed both midterm exams, enroll to a session and do show up, this means that you are not happy with the mark and want to take the full exam. The result of the full exam will be your new mark, you cannot backtrack to the midterm mark.
- If you did not pass both midterm exams, you need to take the full exam at a regular session.
- After the mark of a regular session have been published, you have a week to refuse it, after which it will be registered (silent assent registration).

Full exams

Full exams

January (3h)	TBD
February (3h)	TBD
June (3h)	TBD
July (3h)	TBD
September (3h)	TBD

A typical full exam is composed by (beware weights might change):

1. Theoretical part: 2 questions on theoretical aspects of this second part (~ 6-7 points);
2. Exercise(s) covering part A (~ 12 points);
3. Exercise(s) covering part B (~12 points)

Course material

Lectures:

Material and information: https://sciproalgo2020.readthedocs.io/en/latest/

Lecture recordings on Moodle:
https://didatticaonline.unitn.it/dol/enrol/index.php?id=25445

Practicals:

QCB: https://bitbucket.org/erikdassi/sciprog2020

Data science: https://datasciprolab.readthedocs.io/en/latest/

[Thanks to Prof. Alberto Montresor for the material]

Course material

Scientific Programming: Algorithms

General Info

The contacts to reach me can be found at this page.

Timetable and lecture rooms

Lectures will take place on Tuesdays from 15:30 to 17:30 (synchronous online if not otherwise communicated) and on Thursdays from 15:30 to 17:30 (synchronous online if not otherwise communicated). This second part of the Scientific Programming course will tentatively run from 03/11/2020 to 14/12/2020.

Midterm

The midterm of this part of the course will take place on Wednesday, December 16th, online at 11:30-13.30.

Moodle

In the moodle page of the course you can find announcements and videos of the lectures. It can be found here.

Zoom links

The zoom links for the lectures can be found in the Announcements section of the moodle web page.

Slides

The slides shown during the lectures will gradually appear below:

Teaching assistants

David Leoni (for Data Science)

Erik Dassi (for QCB)

Lectures Part B (Algorithms and Data Structures)

Details for the Zoom connection:

Time: Tuesdays and Thursdays 15.30 - 17.30

Starting date: Tuesday, November 3rd

Topic: Algorithms and Data Structures

Join Zoom Meeting

https://unitn.zoom.us/j[redacted]

Meeting ID: [redacted]

Passcode: [redacted]

Course Material

same moodle page as part A

slides will appear down here

https://sciproalgo2020.readthedocs.io/en/latest/

Before starting...

Please do not be shy...

Questions **help you/your colleagues** to understand better **and help me** to be clearer in my presentation

Let's try to make this interactive, please use the chat!

For when you have 3 spare minutes…
https://theleadertheteacher.wordpress.com/2016/07/18/be-like-a-child-again-ask-more-questions/

Where we stand...

So far...

we have learnt a bit of Python and we started doing some little examples of data analysis (saw some libraries, etc...)

From now on..

we will focus on:

“Solving problems” providing solutions (**focusing on correctness**), possibly in an efficient way (**assessing their complexity**), organizing data in the most suitable/efficient ways (**choosing the right data structures**)

Maximal sum problem

- **Input**: a list A containing n numbers
- **Output**: a slice (sublist) $A[i:j]$ of maximal sum, i.e. the slice whose element sum $\sum_{k=i}^{j-1} A[k]$ is larger or equal than the sum of any other slice

↓ simpler problem

Find the maximal sum, rather than the interval that provides the maximal sum.

Is the problem clear?

Example:

1	3	4	-8	2	3	-1	3	4	-3	10	-3	2

Maximal sum problem

- **Input**: a list A containing n numbers
- **Output**: a slice (sublist) $A[i:j]$ of maximal sum, i.e. the slice whose element sum $\sum_{k=i}^{j-1} A[k]$ is larger or equal than the sum of any other slice

simpler problem

Find the maximal sum, rather than the interval that provides the maximal sum.

Is the problem clear?

Example:

0				4						10		12
1	3	4	-8	2	3	-1	3	4	-3	10	-3	2

Maximal sum problem

- **Input**: a list A containing n numbers
- **Output**: a slice (sublist) $A[i : j]$ of maximal sum, i.e. the slice whose element sum $\sum_{k=i}^{j-1} A[k]$ is larger or equal than the sum of any other slice

simpler problem

Find the maximal sum, rather than the interval that provides the maximal sum.

Is the problem clear?

Example:

0				4						10		12
1	3	4	-8	2	3	-1	3	4	-3	10	-3	2

Maximal sum: 18. **Any ideas on how to solve this problem?**

Solution 1 ~ N^3

- **Input**: a list A containing n numbers
- **Output**: a slice (sublist) $A[i:j]$ of maximal sum, i.e. the slice whose element sum $\sum_{k=i}^{j-1} A[k]$ is larger or equal than the sum of any other slice

1	3	4	-8	2	3	-1	3	4	-3	10	-3	2

Idea:

Given the list A with N elements

Consider **all pairs** (i,j) such that i ≤ j
Get the elements in A[i:j+1]
Compute the **sum** of all elements in A[i:j+1]
Update max_so_far **if sum ≥ max_so_far**

```
def max_sum_v1(A):
    max_so_far = 0
    N = len(A)
    for i in range(N):
        for j in range(i,N):
            tmp_sum = sum (A[i:j+1])
            max_so_far = max(tmp_sum, max_so_far)

    return max_so_far
```

```
A = [1,3,4,-8,2, 3,-1,3,4,-3,10,-3,2]
print(A)
print(max_sum_v1(A))
```

```
[1, 3, 4, -8, 2, 3, -1, 3, 4, -3, 10, -3, 2]
18
```

List comprehension... ?

- **Input**: a list A containing n numbers
- **Output**: a slice (sublist) $A[i:j]$ of maximal sum, i.e. the slice whose element sum $\sum_{k=i}^{j-1} A[k]$ is larger or equal than the sum of any other slice

1	3	4	-8	2	3	-1	3	4	-3	10	-3	2

```
def max_sum_v1_listc_1(A):
    N = len(A)
    sums = [sum(A[i:j+1]) for i in range(N) for j in range(i,N)]

    return max(sums)

A = [1,3,4,-8,2, 3,-1,3,4,-3,10,-3,2]
print(A)
print(max_sum_v1_listc_1(A))
```

```
[1, 3, 4, -8, 2, 3, -1, 3, 4, -3, 10, -3, 2]
18
```

List comprehension... ?

- **Input**: a list A containing n numbers
- **Output**: a slice (sublist) $A[i:j]$ of maximal sum, i.e. the slice whose element sum $\sum_{k=i}^{j-1} A[k]$ is larger or equal than the sum of any other slice

1	3	4	-8	2	3	-1	3	4	-3	10	-3	2

```
def max_sum_v1_listc_1(A):
    N = len(A)
    sums = [sum(A[i:j+1]) for i in range(N) for j in range(i,N)]

    return max(sums)

A = [1,3,4,-8,2, 3,-1,3,4,-3,10,-3,2]
print(A)
print(max_sum_v1_listc_1(A))
```

```
[1, 3, 4, -8, 2, 3, -1, 3, 4, -3, 10, -3, 2]
18
```

How many elements?

List comprehension... ?

- **Input**: a list A containing n numbers
- **Output**: a slice (sublist) $A[i:j]$ of maximal sum, i.e. the slice whose element sum $\sum_{k=i}^{j-1} A[k]$ is larger or equal than the sum of any other slice

1	3	4	-8	2	3	-1	3	4	-3	10	-3	2

No thanks!

```
def max_sum_v1_listc_1(A):
    N = len(A)
    sums = [sum(A[i:j+1]) for i in range(N) for j in range(i,N)]

    return max(sums)

A = [1,3,4,-8,2, 3,-1,3,4,-3,10,-3,2]
print(A)
print(max_sum_v1_listc_1(A))
```

```
[1, 3, 4, -8, 2, 3, -1, 3, 4, -3, 10, -3, 2]
18
```

How many elements?

N*(N+1)/2 ~ N^2

[1, 4, 8, 0, 2, 5, 4, 7, 11, 8, 18, 15, 17, 3, 7, -1, 1, 4, 3, 6, 10, 7, 17, 14, 16, 4, -4, -2, 1, 0, 3, 7, 4, 14, 11, 13, -8, -6, -3, -4, -1, 3, 0, 10, 7, 9, 2, 5, 4, 7, 11, 8, 18, 15, 17, 3, 2, 5, 9, 6, 16, 13, 15, -1, 2, 6, 3, 13, 10, 12, 3, 7, 4, 14, 11, 13, 4, 1, 11, 8, 10, -3, 7, 4, 6, 10, 7, 9, -3, -1, 2]

→ 91 elements! (= 13*14/7)

If A has 100,000 elements → ~ 40 GB RAM!!!

List comprehension... ?

- **Input**: a list A containing n numbers
- **Output**: a slice (sublist) $A[i:j]$ of maximal sum, i.e. the slice whose element sum $\sum_{k=i}^{j-1} A[k]$ is larger or equal than the sum of any other slice

No thanks!

```
def max_sum_v1_listc(A):
    N = len(A)
    intervals = [A[i:j+1] for i in range(N) for j in range(i,N)]
    sums = [sum(vals) for vals in intervals]
    return max(sums)

A = [1,3,4,-8,2, 3,-1,3,4,-3,10,-3,2]
print(A)
print(max_sum_v1_listc(A))
```

```
[1, 3, 4, -8, 2, 3, -1, 3, 4, -3, 10, -3, 2]
18
```

Stores intervals and sums!!!

If A has 100,000 elements ➔ ~ 1.3 PB RAM!!!

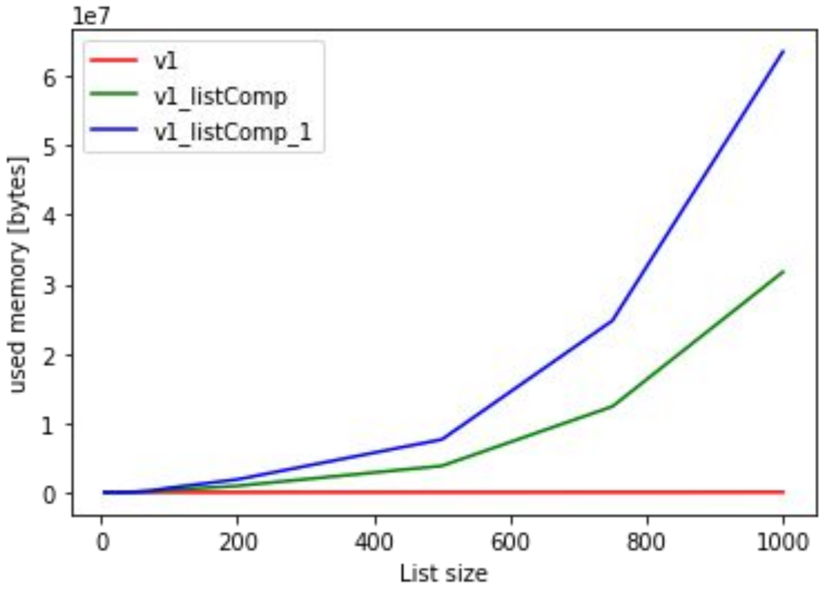
List comprehension… ?

- **Input**: a list A containing n numbers
- **Output**: a slice (sublist) $A[i:j]$ of maximal sum, i.e. the slice whose element sum $\sum_{k=i}^{j-1} A[k]$ is larger or equal than the sum of any other slice

v1: two variables

v1_listComp: 1 list with ~ N^2 numbers

v1_listComp_1: 2 lists
- 1 with ~ N^2 numbers
- 1 with ~ N^2 sublists of numbers



[size computed with **sys.getsizeof(DATA)**]

Important note:

Time and space (memory) are two important resources!

https://docs.python.org/3/library/sys.html?highlight=sizeof#sys.getsizeof

Solution 1 ~ N^3

Idea:

Given the list A with N elements

Consider all pairs (i,j) such that i ≤ j
Get the elements in A[i:j+1]
Compute the sum of all elements in A[i:j+1]
Update max_so_far if sum ≥ max_so_far

```
def max_sum_v1(A):
    max_so_far = 0
    N = len(A)
    for i in range(N):
        for j in range(i,N):
            tmp_sum = sum (A[i:j+1])
            max_so_far = max(tmp_sum, max_so_far)

    return max_so_far
```

- **Input**: a list A containing n numbers
- **Output**: a slice (sublist) $A[i:j]$ of maximal sum, i.e. the slice whose element sum $\sum_{k=i}^{j-1} A[k]$ is larger or equal than the sum of any other slice

1	3	4	-8	2	3	-1	3	4	-3	10	-3	2

Why N^3 ?

Intuitively,

We have N*(N+1)/2 intervals and the sum of N numbers takes N operations.

So: N * [N*(N+1)/2] ~ N^3

Can we do any better than this?

Solution 2 ~ N^2

- **Input**: a list A containing n numbers
- **Output**: a slice (sublist) $A[i:j]$ of maximal sum, i.e. the slice whose element sum $\sum_{k=i}^{j-1} A[k]$ is larger or equal than the sum of any other slice

1	3	4	-8	2	3	-1	3	4	-3	10	-3	2

Observation: There is no point in computing the same sums over and over again!

If S = sum(A[i:j]) ➔ sum(A[i:j+1]) = S + A[j+1]

```
def max_sum_v2(A):
    N = len(A)
    max_so_far = 0
    for i in range(N):
        tot = 0 #ACCUMULATOR!
        for j in range(i,N):
            tot = tot + A[j]
            max_so_far = max(max_so_far, tot)
    return max_so_far

A = [1,3,4,-8,2, 3,-1,3,4,-3,10,-3,2]
print(A)
print(max_sum_v2(A))
```

```
[1, 3, 4, -8, 2, 3, -1, 3, 4, -3, 10, -3, 2]
18
```

Solution 2 ~ N^2

Observation: There is no point in computing the same sums over and over again!

If S = sum(A[i:j]) ➔ sum(A[i:j+1]) = S + A[j+1]

```
def max_sum_v2(A):
    N = len(A)
    max_so_far = 0
    for i in range(N):
        tot = 0 #ACCUMULATOR!
        for j in range(i,N):
            tot = tot + A[j]
            max_so_far = max(max_so_far, tot)
    return max_so_far

A = [1,3,4,-8,2, 3,-1,3,4,-3,10,-3,2]
print(A)
print(max_sum_v2(A))
```

```
[1, 3, 4, -8, 2, 3, -1, 3, 4, -3, 10, -3, 2]
18
```

- **Input**: a list A containing n numbers
- **Output**: a slice (sublist) $A[i:j]$ of maximal sum, i.e. the slice whose element sum $\sum_{k=i}^{j-1} A[k]$ is larger or equal than the sum of any other slice

1	3	4	-8	2	3	-1	3	4	-3	10	-3	2

Tot (i, j)

0, 1, 4, 8, 0, 2, 5, 4, 7, 11, 8, 18, 15, 17, ← (0, x)
0, 3, 7, -1, 1, 4, 3, 6, 10, 7, 17, 14, 16, ← (1, x)
0, 4, -4, -2, 1, 0, 3, 7, 4, 14, 11, 13, ← (2, x)
0, -8, -6, -3, -4, -1, 3, 0, 10, 7, 9,
0, 2, 5, 4, 7, 11, 8, 18, 15, 17,
0, 3, 2, 5, 9, 6, 16, 13, 15,
0, -1, 2, 6, 3, 13, 10, 12,
0, 3, 7, 4, 14, 11, 13,
0, 4, 1, 11, 8, 10,
0, -3, 7, 4, 6,
0, 10, 7, 9,
0, -3, -1,
0, 2 ← (N-1, x)

Maxes (max_so_far)
[0, 1, 4, 8, 8, 8, 8, 8, 8, 11, 11, 18, 18, 18, .., 18]

Solution 2 ~ N^2

Observation: There is no point in computing the same sums over and over again!

If S = sum(A[i:j]) ➜ sum(A[i:j+1]) = S + A[j+1]

```
def max_sum_v2(A):
    N = len(A)
    max_so_far = 0
    for i in range(N):
        tot = 0 #ACCUMULATOR!
        for j in range(i,N):
            tot = tot + A[j]
            max_so_far = max(max_so_far, tot)
    return max_so_far

A = [1,3,4,-8,2, 3,-1,3,4,-3,10,-3,2]
print(A)
print(max_sum_v2(A))
```

```
[1, 3, 4, -8, 2, 3, -1, 3, 4, -3, 10, -3, 2]
18
```

- **Input**: a list A containing n numbers
- **Output**: a slice (sublist) $A[i:j]$ of maximal sum, i.e. the slice whose element sum $\sum_{k=i}^{j-1} A[k]$ is larger or equal than the sum of any other slice

Why N^2 ?

Intuitively, we have to consider **N*(N+1)/2 ~ N^2** intervals (for each interval we compute **ONE sum** and **the maximum of TWO values**: constant time!)

The space required is just a couple of variables: **constant**!

Solution 2 ~ N^2

Tip: use itertools (similar to np.cumsum())

Accumulate of itertools is done in C so it is faster

Table of Contents

Previous topic

Functional Programming Modules

Next topic

functools — Higher-order functions and operations on callable objects

This Page

Report a Bug
Show Source

`itertools` — Functions creating iterators for efficient looping

This module implements a number of iterator building blocks inspired by constructs from APL, Haskell, and SML. Each has been recast in a form suitable for Python.

The module standardizes a core set of fast, memory efficient tools that are useful by themselves or in combination. Together, they form an “iterator algebra” making it possible to construct specialized tools succinctly and efficiently in pure Python.

For instance, SML provides a tabulation tool: `tabulate(f)` which produces a sequence `f(0), f(1), ...`. The same effect can be achieved in Python by combining `map()` and `count()` to form `map(f, count())`.

These tools and their built-in counterparts also work well with the high-speed functions in the `operator` module. For example, the multiplication operator can be mapped across two vectors to form an efficient dot-product: `sum(map(operator.mul, vector1, vector2))`.

Infinite iterators:

Iterator	Arguments	Results	Example
`count()`	start, [step]	start, start+step, start+2*step, …	`count(10) --> 10 11 12 13 14 ...`
`cycle()`	p	p0, p1, … plast, p0, p1, …	`cycle('ABCD') --> A B C D A B C D ...`
`repeat()`	elem [,n]	elem, elem, elem, … endlessly or up to n times	`repeat(10, 3) --> 10 10 10`

Iterators terminating on the shortest input sequence:

Iterator	Arguments	Results	Example
`accumulate()`	p [,func]	p0, p0+p1, p0+p1+p2, …	`accumulate([1,2,3,4,5]) --> 1 3 6 10 15`

https://docs.python.org/3/library/itertools.html

- **Input**: a list A containing n numbers
- **Output**: a slice (sublist) $A[i:j]$ of maximal sum, i.e. the slice whose element sum $\sum_{k=i}^{j-1} A[k]$ is larger or equal than the sum of any other slice

```python
from itertools import accumulate

A = [1,2,3,-3,12,-4,1,-1,-2,1]
print(A)
print(list(accumulate(A)))
```

```
[1, 2, 3, -3, 12, -4, 1, -1, -2, 1]
[1, 3, 6, 3, 15, 11, 12, 11, 9, 10]
```

Solution 2 ~ N^2

Tip: use itertools (similar to np.cumsum())

Accumulate of itertools is done in C so it is faster

```
from itertools import accumulate

def max_sum_v2_bis(A):
    N = len(A)
    max_so_far = 0
    for i in range(N):
        tot = max(accumulate(A[i:]))
        max_so_far = max(max_so_far,tot)
    return max_so_far


A = [1,3,4,-8,2, 3,-1,3,4,-3,10,-3,2]
print(A)
print(max_sum_v2_bis(A))
```

```
[1, 3, 4, -8, 2, 3, -1, 3, 4, -3, 10, -3, 2]
18
```

- **Input**: a list A containing n numbers
- **Output**: a slice (sublist) $A[i:j]$ of maximal sum, i.e. the slice whose element sum $\sum_{k=i}^{j-1} A[k]$ is larger or equal than the sum of any other slice

```
from itertools import accumulate

A = [1,2,3,-3,12,-4,1,-1,-2,1]
print(A)
print(list(accumulate(A)))
```

```
[1, 2, 3, -3, 12, -4, 1, -1, -2, 1]
[1, 3, 6, 3, 15, 11, 12, 11, 9, 10]
```

Similar as before but max computed on the accumulated sum (accumulate “hides” a for loop)

N intervals, sum of N elements each time: ~ N^2 operations

IMPORTANT NOTE:
The improvement comes from implementation not algorithm! (code faster by a constant factor)

Can we do any better than this?

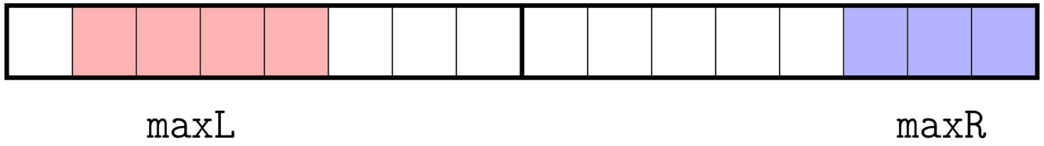
Solution 3 ~ N log(N)

- **Input**: a list A containing n numbers
- **Output**: a slice (sublist) $A[i:j]$ of maximal sum, i.e. the slice whose element sum $\sum_{k=i}^{j-1} A[k]$ is larger or equal than the sum of any other slice

Divide et impera (Divide and conquer)

Idea:

- Split it in two equally sized sublists
- Find maxL as the sum of the maximal sublist on the left part
- Find maxR as the sum of the maximal sublist on the right part
- Get the solution as max(maxL, maxR)



Is this correct? Do you see any problem with this?

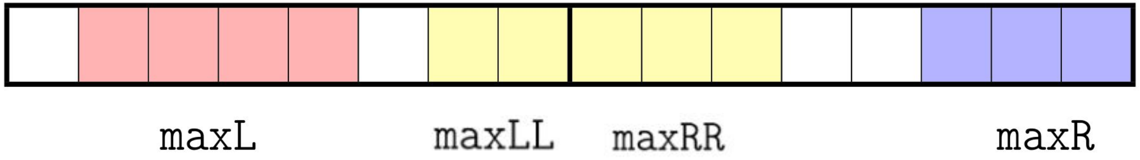
Solution 3 ~ N log(N)

- **Input**: a list A containing n numbers
- **Output**: a slice (sublist) $A[i:j]$ of maximal sum, i.e. the slice whose element sum $\sum_{k=i}^{j-1} A[k]$ is larger or equal than the sum of any other slice

Divide et impera (Divide and conquer)

Idea:

- Split it in two equally sized sublists
- Find **maxL** as the sum of the maximal sublist on the left part
- Find **maxR** as the sum of the maximal sublist on the right part
- **maxLL+maxRR** is the value of the maximal sublist accross the two parts



Solution 3 ~ N log(N)

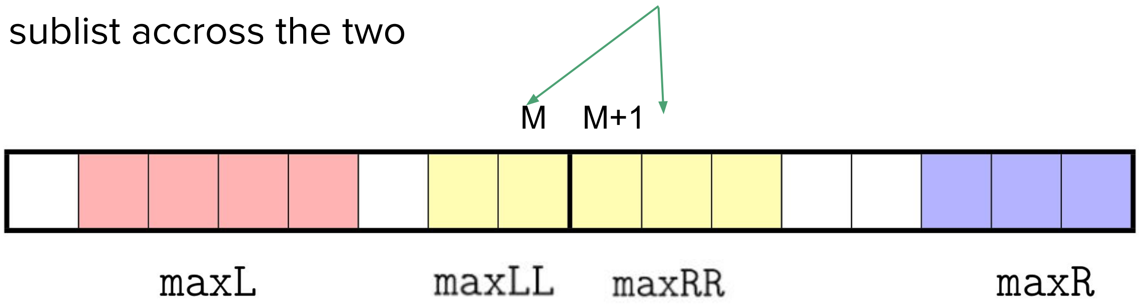
- **Input**: a list A containing n numbers
- **Output**: a slice (sublist) $A[i:j]$ of maximal sum, i.e. the slice whose element sum $\sum_{k=i}^{j-1} A[k]$ is larger or equal than the sum of any other slice

Divide et impera (Divide and conquer)

Idea:

- Split it in two equally sized sublists
- Find **maxL** as the sum of the maximal sublist on the left part
- Find **maxR** as the sum of the maximal sublist on the right part
- **maxLL+maxRR** is the value of the maximal sublist accross the two parts

Get the point before the mid-point M and go to the left until the sum increases.
Repeat starting from M+1 and going to the right.
Result is: max(maxL, maxLL+maxRR, maxR)



Solution 3 ~ N log(N)

Divide et impera (Divide and conquer)

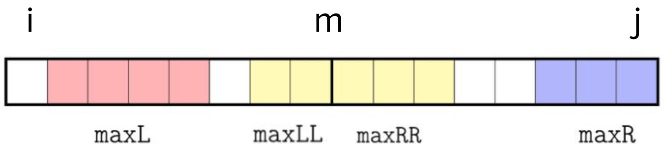
```
def max_sum_v3_rec(A, i, j):
    if i == j:
        return max(0, A[i])
    m = (i+j)//2
    maxML = 0
    s = 0
    for k in range(m,i-1,-1):
        s = s + A[k]
        maxML = max(maxML, s)

    maxMR = 0
    s = 0
    for k in range(m+1, j+1):
        s = s + A[k]
        maxMR = max(maxMR, s)
    maxL = max_sum_v3_rec(A,i,m) #Left maximal subvector
    maxR = max_sum_v3_rec(A,m+1,j) #Right maximal subvector

    return max(maxL, maxR, maxML + maxMR)

def max_sum_v3(A):
    return max_sum_v3_rec(A,0,len(A) - 1)


A = [1,3,4,-8,2, 3,-1,3,4,-3,10,-3,2]
print(A)
print(max_sum_v3(A))
```

```
[1, 3, 4, -8, 2, 3, -1, 3, 4, -3, 10, -3, 2]
18
```

- **Input**: a list A containing n numbers
- **Output**: a slice (sublist) $A[i:j]$ of maximal sum, i.e. the slice whose element sum $\sum_{k=i}^{j-1} A[k]$ is larger or equal than the sum of any other slice

Recursive code: calls itself on a smaller sublist.

Runs in **N*log(N)** … more on this later



Solution 3 ~ N log(N)

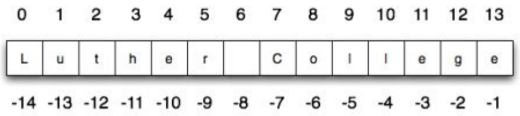
- **Input**: a list A containing n numbers
- **Output**: a slice (sublist) $A[i:j]$ of maximal sum, i.e. the slice whose element sum $\sum_{k=i}^{j-1} A[k]$ is larger or equal than the sum of any other slice

Divide et impera (Divide and conquer)

Tip: use itertools

```
def max_sum_v3_rec_bis(A,i,j):
    if i == j:
        return max(0,A[i])
    m = (i+j)//2
    maxL  = max_sum_v3_rec_bis(A,i,m)
    maxR  = max_sum_v3_rec_bis(A, m+1, j)
    maxML  = max(accumulate(A[m:-len(A) + i -1: -1]))
    maxMR  = max(accumulate(A[m+1:j+1]))
    return max(maxL, maxR, maxML+ maxMR)

def max_sum_v3(A):
    return max_sum_v3_rec_bis(A,0,len(A) - 1)


A = [1,3,4,-8,2, 3,-1,3,4,-3,10,-3,2]
print(A)
print(max_sum_v3(A))
```

```
[1, 3, 4, -8, 2, 3, -1, 3, 4, -3, 10, -3, 2]
18
```

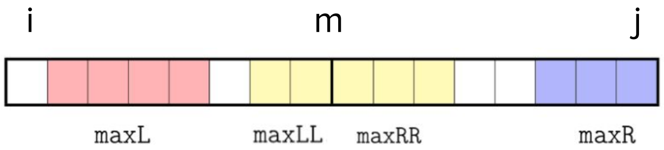



```
A = list(range(10))
print(A)

#interval 4-2 going to the left...
M = 4
i = 2
print(-len(A) + i - 1)
A[M: -len(A) + i -1 : -1]
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
-9

[4, 3, 2]
```

Recursive code: can use itertools as before to accumulate the sum.

Runs in **N*log(N)** …just a little bit faster, more on this later



Solution 4 ~ N

- **Input**: a list A containing n numbers
- **Output**: a slice (sublist) $A[i:j]$ of maximal sum, i.e. the slice whose element sum $\sum_{k=i}^{j-1} A[k]$ is larger or equal than the sum of any other slice

Dynamic Programming

Let's define **maxHere[i]** as the maximum value of each sublist that ends in i.

The **maximum value in maxHere** is the **maximal sum**.

$$maxHere[i] = \begin{cases} 0 & i < 0 \\ \max(maxHere[i-1] + A[i], 0) & i \geq 0 \end{cases}$$

condition i<0
needs to fix the first element of the list (i.e. when i=0)

Solution 4 ~ N

- **Input**: a list A containing n numbers
- **Output**: a slice (sublist) $A[i:j]$ of maximal sum, i.e. the slice whose element sum $\sum_{k=i}^{j-1} A[k]$ is larger or equal than the sum of any other slice

Dynamic Programming

Let's define maxHere[i] as the maximum value of each sublist that ends in i.

The maximum value in maxHere is the maximal sum.

$$maxHere[i] = \begin{cases} 0 & i < 0 \\ \max(maxHere[i-1] + A[i], 0) & i \geq 0 \end{cases}$$

```
def max_sum_v4(A):
    max_so_far = 0 #Max found so far
    max_here = 0   #Max slice ending at cur pos
    for i in range(len(A)):
        max_here = max(A[i] + max_here, 0)
        max_so_far = max(max_so_far, max_here)
    return max_so_far

A = [1,3,4,-8,2, 3,-1,3,4,-3,10,-3,2]
print("{}".format(A))
print(max_sum_v4(A))
```

```
[1, 3, 4, -8, 2, 3, -1, 3, 4, -3, 10, -3, 2]
18
```

Solution 4 ~ N

Dynamic Programming

- **Input**: a list A containing n numbers
- **Output**: a slice (sublist) $A[i:j]$ of maximal sum, i.e. the slice whose element sum $\sum_{k=i}^{j-1} A[k]$ is larger or equal than the sum of any other slice

```
def max_sum_v4(A):
    max_so_far = 0 #Max found so far
    max_here = 0   #Max slice ending at cur pos
    for i in range(len(A)):
        max_here = max(A[i] + max_here, 0)
        max_so_far = max(max_so_far, max_here)
    return max_so_far

A = [1,3,4,-8,2, 3,-1,3,4,-3,10,-3,2]
print("{}".format(A))
print(max_sum_v4(A))
```

```
[1, 3, 4, -8, 2, 3, -1, 3, 4, -3, 10, -3, 2]
18
```

A:		1	3	4	-8	2	3	-1	3	4	-3	10	-3	2
max_here:	0	1	4	8	0	2	5	4	7	11	8	18	15	17
max_so_far:	0	1	4	8	8	8	8	8	8	11	11	18	18	18

Goes through A once and computes ONE sum and two max of TWO values:
runs in ~**N**

Solution 4 ~ N

- **Input**: a list A containing n numbers
- **Output**: a slice (sublist) $A[i:j]$ of maximal sum, i.e. the slice whose element sum $\sum_{k=i}^{j-1} A[k]$ is larger or equal than the sum of any other slice

Dynamic Programming

Stores also the indexes

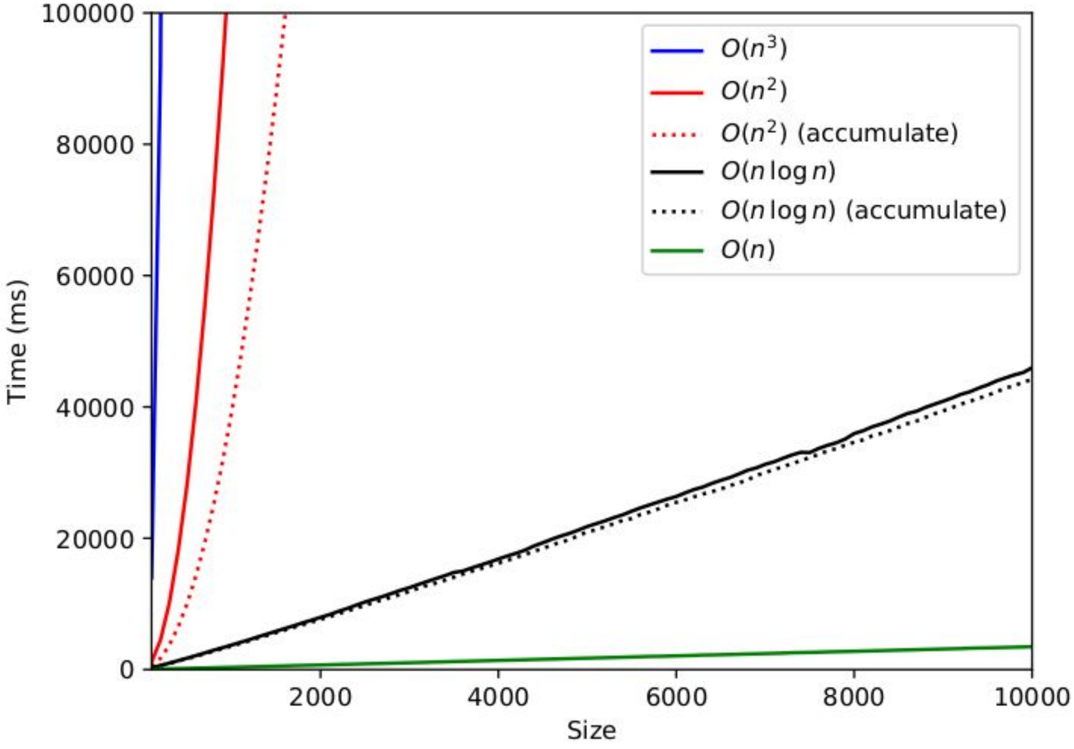
```
def max_sum_v4_bis(A):
    max_so_far = 0 #Max found so far
    max_here = 0   #Max slice ending at cur pos
    start = 0 #start of cur maximal slice
    end = 0   #end of cur maximal slice
    last = 0  #beginning of max slice ending here
    for i in range(len(A)):
        max_here = A[i] + max_here
        if max_here <= 0:
            max_here = 0
            last = i + 1
        if max_here > max_so_far:
            max_so_far = max_here
            start = last
            end = i

    return (start,end,max_so_far)

A = [1,3,4,-8,2, 3,-1,3,4,-3,10,-3,2]
print("A: {}".format(A))
print(max_sum_v4_bis(A))
```

```
A: [1, 3, 4, -8, 2, 3, -1, 3, 4, -3, 10, -3, 2]
(4, 10, 18)
```

A:		1	3	4	-8	2	3	-1	3	4	-3	10	-3	2
max_here:	0	1	4	8	0	2	5	4	7	11	8	18	15	17
max_so_far:	0	1	4	8	8	8	8	8	8	11	11	18	18	18
start:	0	0	0	0	0	0	0	0	0	4	4	4	4	4
last:	0	0	0	0	4	4	4	4	4	4	4	4	4	4
end:	0	0	1	2	2	2	2	2	2	8	8	10	10	10

Running times...

- **Input**: a list A containing n numbers
- **Output**: a slice (sublist) $A[i:j]$ of maximal sum, i.e. the slice whose element sum $\sum_{k=i}^{j-1} A[k]$ is larger or equal than the sum of any other slice



Some definitions...

Computational problem

The formal relationship between the input and the desired output

Algorithm

- The description of the sequence of actions that an executor must execute to solve the problem
- Among their tasks, algorithms represent and organize the input, the output, and all the intermediate data required for the computation

Some history...

- Ahmes' Papyrus (1850 BC, peasant algorithm for multiplication)
- Numerical algorithms have been studied by Babylonians and Indian mathematicians
- Algorithms used even today have been studies by Greek mathematicians more than 2000 years ago
  - Euclid's Algorithm for the greatest common divisor
  - Geometrical algorithms (angle bisection and trisection, tangent drawing, etc)

https://en.wikipedia.org/wiki/Rhind_Mathematical_Papyrus

Algorithms: the name...

Abu Abdullah Muhammad bin Musa al-Khwarizmi

- He was a Persian mathematician, astronomer, astrologer, geographer
- He introduced the indian numbers in the western world
- From his name: algorithm

Al-Kitab al-muhtasar fi hisab al-gabr wa-l-muqabala

- His most famous work (820 AC)
- Translated in Latin with the title: *Liber algebrae et almucabala*

https://en.wikipedia.org/wiki/Muhammad_ibn_Musa_al-Khwarizmi

Computational problems: examples

Minimum

The minimum of a set S is the element of S which is smaller or equal that any other element of S.

$$min(S) = a \Leftrightarrow \exists a \in S : \forall b \in S : a \leq b$$

Looukp

Let $S = s_0, s_1, \ldots, s_{n-1}$ be a sequence of distinct, sorted numbers, i.e. $s_0 < s_1 < \ldots < s_{n-1}$. To perform a lookup of the position of value v in S corresponds to returning the index i such that $0 \leq i < n$, if v is contained at position i, -1 otherwise.

$$lookup(S, v) = \begin{cases} i & \exists i \in \{0, \ldots, n-1\} : S_i = v \\ -1 & \text{otherwise} \end{cases}$$

Computational problems: examples

Minimum

The minimum of a set S is the element of S which is smaller or equal that any other element of S.

$$min(S) = a \Leftrightarrow \exists a \in S : \forall b \in S : a \leq b$$

Looukp

Let $S = s_0, s_1, \ldots, s_{n-1}$ be a sequence of distinct, sorted numbers, i.e. $s_0 < s_1 < \ldots < s_{n-1}$. To perform a lookup of the position of value v in S corresponds to returning the index i such that $0 \leq i < n$, if v is contained at position i, -1 otherwise.

$$lookup(S, v) = \begin{cases} i & \exists i \in \{0, \ldots, n-1\} : S_i = v \\ -1 & \text{otherwise} \end{cases}$$

Note: we described a relationship between input and output. Nothing is said on <u>how to compute the result</u> (that's the difference between math and computer science :-))

Naive solutions

Minimum

To find the minimum of a set, compare each element with every other element; the element that is smaller than any other is the minimum.

Lookup

To find a value v in the sequence S, compare v with any other element of S, in order, and return the corresponding index if a correspondence is found; returns -1 if none of the elements is equal to v.

Computational Problem

↓

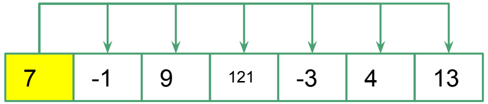
First, let's **translate** the computational problem into an algorithm to solve it.

Then, make it **more efficient,** if possible!

Naive solutions: the code

```
def my_min(S):
    for x in S:
        isMin = True
        for y in S:
            if x > y:
                isMin = False
        if isMin:
            return x

A = [7, -1, 9,121, -3, 4, 13]

print(A)
print("min: {}".format(my_min(A)))
```

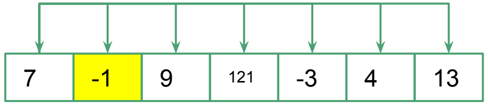
```
[7, -1, 9, 121, -3, 4, 13]
min: -3
```



Naive solutions: the code

```
def my_min(S):
    for x in S:
        isMin = True
        for y in S:
            if x > y:
                isMin = False
        if isMin:
            return x

A = [7, -1, 9,121, -3, 4, 13]

print(A)
print("min: {}".format(my_min(A)))
```

```
[7, -1, 9, 121, -3, 4, 13]
min: -3
```



This code also compares an element with itself...

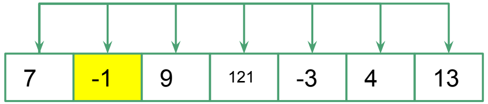
Naive solutions: the code

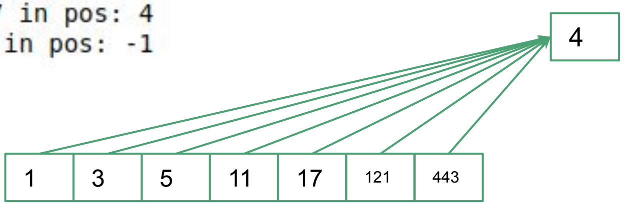
```
def my_min(S):
    for x in S:
        isMin = True
        for y in S:
            if x > y:
                isMin = False
        if isMin:
            return x

A = [7, -1, 9,121, -3, 4, 13]

print(A)
print("min: {}".format(my_min(A)))
```

```
[7, -1, 9, 121, -3, 4, 13]
min: -3
```

```
def lookup(L, v):
    for i in range(len(L)):
        if L[i] == v:
            return i
    return -1

my_list = [1, 3, 5, 11, 17, 121, 443]
print(my_list)
print("{} in pos: {}".format(17,
                             lookup(my_list, 17)))
print("{} in pos: {}".format(4,
                             lookup(my_list, 4)))
```

```
[1, 3, 5, 11, 17, 121, 443]
17 in pos: 4
4 in pos: -1
```





These are direct translations of the computational problems. Can we do better?

Algorithm evaluation

Does it solve the problem in a correct way?

- Mathematical proof vs informal description
- Some problems can only be solved in an approximate way
- Some problems cannot be solved at all

Does it solve the problem in an efficient way?

- How to measure efficiency
- Some solutions are optimal: you cannot find better solutions
- For some problems, there are no efficient solutions

Note on efficiency: algorithm efficiency has a bigger impact on performance than technical details (e.g. using Python vs. C, itertools vs sum etc...)

Efficiency: time and space

Algorithm complexity

Analysis of the resources employed by an algorithm to solve a problem, depending on the size and the type of input

Resources

- Time: time needed to execute the algorithm
  - Should we measure it with a cronometer?
  - Should I measure it by counting the number of elementary operations?
- Space: amount of used memory
- Bandwidth: amount of bit transmitted (distributed algorithms)

we did this to have an "informal" idea of the performance but this is a bad idea because the time depends on very many factors!

A more abstract representation is needed!

Normally, we focus on **time** because there is a relationship between TIME and SPACE. Intuitively, Using N^2 space will require at least N^2 time to read the input… **Normally, TIME > SPACE**

Algorithm evaluation: minimum

How many comparisons do we perform?
Compare each element with ALL the elements

```python
def my_min(S):
    for x in S:
        isMin = True
        for y in S:
            if x > y:
                isMin = False
        if isMin:
            return x

A = [7, -1, 9,121, -3, 4, 13]

print(A)
print("min: {}".format(my_min(A)))
```

```
[7, -1, 9, 121, -3, 4, 13]
min: -3
```

This is the most expensive operation (might work on ints, strings, files,...)

If len(S) = n:
for x in S:
for y in S:
x>y
...
➜ n*n comparisons

Naive algorithm "has complexity": n^2

In more details:
Best case: n comparisons (S[0] min)

Worst case: n*n comparisons (S[-1] is min)

Any ideas?

Can we do better?

Algorithm evaluation: minimum

How many comparisons do we perform?
Compare every element with the OTHER elements

```python
def my_min_v2(S):

    for i in range(len(S)):
        isMin = True
        for j in range(len(S)):
            if i != j:

                if S[i] > S[j]:
                    isMin = False
        if isMin:
            return S[i]

A = [7, -1, 9,121, -3, 4, 13]

print(A)
print("min: {}".format(my_min_v2(A)))
```

```
[7, -1, 9, 121, -3, 4, 13]
min: -3
```

How about this?
Easy, we compare integers!

This is the most expensive operation (might work on ints, strings, files,...)

If len(S) = n:
    for x in 1,...,n:
        for y in 1,...,n:
            if x!=j:
                x>y
            ...

→ **n*(n-1) = n^2 - n** comparisons

Naive algorithm "has complexity": n^2
This algorithm: n^2 - n

In more details:
Best case: n-1 comparisons (S[0] min)

Worst case: n*(n-1) comparisons (S[-1] is min)

Can we do better?

Algorithm evaluation: minimum, a better solution

How many comparisons do we perform?
Accumulate the minimum found so far and compare the others with it

```
def my_faster_min(S):
    min_so_far = S[0] #first element
    i = 1
    while i < len(S):
        if S[i] < min_so_far:
            min_so_far = S[i]
        i = i +1
    return min_so_far

A = [7, -1, 9,121, -3, 4, 13]

print(A)
print("min: {}".format(my_min(A)))
```

```
[7, -1, 9, 121, -3, 4, 13]
min: -3
```

This is the most expensive operation (might work on ints, strings, files,...)

If len(S) = n:
while i= 1,...,n-1
S[i] < min_so_far

➜ n-1 comparisons

Naive algorithm "has complexity": n^2 - n

Better algorithm "has complexity": n-1
(regardless if S[0] is the min of if S[-1] is)

Algorithm evaluation: lookup

How many comparisons do we perform?

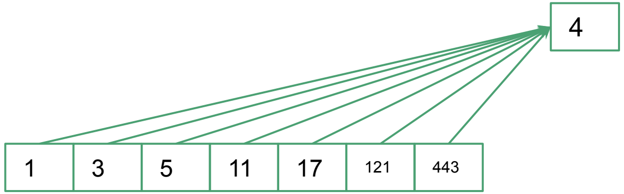
```
def lookup(L, v):
    for i in range(len(L)):
        if L[i] == v:
            return i
    return -1

my_list = [1, 3, 5, 11, 17, 121, 443]
print(my_list)
print("{} in pos: {}".format(17,
                             lookup(my_list, 17)))
print("{} in pos: {}".format(4,
                             lookup(my_list, 4)))
```

```
[1, 3, 5, 11, 17, 121, 443]
17 in pos: 4
4 in pos: -1
```

I compare v with first element, then to the second etc. when I find it or when I checked the whole list I stop.

➔ n comparisons

Naive algorithm "has complexity": n
(in the worst case: v is not there!)



Algorithm evaluation: lookup, better solution (?)

How many comparisons do we perform?

```
def lookup(L, v):
    for i in range(len(L)):
        if L[i] == v:
            return i
        elif L[i] > v:
            return -1
    return -1

my_list = [1, 3, 5, 11, 17, 121, 443]
print(my_list)
print("{} in pos: {}".format(17,
                             lookup(my_list, 17)))
print("{} in pos: {}".format(4,
                             lookup(my_list, 4)))

print("{} in pos: {}".format(500,
                             lookup(my_list, 4)))
```

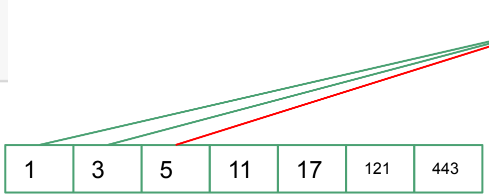
```
[1, 3, 5, 11, 17, 121, 443]
17 in pos: 4
4 in pos: -1
500 in pos: -1
```

The list is sorted. I loop through the list, if I find value > v I can stop.

Generally faster, if L is big (i.e. n is big) , but worst case (es. 500 below)

➜ 2*n comparisons

Naive algorithm “has complexity”: n
“Better” algorithm “has complexity”: 2*n



Algorithm evaluation: best, worst and average case

What is the most important case?

Best: lookup(L,1) solved in 1 step

1	2	5	6	7	8	9

Not interested. We are never lucky!!!

Worst: lookup(L,10) solved in 9 steps

1	2	5	6	7	8	9

Normally, **the most informative case**

Average: lookup(L,6) solved in 4 steps

1	2	5	6	7	8	9

Sometimes interesting

Lookup: more efficient algorithm

The list is sorted...

lookup(L,v)

ex. lookup(L,28)

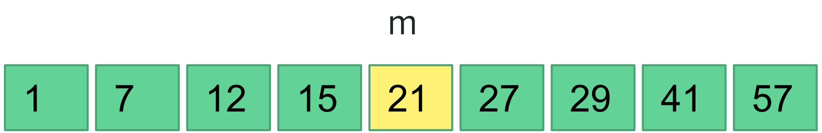
1	7	12	15	21	27	29	41	57

Lookup: a more efficient algorithm

The list is sorted...

lookup(L,v)

ex. lookup(L,28)



Let’s start considering the **median value**, m.

If L[m] = v. Found it!

if L[m] > v. Search L[0:m]

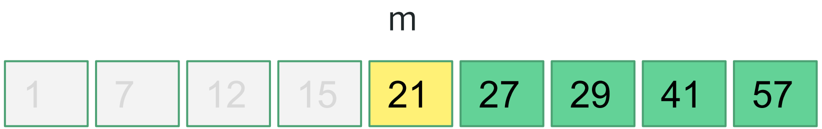
if L[m] <v. Search L[m+1:]

Lookup: a more efficient algorithm

The list is sorted...

lookup(L,v)

ex. lookup(L,28)



Let’s start considering the **median value**, m.

If L[m] = v. Found it!

if L[m] > v. Search L[0:m]

21 < **28** → ignore L[0:m]

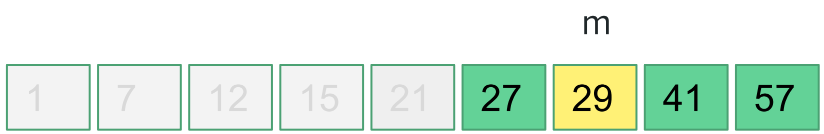
if L[m] <v. Search L[m+1:]

Lookup: a more efficient algorithm

The list is sorted…

lookup(L,v)

ex. lookup(L,28)



Let's start considering the **median value**, m.

If L[m] = v. Found it!

if L[m] > v. Search L[0:m]

28 < 29 → ignore L[m+1:]

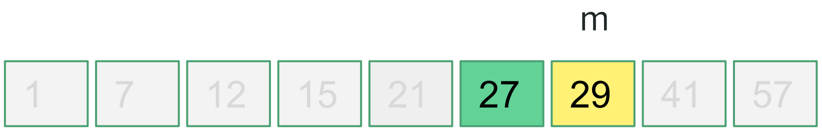
if L[m] <v. Search L[m+1:]

Lookup: a more efficient algorithm

The list is sorted...

lookup(L,v)

ex. lookup(L,28)



Let's start considering the **median value**, m.

If L[m] = v. Found it!

if L[m] > v. Search L[0:m]

28 < 29 ➔ ignore L[m+1:]

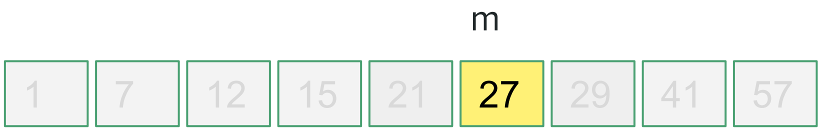
if L[m] <v. Search L[m+1:]

Lookup: a more efficient algorithm

The list is sorted...

lookup(L,v)

ex. lookup(L,28)



Let's start considering the **median value**, m.

If L[m] = v. Found it!

if L[m] > v. Search L[0:m]

if L[m] <v. Search L[m+1:]

27 != **28** → NOT FOUND

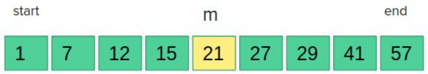
Lookup: the recursive code (binary search)

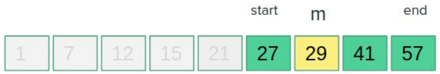
```
def lookup_rec(L, v, start,end):
    if end < start:
        return -1
    else:
        m = (start + end)//2
        if L[m] == v: #found!
            return m
        elif v < L[m]: #look to the left
            return lookup_rec(L, v, start, m-1)
        else: #look to the right
            return lookup_rec(L, v, m+1, end)




my_list = [1, 3, 5, 11, 17, 121, 443]
print(my_list)
print("{} in pos: {}".format(17,
                             lookup_rec(my_list, 17, 0, len(my_list)-1)))
print("{} in pos: {}".format(4,
                             lookup_rec(my_list, 4, 0, len(my_list)-1)))

print("{} in pos: {}".format(443,
                             lookup_rec(my_list, 443, 0, len(my_list)-1)))
```

```
[1, 3, 5, 11, 17, 121, 443]
17 in pos: 4
4 in pos: -1
443 in pos: 6
```

when only one element left
start = end = m
at next iteration end < start

v = 28





Lookup: the recursive code (binary search)

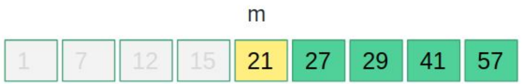
```
def lookup_rec(L, v, start,end):
    if end < start:
        return -1
    else:
        m = (start + end)//2
        if L[m] == v: #found!
            return m
        elif v < L[m]: #look to the left
            return lookup_rec(L, v, start, m-1)
        else: #look to the right
            return lookup_rec(L, v, m+1, end)



my_list = [1, 3, 5, 11, 17, 121, 443]
print(my_list)
print("{} in pos: {}".format(17,
                             lookup_rec(my_list, 17, 0, len(my_list)-1)))
print("{} in pos: {}".format(4,
                             lookup_rec(my_list, 4, 0, len(my_list)-1)))

print("{} in pos: {}".format(443,
                             lookup_rec(my_list, 443, 0, len(my_list)-1)))
```

```
[1, 3, 5, 11, 17, 121, 443]
17 in pos: 4
4 in pos: -1
443 in pos: 6
```

2 comparisons (==, <) at each call

How many total comparisons?

Anyone wants to try?



Lookup: the recursive code (binary search)

```python
def lookup_rec(L, v, start,end):
    if end < start:
        return -1
    else:
        m = (start + end)//2
        if L[m] == v: #found!
            return m
        elif v < L[m]: #look to the left
            return lookup_rec(L, v, start, m-1)
        else: #look to the right
            return lookup_rec(L, v, m+1, end)



my_list = [1, 3, 5, 11, 17, 121, 443]
print(my_list)
print("{} in pos: {}".format(17,
                             lookup_rec(my_list, 17, 0, len(my_list)-1)))
print("{} in pos: {}".format(4,
                             lookup_rec(my_list, 4, 0, len(my_list)-1)))

print("{} in pos: {}".format(443,
                             lookup_rec(my_list, 443, 0, len(my_list)-1)))
```

```
[1, 3, 5, 11, 17, 121, 443]
17 in pos: 4
4 in pos: -1
443 in pos: 6
```

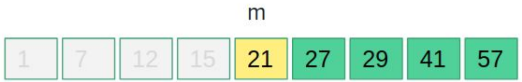
2 comparisons (==, <) at each call

How many total comparisons?

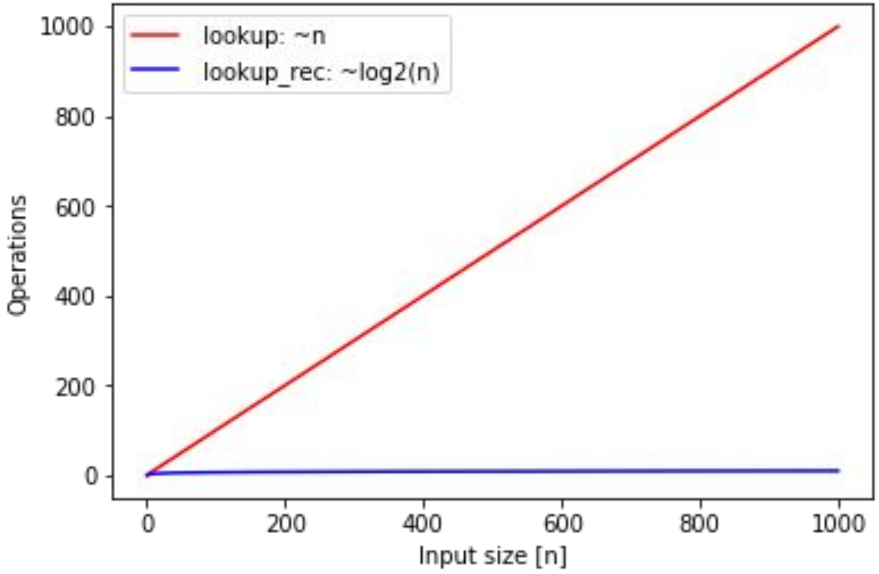
At beginning 1024 elements...
then 512...
then 256...
then 128...
then 64...
then 32...
then 16...
then 8...
then 4...
then 2...
then 1

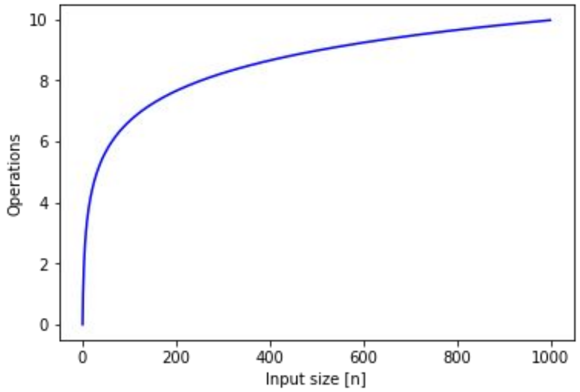
➜ log2(1024) +1 iterations

“Complexity” ~ 2 * log2 n



Lookup analysis





Correctness

Invariant

A condition that is always true in a specific point in an algorithm

Loop invariant

- A condition that is always true at the beginning of a loop iteration
- what is exactly the beginning of a loop iteration?

Class invariant

- A condition always true when the execution of a class method is completed

Correctness

The **loop invariant** helps us proving that an iterative algorithm is **correct:**

By induction...

Initialization (base case):

Prove that the condition is true before the first iteration

Conservation (inductive step):

If the condition is true before the iteration of the loop, then **prove** that it remains true at the end (before the next iteration)

Conclusion:

At the end, **the invariant** must represent the "correctness" of the algorithm

Correctness of min

Invariant: At the beginning of **iteration i** of the while loop, min_so_far contains the partial minimum of the elements in S[0:i].

```
def my_faster_min(S):
    min_so_far = S[0] #first element
    i = 1
    while i < len(S):
        if S[i] < min_so_far:
            min_so_far = S[i]
        i = i +1
    return min_so_far

A = [7, -1, 9,121, -3, 4, 13]

print(A)
print("min: {}".format(my_min(A)))
```

```
[7, -1, 9, 121, -3, 4, 13]
min: -3
```

Base case:

min_so_far = S[0] **IS** the minimum of elements in S[0:1]

Induction step:

(assuming min_so_far is the minimum of S[0:i]) at each iteration i, min_so_far is updated **IFF S[i] < min_so_far**

→ **min_so_far always contains min of elements S[0:i]**

Correctness of lookup

Exercise: prove the correctness of lookup_rec

```
def lookup_rec(L, v, start,end):
    if end < start:
        return -1
    else:
        m = (start + end)//2
        if L[m] == v: #found!
            return m
        elif v < L[m]: #look to the left
            return lookup_rec(L, v, start, m-1)
        else: #look to the right
            return lookup_rec(L, v, m+1, end)



my_list = [1, 3, 5, 11, 17, 121, 443]
print(my_list)
print("{} in pos: {}".format(17,
                             lookup_rec(my_list, 17, 0, len(my_list)-1)))
print("{} in pos: {}".format(4,
                             lookup_rec(my_list, 4, 0, len(my_list)-1)))

print("{} in pos: {}".format(443,
                             lookup_rec(my_list, 443, 0, len(my_list)-1)))
```

```
[1, 3, 5, 11, 17, 121, 443]
17 in pos: 4
4 in pos: -1
443 in pos: 6
```

This is a recursive code, we cannot use the loop invariant but can still prove the correctness by induction.

The induction can be done on the number of elements in the list:

n = end - start

Correctness of lookup

```
def lookup_rec(L, v, start,end):
    if end < start:
        return -1
    else:
        m = (start + end)//2
        if L[m] == v: #found!
            return m
        elif v < L[m]: #look to the left
            return lookup_rec(L, v, start, m-1)
        else: #look to the right
            return lookup_rec(L, v, m+1, end)
```

Exercise: prove the correctness of lookup_rec.
By induction on **n = end - start**

Base case (n = 0)

Inductive hypothesis: given a size n, let us assume that the algorithm is correct for all sizes n’ < n

Inductive step: given inductive hypothesis, prove invariant still holds for size n.

Correctness of lookup

```
def lookup_rec(L, v, start,end):
    if end < start:
        return -1
    else:
        m = (start + end)//2
        if L[m] == v: #found!
            return m
        elif v < L[m]: #look to the left
            return lookup_rec(L, v, start, m-1)
        else: #look to the right
            return lookup_rec(L, v, m+1, end)
```

Exercise: prove the correctness of lookup_rec.
By induction on **n = end - start**

Base case (n = 0): if n == 0, this means that **end < start**.
The algorithm **returns −1**. Correct given that **if n == 0, v is not present.**

Inductive hypothesis: given a size n, let us assume that the algorithm is correct **for all sizes n' < n**

Inductive step: given a size n > 0, let m be the median element.

If L[m]==v, then the algorithm returns m, because m is the actual position of v —> hence v is in m = start+end//2 that **is in L[start:end]**

If v < L[m], then if v is present, since S is sorted, it must be located in **L[start:m]**. By inductive hypothesis, **lookup_rec(L, v,start, m-1) will return the correct position** of v if present, or -1 if not present (since **m-1 - start is smaller than n**).

if v > L[m] is symmetric.