

Scientific Programming: Algorithms (part B)

Introduction

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

Organization

145540 Scientific Programming (12 ECTS, LM QCB)

145685 Scientific Programming (12 ECTS, LM Data Science)

Part A - Programming (23/9-31/10)

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 (4/11-12/12)

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. Massimiliano Luca
 - Lab sessions on algorithms (QCB)
 - massimiliano.luca [AT] unitn.it
- Teaching assistant: Dr. David Leoni
 - Lab sessions on algorithms (data science)
 - david.leoni [AT] unitn.it

Schedule

Week day	Time	Room	Description
Monday	14.30-16.30	A107	Lecture
Tuesday	15.30-17.30	A107	Lab. QCB
Tuesday	15.30-17.30	A103	Lab. Data Science
Wednesday	11.30-13.30	A107	Lecture
Thursday	15.30-17.30	A107	Lab. QCB
Thursday	15.30-17.30	A208	Lab. Data Science



midterms:

Part A (tomorrow morning — no lab in the afternoon)

Part B (tentatively ~ December, 17th or 19th)

Course material

Lectures:

Material and information: <https://sciproalgo2019.readthedocs.io/en/latest/>

Practicals:

QCB: <https://massimilianoluca.github.io/algoritmi/index.html>

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

Course material

<https://sciproalgo2019.readthedocs.io/en/latest/>

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 Mondays from 14:30 to 16:30 (in lecture room A107) and on Wednesdays from 11:30 to 13:30 (in lecture room A107). This second part of the Scientific Programming course will tentatively run from 06/11/2019 to 20/12/2019.

Slides

The slides shown during the lectures will gradually appear below:

- Lecture 1: [Introduction to algorithms](#)

Teaching assistants

[David Leoni](#) (for Data Science)

[Massimiliano Luca](#) (for QCB)

Course material

Brad Miller and David Ranum. *Problem Solving with Algorithms and Data Structures using Python*. An interactive version is freely available at [this link](#).

Other material includes the following books:

- Lutz. *Learning Python* (5th edition). O'REILLY (2013)
- Hetland. *Python Algorithms: Mastering Basic Algorithms in the Python Language*. Apress, 2nd

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 (**correctness**), possibly in an efficient way (**complexity**), organizing data in the most suitable ways (**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:

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:

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 $\sim N^3$

Idea:

Given the list A with N elements

Consider all pairs (i,j) such that $i \leq j$

Get the elements in $A[i:j+1]$

Compute the sum of all elements in $A[i:j+1]$

Update max_so_far if $\text{sum} \geq \text{max_so_far}$

```
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

```
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

```
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

```
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... ?

No thanks!

- **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_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?

$$N*(N+1)/2 \sim N^2$$

```
[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!
```

If A has 100,000 elements $\rightarrow \sim 40$ 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

```
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
```

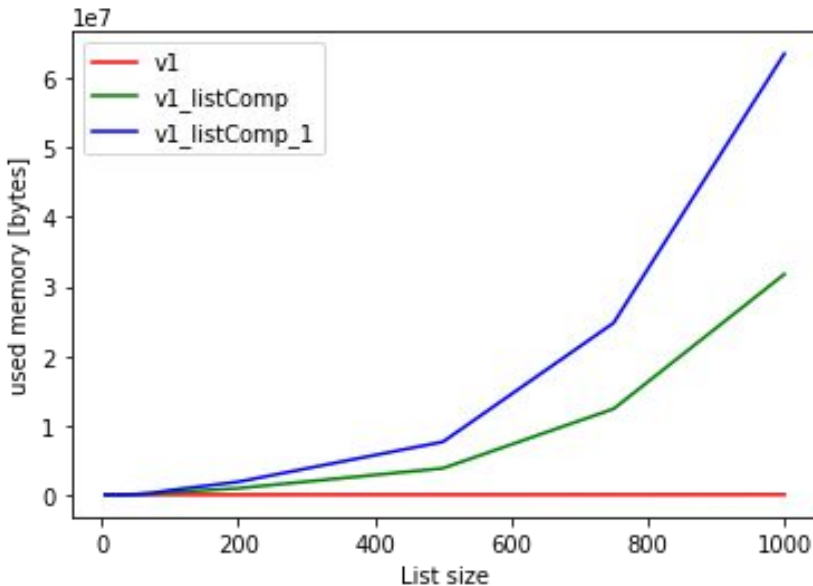


Stores intervals and
sums!!!

If A has 100,000 elements $\rightarrow \sim 1.3$ 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



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

Important note:

Time and space (memory) are two important resources!

Solution 1 $\sim N^3$

Idea:

Given the list A with N elements

Consider all pairs (i,j) such that $i \leq j$

Get the elements in $A[i:j+1]$

Compute the sum of all elements in $A[i:j+1]$

Update max_so_far if $\text{sum} \geq \text{max_so_far}$

```
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

Why N^3 ?

Intuitively,

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

So: $N*[(N+1)/2]*N \sim N^3$

Solution 2 $\sim N^2$

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

If $S = \text{sum}(A[i:j]) \rightarrow \text{sum}(A[i:j+1]) = S + A[j+1]$

- **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_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 $\sim N^2$

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

If $S = \text{sum}(A[i:j]) \rightarrow \text{sum}(A[i:j+1]) = S + A[j+1]$

```
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

Tot	(i, j)
0, 1, 4, 8, 0, 2, 5, 4, 7, 11, 8, 18, 15, 17,	$\leftarrow (0, x)$
0, 3, 7, -1, 1, 4, 3, 6, 10, 7, 17, 14, 16,	$\leftarrow (1, x)$
0, 4, -4, -2, 1, 0, 3, 7, 4, 14, 11, 13,	$\leftarrow (2, x)$
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	$\leftarrow (N-1, x)$

Maxes

[1, 4, 8, 8, 8, 8, 8, 8, 11, 11, 18, 18, 18, ..., 18]

Solution 2 $\sim N^2$

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

If $S = \text{sum}(A[i:j]) \rightarrow \text{sum}(A[i:j+1]) = S + A[j+1]$

```
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

Intuitively, we have to consider $N*(N+1)/2 \sim N^2$ intervals

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

Solution 2 $\sim N^2$

Tip: use itertools

Accumulate of itertools is done in C so it is faster

- **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 = list(range(10))
```

```
print(A)
```

```
print(list(accumulate(A)))
```

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

```
[0, 1, 3, 6, 10, 15, 21, 28, 36, 45]
```


Solution 2 $\sim N^2$

Tip: use itertools

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

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[0, 1, 3, 6, 10, 15, 21, 28, 36, 45]
```



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

Important note: N intervals, sum of N elements each time: $\sim N^2$ operations

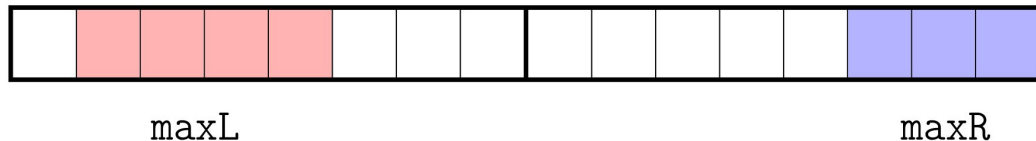
The improvement comes from implementation not algorithm! (code faster by a constant factor)

Solution 3 $\sim N \log(N)$

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



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

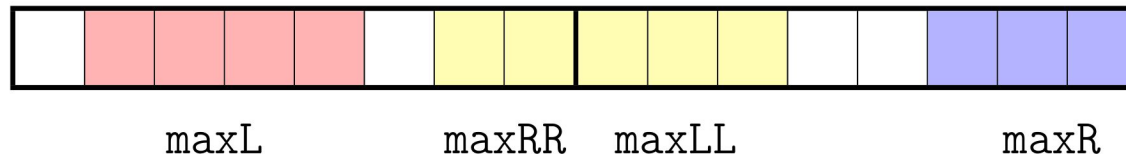
- **Input:** a list A containing n numbers
- **Output:** a slice (sublist) $A[i : j]$ of maximal sum, i.e. the slice whose element sum $\sum_{k=i}^{j-1} A[k]$ is larger or equal than the sum of any other slice

Solution 3 $\sim N \log(N)$

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



- **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

Solution 3 $\sim N \log(N)$

Divide et impera (Divide and conquer)

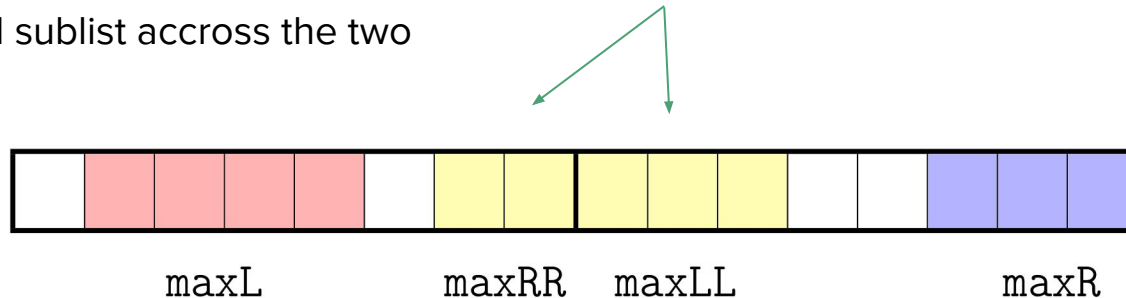
Idea:

- 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

- **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

Get the point before the mid-point M and go to the left until the sum increases.
Repeat starting from $M+1$.

Result is: $\max(\maxL, \maxRR, \maxLL, \maxR)$



Solution 3 $\sim N \log(N)$

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)

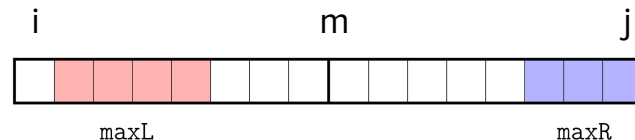
A = [1, 3, 4, -8, 2, 3, -1, 3, 4, -3, 10, -3, 2]
print(A)
print(max_sum_v3(A))

[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 $\sim N \log(N)$

Divide et impera (Divide and conquer)

```
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
```

- **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

0	1	2	3	4	5	6	7	8	9	10	11	12	13															
<table><tr><td>L</td><td>u</td><td>t</td><td>h</td><td>e</td><td>r</td><td></td><td>C</td><td>o</td><td>l</td><td>i</td><td>l</td><td>e</td><td>g</td><td>e</td></tr></table>														L	u	t	h	e	r		C	o	l	i	l	e	g	e
L	u	t	h	e	r		C	o	l	i	l	e	g	e														
-14	-13	-12	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1															



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

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

Dynamic Programming

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

The result is computed from the **maximum slice that ends in any position**.

- **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

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

Solution 4 ~ N

Dynamic Programming

Let's define $\text{maxHere}[i]$ as the maximum value of each sublist that ends in i .

The result is computed from the maximum slice that ends in any position.

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

Goes through A once: runs in **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

```
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(A)  
print(max_sum_v3(A))
```

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


Solution 4 \sim N

Dynamic Programming

```
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(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

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, 15, 17]
max_so_far: [0, 1, 4, 8, 8, 8, 8, 8, 8, 11, 11, 18, 18]

Solution 4 ~ N

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)
```

- **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

A: [1, 3, 4, -8, 2, 3, -1, 3, 4, -3, 10, -3, 2]

Max_so_far: [0, 1, 4, 8, 8, 8, 8, 8, 8, 11, 11, 18, 18, 18]

Max_here: [0, 1, 4, 8, 0, 2, 5, 4, 7, 11, 8, 18, 15, 17]

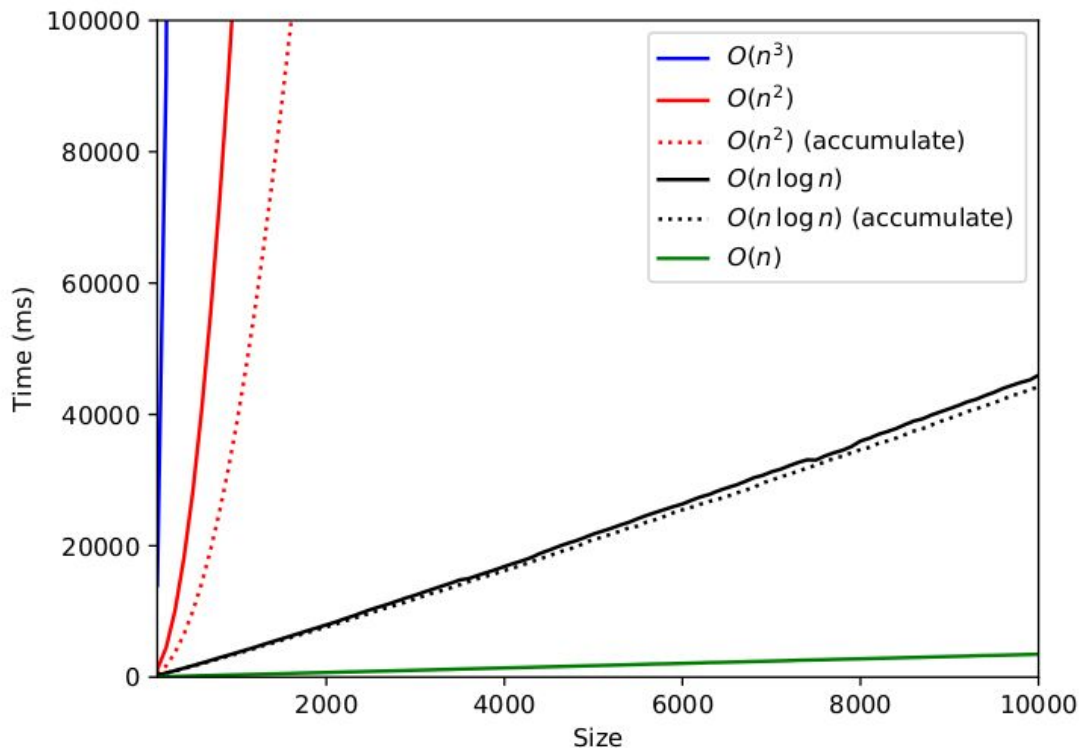
Last: [0, 0, 0, 0, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4]

Start: [0, 0, 0, 0, 0, 0, 0, 0, 0, 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 studied 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)



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*



Computational problems: examples

Minimum

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

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

Lookup

Let $S = s_0, s_1, \dots, s_{n-1}$ be a sequence of distinct, sorted numbers, i.e. $s_0 < s_1 < \dots < s_{n-1}$. 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.

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

Computational problems: examples

Minimum

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

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

Lookup

Let $S = s_0, s_1, \dots, s_{n-1}$ be a sequence of distinct, sorted numbers, i.e. $s_0 < s_1 < \dots < s_{n-1}$. 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.

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

Note: we described a relationship between input and output. Nothing is said on how to compute the result (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
```

```
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
```

This is a direct translation of the computational problem. 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)

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?

```
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 $\text{len}(S) = n$:

- for x in $1, \dots, n$:
- for y in $1, \dots, n$:
- $x > y$
- ...

→ $n * n$ comparisons

Naive algorithm has complexity: n^2

Algorithm evaluation: minimum, a better solution

How many comparisons do we perform?

```
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 $\text{len}(S) = n$:
 $i = 1, \dots, n-1$
 $S[i] < \text{min_so_far}$

→ $n-1$ comparisons

Naive algorithm “has complexity”: n^2

Better algorithm “has complexity”: $n-1$

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 I stop.

→ n comparisons

Naive algorithm “has complexity”: n

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
```

I loop through the list, if I find value $> v$ I can stop.

Generally faster, but worst case (es. 500 below)

→ n comparisons

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

Algorithm evaluation: best, worst and average case

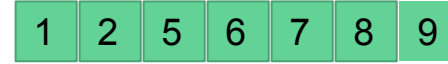
What is the most important case?

Best: $\text{lookup}(L,1)$ solved in 1 step.



Not interested.
We are never
lucky!!!

Worst: $\text{lookup}(L,10)$ solved in 9 steps



Normally, **the
most informative
case**

Average: $\text{lookup}(L,6)$ solved in 4 steps



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 $L[0:m]$

$21 < \mathbf{28} \rightarrow$ 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 $L[m+1:]$

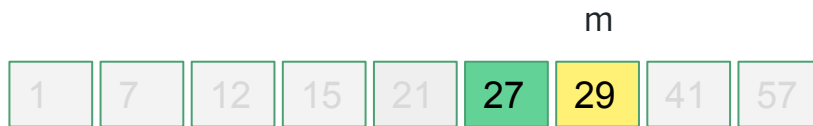
if $L[m] < v$. Search $L[m+1:]$

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 \neq **28** → NOT FOUND

Lookup: the recursive code

```
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)
```



can stop and check when $end == start$
but it is similar

```
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
```


Lookup: the recursive code

```
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 comparisons?

Anyone wants to try?

Lookup: the recursive code

```
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 comparisons?

At beginning 1024 elements...

then 512...

then 256...

then 128...

then 64...

then 32...

then 16...

then 8...

then 4...

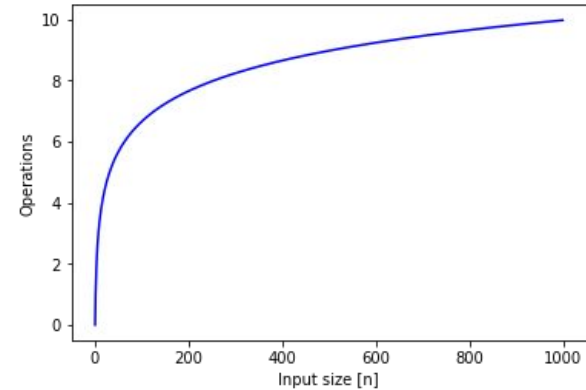
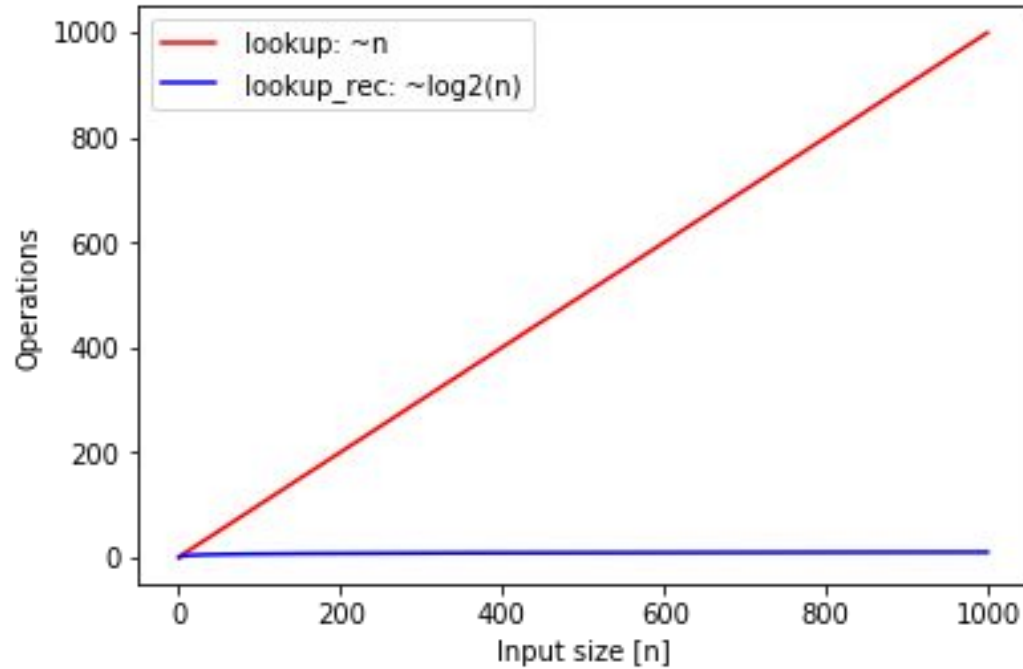
then 2...

then 1

→ $\log_2(1024) + 1$ iterations

Complexity $\sim \log_2 n$

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 the 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 each iteration 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:

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
```

What is the invariant?

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
```

What is the invariant?

If v is in L , it is located in $L[\text{start}:\text{end}+1]$

Correctness of lookup

Exercise: prove the correctness of `lookup_rec`.
By induction on $n = \text{end} - \text{start}$

Base case ($n = 0$): if $n == 0$, this means that $\text{end} < \text{start}$.
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 $S[m] == v$, then the algorithm returns m , because m is the actual position of v

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

if $v > S[m]$ is symmetric.

```
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)
```

Questions?

