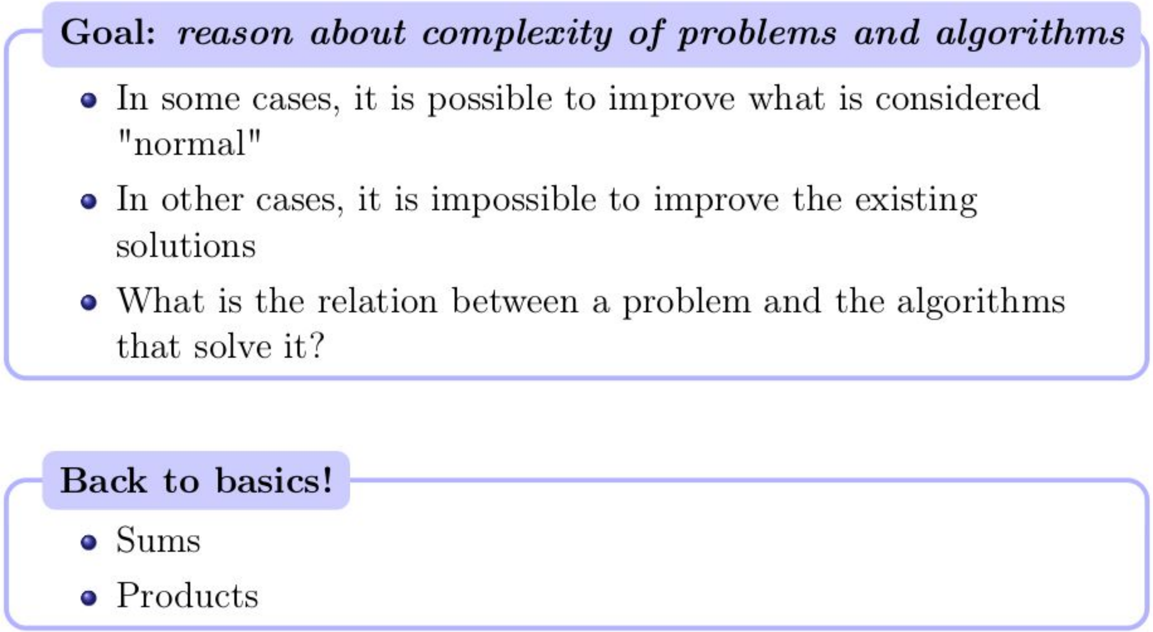
Scientific Programming: Part B

Lecture 3

Luca Bianco - Academic Year 2019-20
luca.bianco@fmach.it

[credits: thanks to Prof. Alberto Montresor]

Problem vs. algorithm complexity



Goal: ***reason about complexity of problems and algorithms***

- In some cases, it is possible to improve what is considered "normal"
- In other cases, it is impossible to improve the existing solutions
- What is the relation between a problem and the algorithms that solve it?

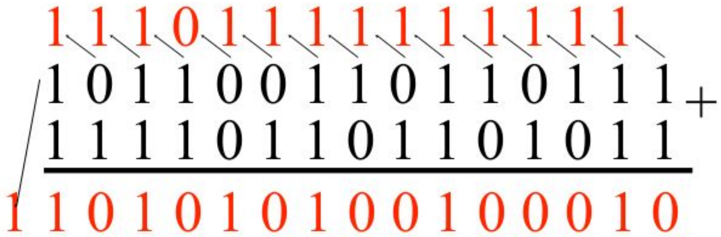
Back to basics!

- Sums
- Products

Sum of binary (or any other base) numbers

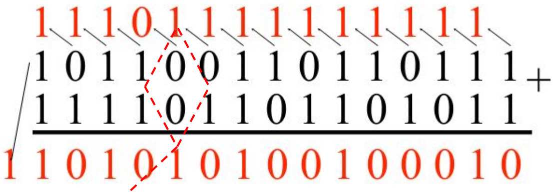
Basic sum algorithm – sum()

- each of the n bits have to be considered
- total cost is equal to cn ($c \equiv$ the cost required to sum three bits and generate the carry-over)



Note: in programming languages like python the sum is a basic operation, up to the biggest possible integer it is done by the CPU.
After that → arbitrary precision and therefore we would need an algorithm like this.

Sum of binary (or any other base) numbers



Question

Is there a better method?

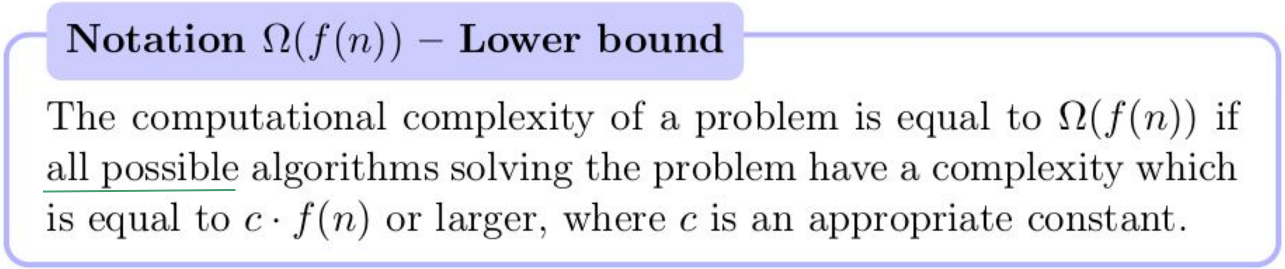
Nope. There is no better way (improvements like grouping bits just deliver better constants)

Sketch of the proof. Reasoning by contradiction:

since to compute the result I have to consider all bits,
if there existed a better method, it would skip some bits

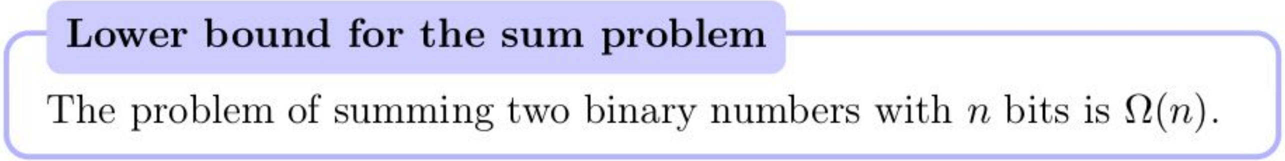
→ hence the solution might not be correct if only those bits are changed!

Lower bound to the complexity of a problem



Notation $\Omega(f(n))$ – Lower bound

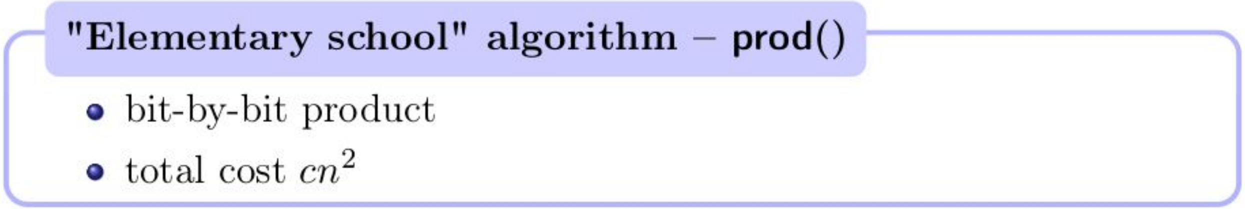
The computational complexity of a problem is equal to $\Omega(f(n))$ if all possible algorithms solving the problem have a complexity which is equal to $c \cdot f(n)$ or larger, where c is an appropriate constant.



Lower bound for the sum problem

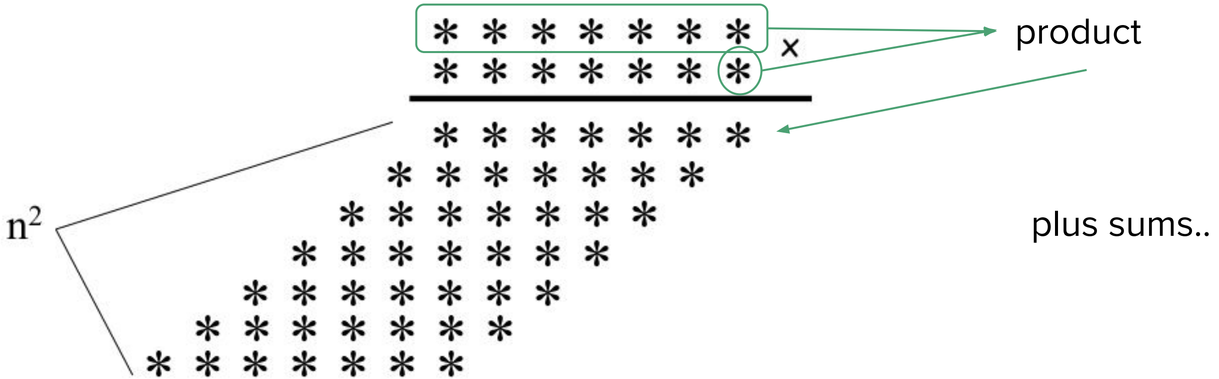
The problem of summing two binary numbers with n bits is $\Omega(n)$.

Product of binary (or any other base) numbers

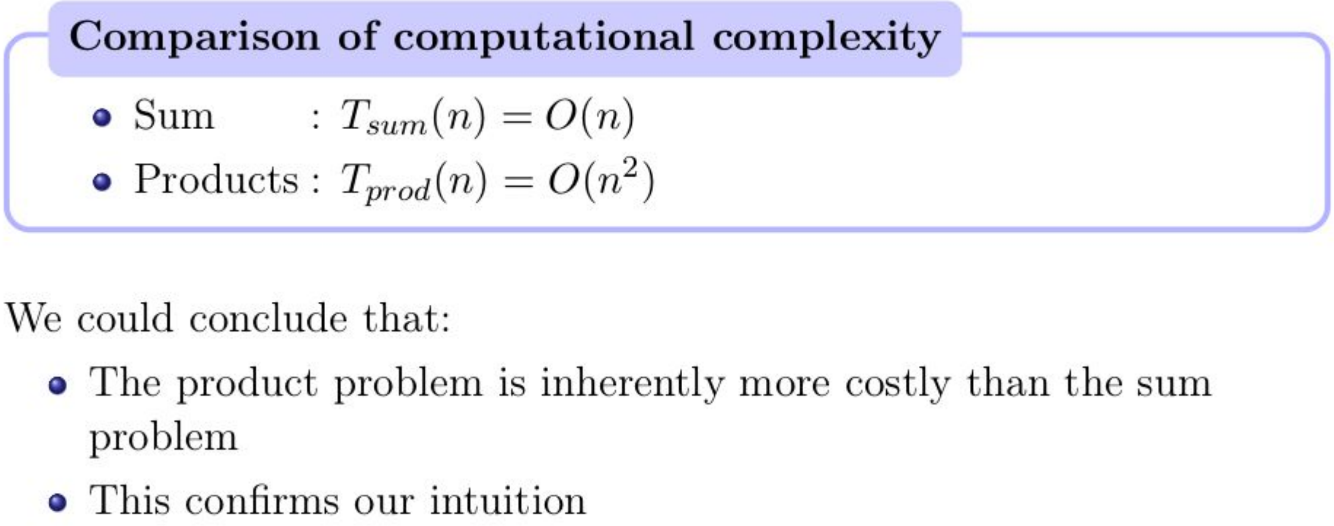


"Elementary school" algorithm – prod()

- bit-by-bit product
- total cost cn^2



Arithmetic algorithms



Comparison of computational complexity

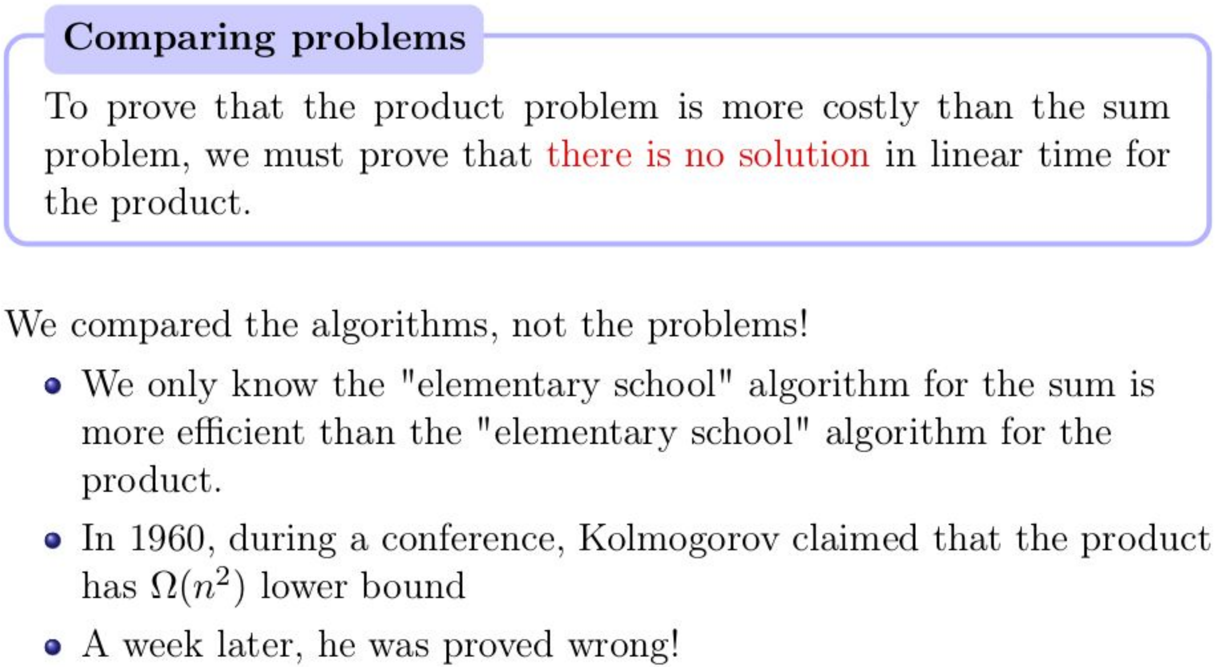
- Sum : $T_{sum}(n) = O(n)$
- Products : $T_{prod}(n) = O(n^2)$

We could conclude that:

- The product problem is inherently more costly than the sum problem
- This confirms our intuition

Wrong. We are comparing problems but we here in fact we have solutions. What I am saying is that **THIS solution** to compute the product is more costly than the sum!!!

Arithmetic algorithms



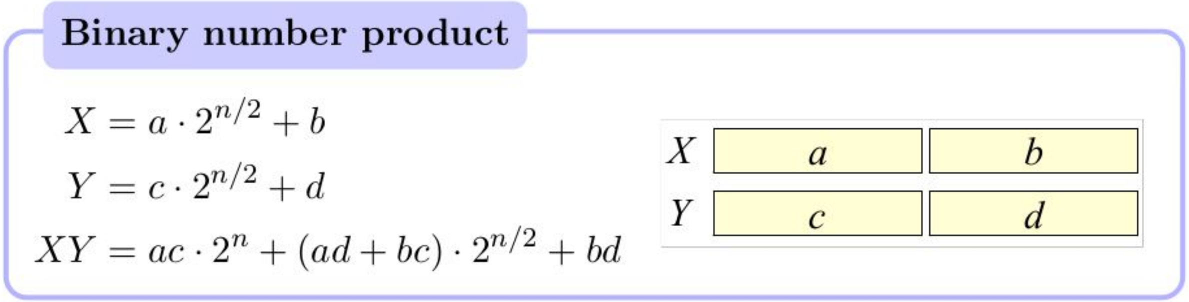
Comparing problems

To prove that the product problem is more costly than the sum problem, we must prove that there is no solution in linear time for the product.

We compared the algorithms, not the problems!

- We only know the "elementary school" algorithm for the sum is more efficient than the "elementary school" algorithm for the product.
- In 1960, during a conference, Kolmogorov claimed that the product has $\Omega(n^2)$ lower bound
- A week later, he was proved wrong!

Product of binary numbers - Divide-et-impera



Binary number product

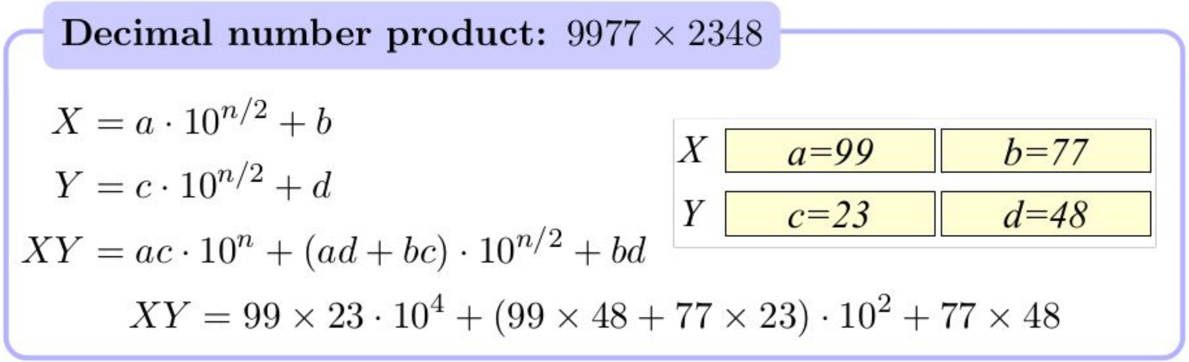
$$X = a \cdot 2^{n/2} + b$$

$$Y = c \cdot 2^{n/2} + d$$

$$XY = ac \cdot 2^{n} + (ad + bc) \cdot 2^{n/2} + bd$$

X	a	b
Y	c	d

Split the numbers in 2. (**n** is the number of digits).
The most significant and least significant part.



Decimal number product: 9977×2348

$$X = a \cdot 10^{n/2} + b$$

$$Y = c \cdot 10^{n/2} + d$$

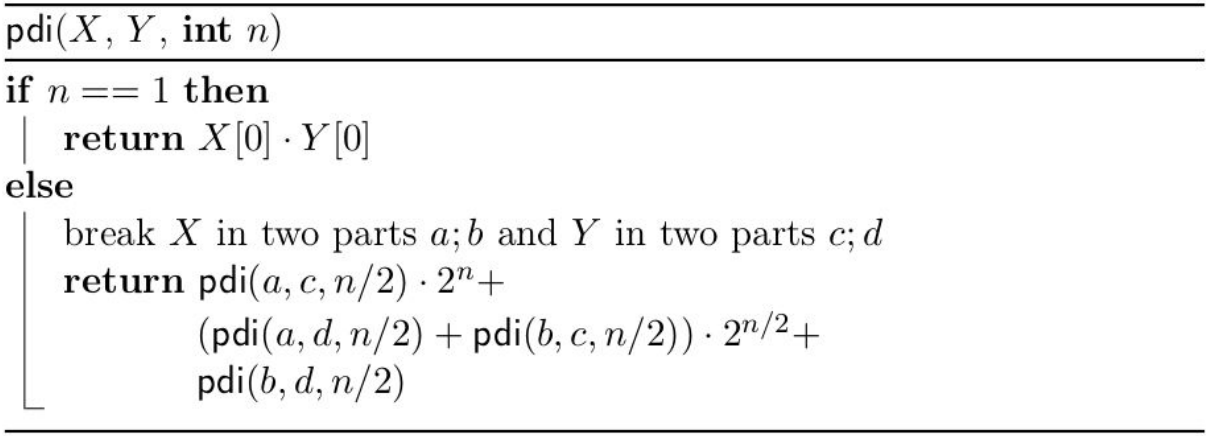
$$XY = ac \cdot 10^{n} + (ad + bc) \cdot 10^{n/2} + bd$$

X	a=99	b=77
Y	c=23	d=48

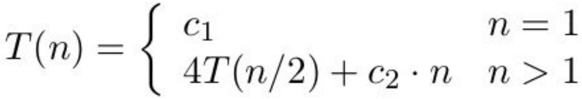
$$XY = 99 \times 23 \cdot 10^{4} + (99 \times 48 + 77 \times 23) \cdot 10^{2} + 77 \times 48$$

Apparently we now have 4 multiplications (multiply by 2^n or 2^n/2 is actually moving the digits: **shift**)

Product of binary numbers - Divide-et-impera



```
pdi($X$, $Y$, int $n$)
if $n == 1$ then
    return $X[0] \cdot Y[0]$
else
    break $X$ in two parts $a; b$ and $Y$ in two parts $c; d$
    return pdi($a, c, n/2$) $\cdot 2^n +$
           (pdi($a, d, n/2$) $+$ pdi($b, c, n/2$)) $\cdot 2^{n/2} +$
           pdi($b, d, n/2$)
```



$$T(n) = \begin{cases} c_1 & n = 1 \\ 4T(n/2) + c_2 \cdot n & n > 1 \end{cases}$$

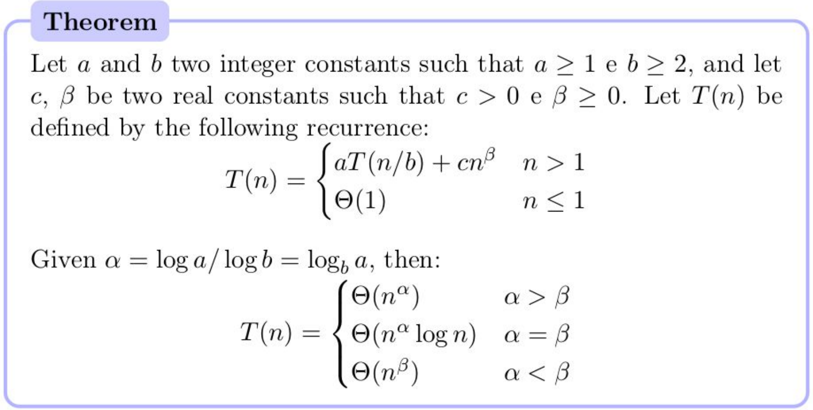
Recursive code for n/2 bits.

Recombination of results: sums and multiplications by 2^n/2 ➔ linear cost.

Note: multiplication by **2^t** is **linear (shift) of t positions**

Product of binary numbers - Divide-et-impera

$$T(n) = \begin{cases} c_1 & n = 1 \\ 4T(n/2) + c_2 \cdot n & n > 1 \end{cases}$$



Theorem

Let a and b two integer constants such that $a \geq 1$ e $b \geq 2$, and let c, β be two real constants such that $c > 0$ e $\beta \geq 0$. Let $T(n)$ be defined by the following recurrence:

$$T(n) = \begin{cases} aT(n/b) + cn^\beta & n > 1 \\ \Theta(1) & n \leq 1 \end{cases}$$

Given $\alpha = \log a / \log b = \log_b a$, then:

$$T(n) = \begin{cases} \Theta(n^\alpha) & \alpha > \beta \\ \Theta(n^\alpha \log n) & \alpha = \beta \\ \Theta(n^\beta) & \alpha < \beta \end{cases}$$

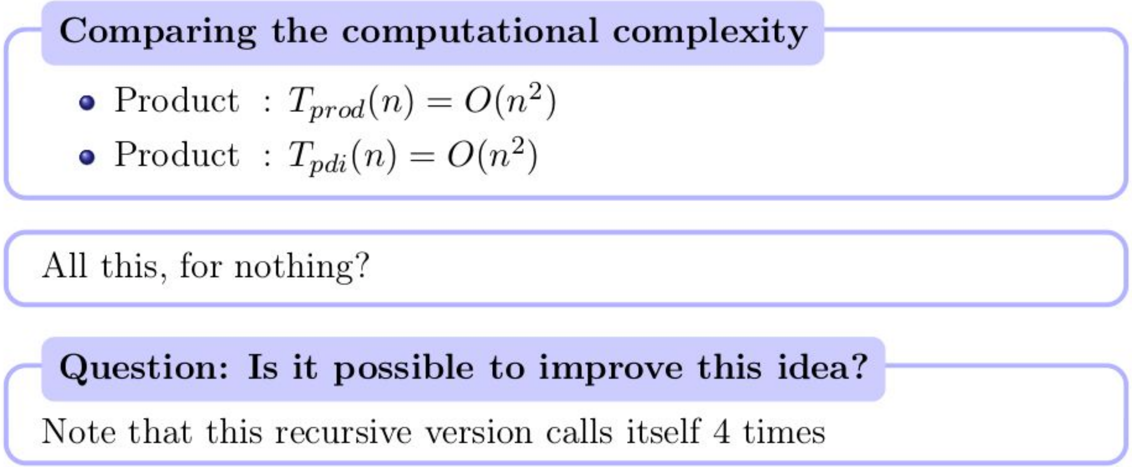
$$\alpha = \log_2 4 = 2$$

$$\beta = 1$$

$$T(n) = \Theta(n^2)$$

was all this mess pointless?!?

Product of binary numbers - Divide-et-impera



Comparing the computational complexity

- Product : $T_{prod}(n) = O(n^2)$
- Product : $T_{pdi}(n) = O(n^2)$

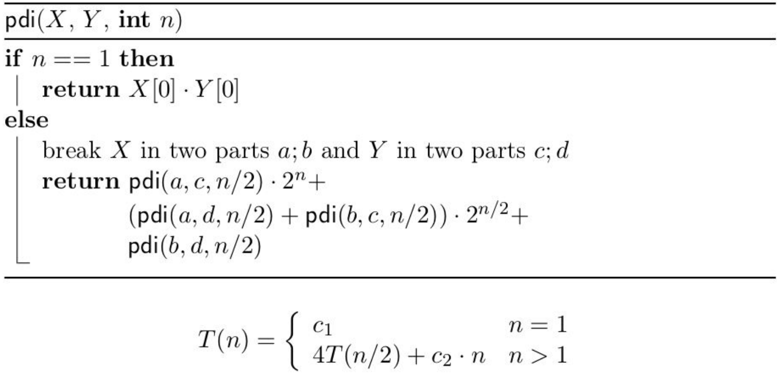
All this, for nothing?

Question: Is it possible to improve this idea?

Note that this recursive version calls itself 4 times

Can we call the function less than 4 times?
Sums and shifts cannot be faster than θ (n).

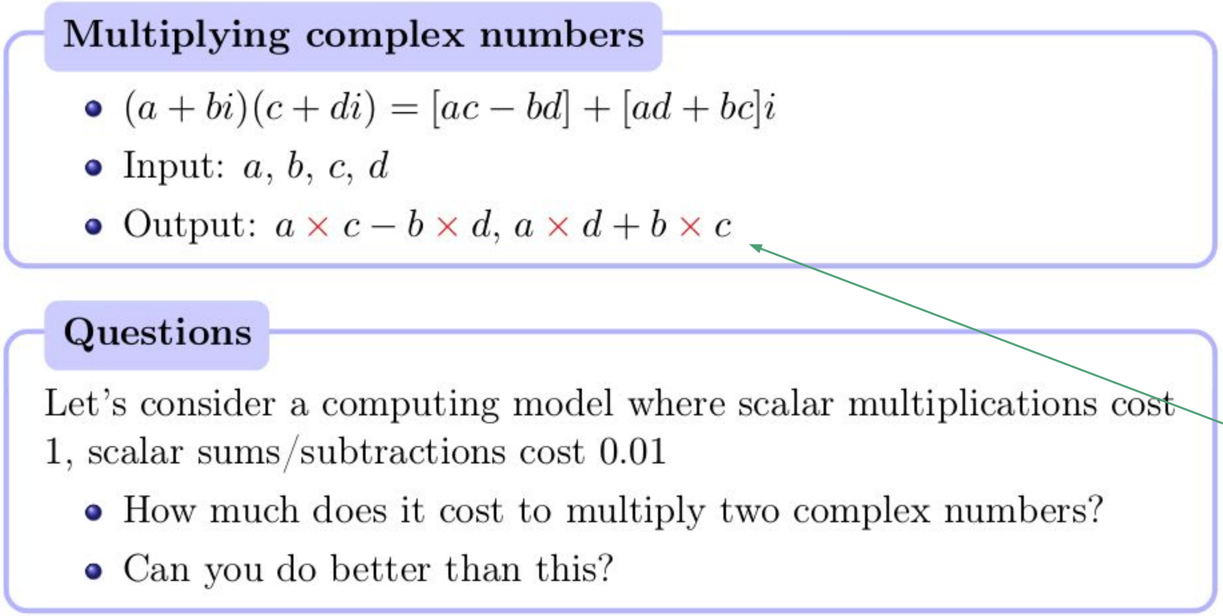
[more on this later… let's step back a sec]



```
pdi($X$, $Y$, int $n$)
if $n == 1$ then
    return $X[0] \cdot Y[0]$
else
    break $X$ in two parts $a; b$ and $Y$ in two parts $c; d$
    return $\mathrm{pdi}(a, c, n/2) \cdot 2^n +$
        $(\mathrm{pdi}(a, d, n/2) + \mathrm{pdi}(b, c, n/2)) \cdot 2^{n/2} +$
        $\mathrm{pdi}(b, d, n/2)$
```

$$T(n) = \begin{cases} c_1 & n = 1 \\ 4T(n/2) + c_2 \cdot n & n > 1 \end{cases}$$

Product of complex numbers (courtesy of Gauss)



Multiplying complex numbers

- $(a+bi)(c+di) = [ac-bd] + [ad+bc]i$
- Input: a, b, c, d
- Output: $a \times c - b \times d$, $a \times d + b \times c$

Questions

Let's consider a computing model where scalar multiplications cost 1, scalar sums/subtractions cost 0.01

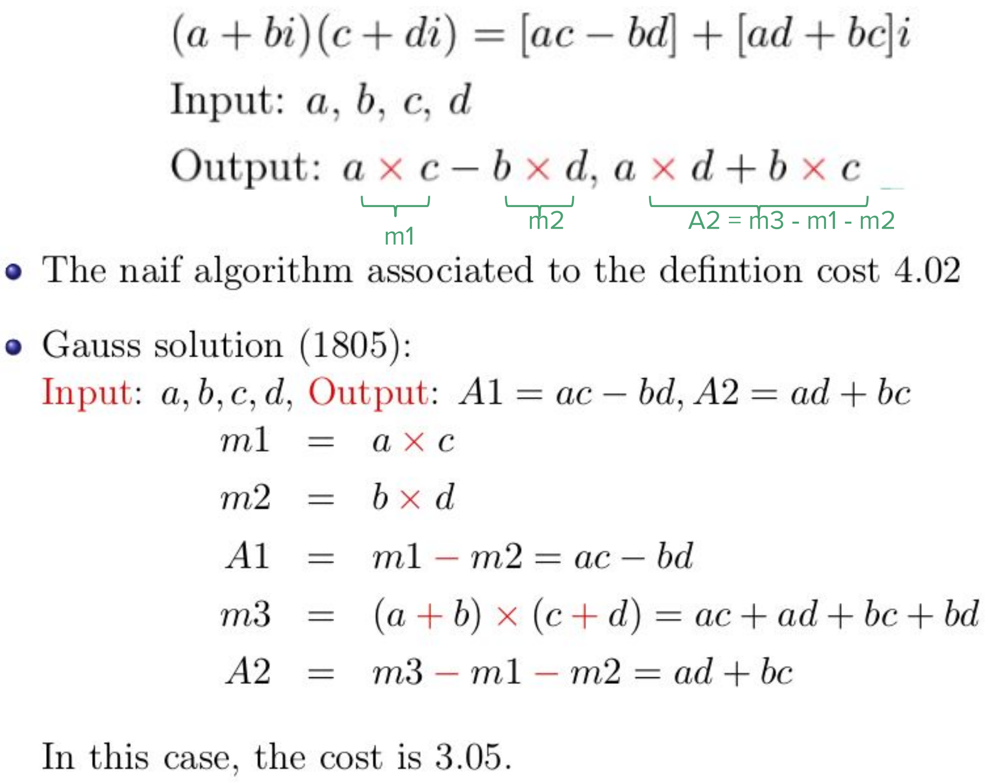
- How much does it cost to multiply two complex numbers?
- Can you do better than this?

a: real part
b * i: imaginary part

i*i = -1

4 multiplications and 2 sums back then very expensive to multiply numbers (cost: 4.02)

Product of complex numbers (courtesy of Gauss)



$$(a+bi)(c+di) = [ac-bd] + [ad+bc]i$$

Input: a, b, c, d

Output: $\underbrace{a \times c}_{m1} - \underbrace{b \times d}_{m2}, \underbrace{a \times d + b \times c}_{A2 = m3 - m1 - m2}$

- The naif algorithm associated to the defintion cost 4.02
- Gauss solution (1805):
  Input: a, b, c, d, Output: $A1 = ac - bd$, $A2 = ad + bc$

$$\begin{aligned} m1 &= a \times c \\ m2 &= b \times d \\ A1 &= m1 - m2 = ac - bd \\ m3 &= (a+b) \times (c+d) = ac + ad + bc + bd \\ A2 &= m3 - m1 - m2 = ad + bc \end{aligned}$$

In this case, the cost is 3.05.

Multiplication: costs 1
Sum/subtraction: cost 0.01

clever part.

This makes it 3 multiplications and 5 sums/subtractions (25% improvement).

Karatsuba Algorithm (1962) (Inspired by Gauss)

$$(a + bi)(c + di) = [ac - bd] + [ad + bc]i$$

Input: $a,\ b,\ c,\ d$

Output: $a \times c - b \times d,\ a \times d + b \times c$

$$A_1 = a \times c$$

$$A_3 = b \times d$$

$$m = (a + b) \times (c + d) = ac + ad + bc + bd$$

$$A_2 = m - A_1 - A_3 = ad + bc$$

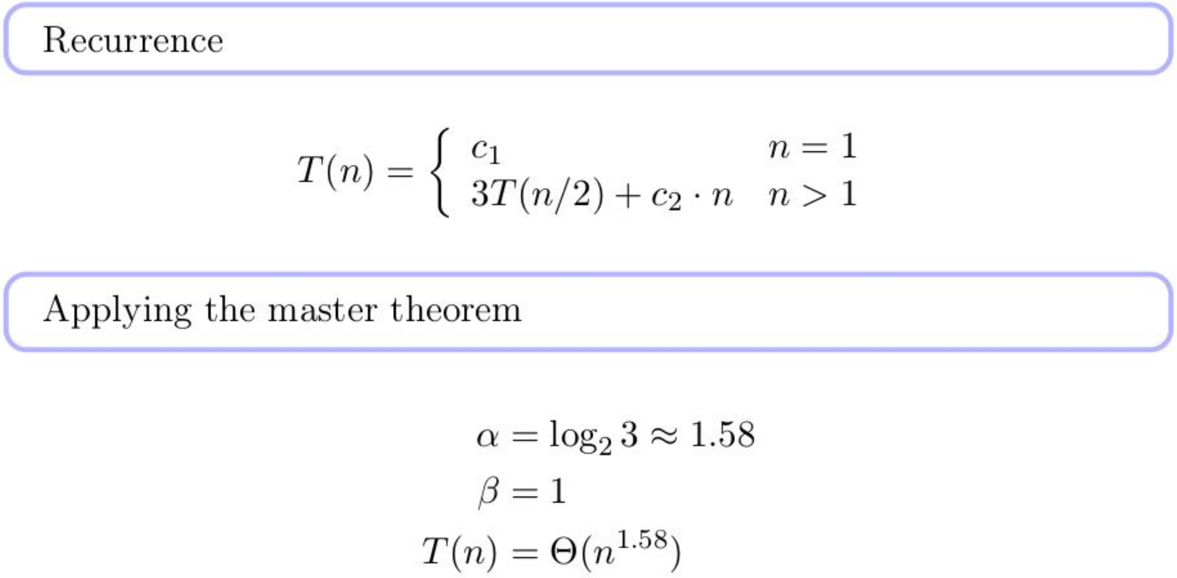
```
boolean [] KARATSUBA(boolean[] X, boolean[] Y, int n)
if $n == 1$ then
    return $X[0] \cdot Y[0]$
else
    break $X$ in $a; b$ e $Y$ in $c; d$
    $A1 =$ KARATSUBA$(a, c, n/2)$
    $A3 =$ KARATSUBA$(b, d, n/2)$
    $m =$ KARATSUBA$(a + b, c + d, n/2)$
    $A2 = m - A1 - A3$
    return $A1 \cdot 2^n + A2 \cdot 2^{n/2} + A3$
```

Three recursive calls that split the numbers in n/2 digits.

Recurrence:

$$T(n) = \begin{cases} c_1 & n = 1 \\ 3T(n/2) + c_2 \cdot n & n > 1 \end{cases}$$

Karatsuba Algorithm (1962) (Inspired by Gauss)



Recurrence

$$T(n) = \begin{cases} c_1 & n = 1 \\ 3T(n/2) + c_2 \cdot n & n > 1 \end{cases}$$

Applying the master theorem

$$\begin{aligned} \alpha &= \log_2 3 \approx 1.58 \\ \beta &= 1 \\ T(n) &= \Theta(n^{1.58}) \end{aligned}$$

Karatsuba Algorithm (1962) (Inspired by Gauss)

Multiplication of real numbers

same reasoning applied to real numbers.

To compute:

$$x = x_1 \cdot 2^{n/2} + x_2$$

$$y = y_1 \cdot 2^{n/2} + y_2$$

$$x \cdot y = x_1 y_1 \cdot 2^n + (x_1 y_2 + x_2 y_1) \cdot 2^{n/2} + x_2 y_2$$

We can calculate:

$$X = x_1 y_1$$

$$Y = x_2 y_2$$

$$Z = (x_1 + x_2) \cdot (y_1 + y_2) - X - Y$$
$$= (x_1 y_1 + x_1 y_2 + x_2 y_1 + x_2 y_2) - x_1 y_1 - x_2 y_2$$

$$x \cdot y = X \cdot 2^n + Z \cdot 2^{n/2} + Y$$

Example (decimal values):

$$1234 \ast 7721 = 12 \ast 10^2 + 34 \cdot 77 \ast 10^2 + 34 = 9527714$$

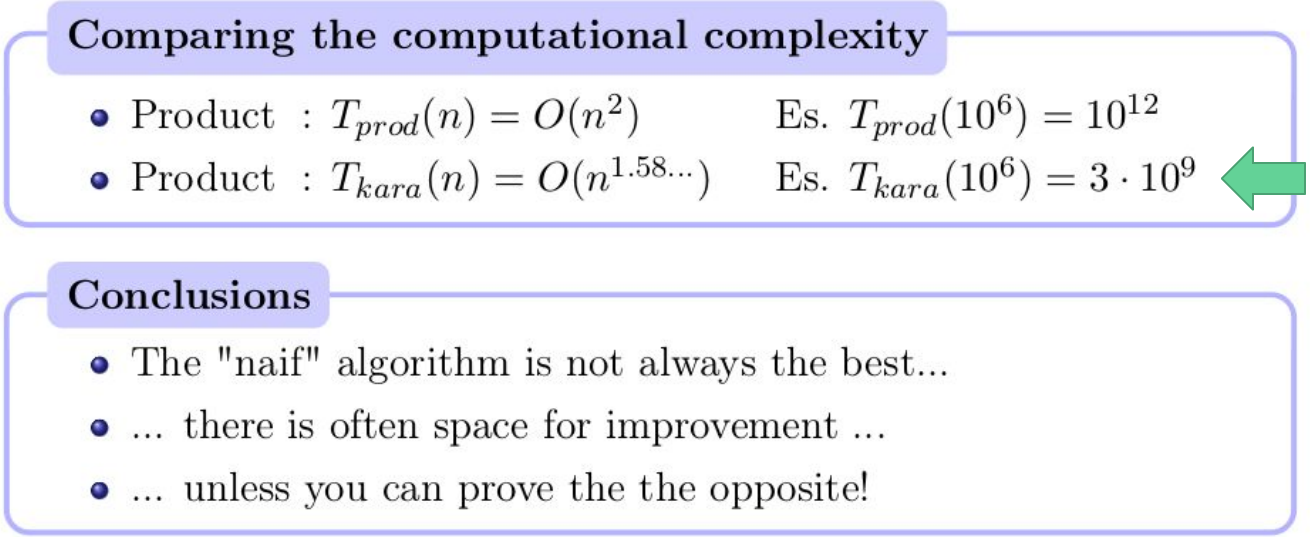
$$X = (12 \ast 77) = 924$$

$$Y = (21 \ast 34) = 714$$

$$Z = (12 + 34) \ast (77 + 21) - X - Y$$
$$= 46 \ast 98 - 924 - 714 = 4508 - 924 - 714 = 2870$$

$$1234 \ast 7721 = 924 \ast 10^4 + 2870 \ast 10^2 + 714 = 9527714$$

Take home message...



Comparing the computational complexity

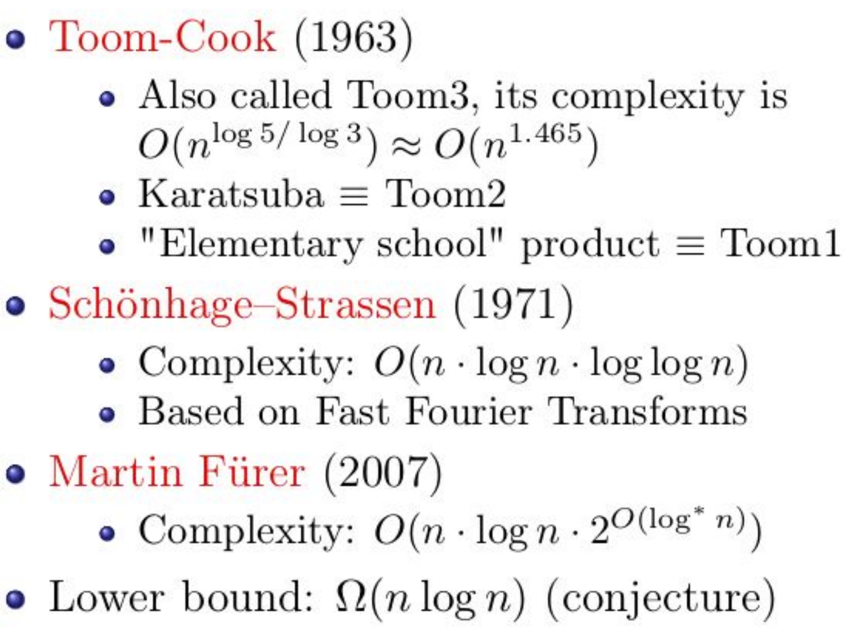
- Product : $T_{prod}(n) = O(n^2)$ Es. $T_{prod}(10^6) = 10^{12}$
- Product : $T_{kara}(n) = O(n^{1.58\ldots})$ Es. $T_{kara}(10^6) = 3 \cdot 10^9$

Conclusions

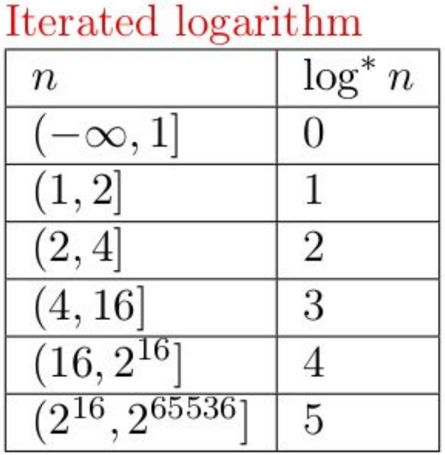
- The "naif" algorithm is not always the best...
- ... there is often space for improvement ...
- ... unless you can prove the the opposite!

1000x
faster

Extensions... Multiplication



- Toom-Cook (1963)
    - Also called Toom3, its complexity is $O(n^{\log 5/\log 3}) \approx O(n^{1.465})$
    - Karatsuba $\equiv$ Toom2
    - "Elementary school" product $\equiv$ Toom1
- Schönhage–Strassen (1971)
    - Complexity: $O(n \cdot \log n \cdot \log \log n)$
    - Based on Fast Fourier Transforms
- Martin Fürer (2007)
    - Complexity: $O(n \cdot \log n \cdot 2^{O(\log^* n)})$
- Lower bound: $\Omega(n \log n)$ (conjecture)

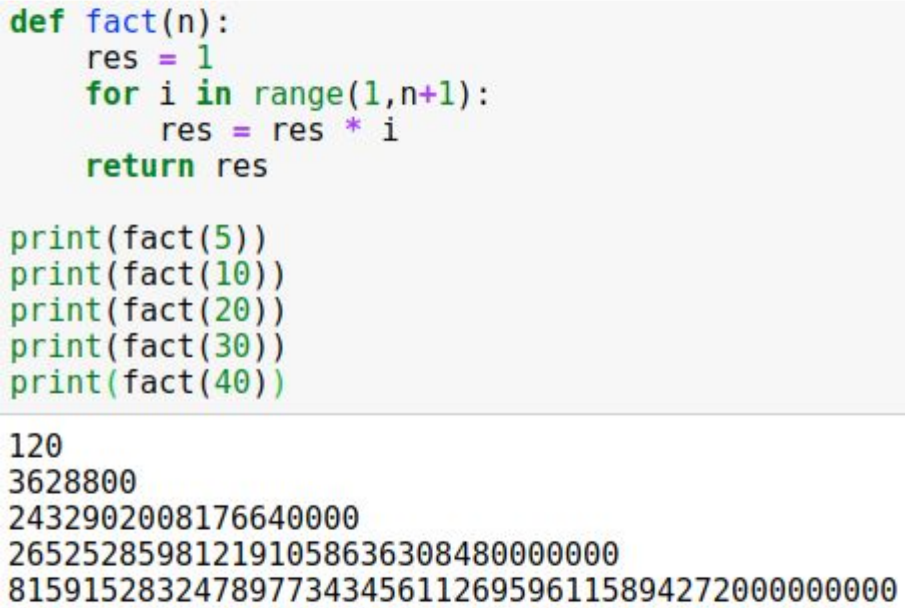


Iterated logarithm

n	$\log^* n$
$(-\infty, 1]$	0
$(1, 2]$	1
$(2, 4]$	2
$(4, 16]$	3
$(16, 2^{16}]$	4
$(2^{16}, 2^{65536}]$	5

still to be proven

Logarithmic cost model



```
def fact(n):
    res = 1
    for i in range(1,n+1):
        res = res * i
    return res

print(fact(5))
print(fact(10))
print(fact(20))
print(fact(30))
print(fact(40))
```

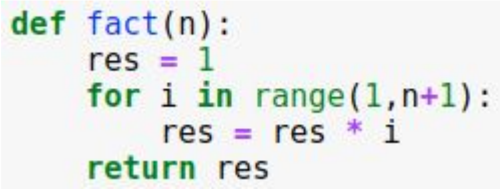
```
120
3628800
2432902008176640000
265252859812191058636308480000000
815915283247897734345611269596115894272000000000
```

This algorithm performs **n** multiplications, so
θ(n)

Is it correct?

Remember that a cost function goes from the **size of the input** to the **time**.

Logarithmic cost model



```python
def fact(n):
    res = 1
    for i in range(1,n+1):
        res = res * i
    return res
```

This algorithm performs **n** multiplications, so
θ(n)

Is it correct?

Remember that a cost function goes from the **size of the input** to the **time**.

n IS the input!

What is the size of the input?

k = ⌊log n⌋

How many multiplications, in terms of k?

n = 2^k

How many bits are necessary, to represent the output?

⌊log n!⌋ = θ(n log n)
= 2^k * k

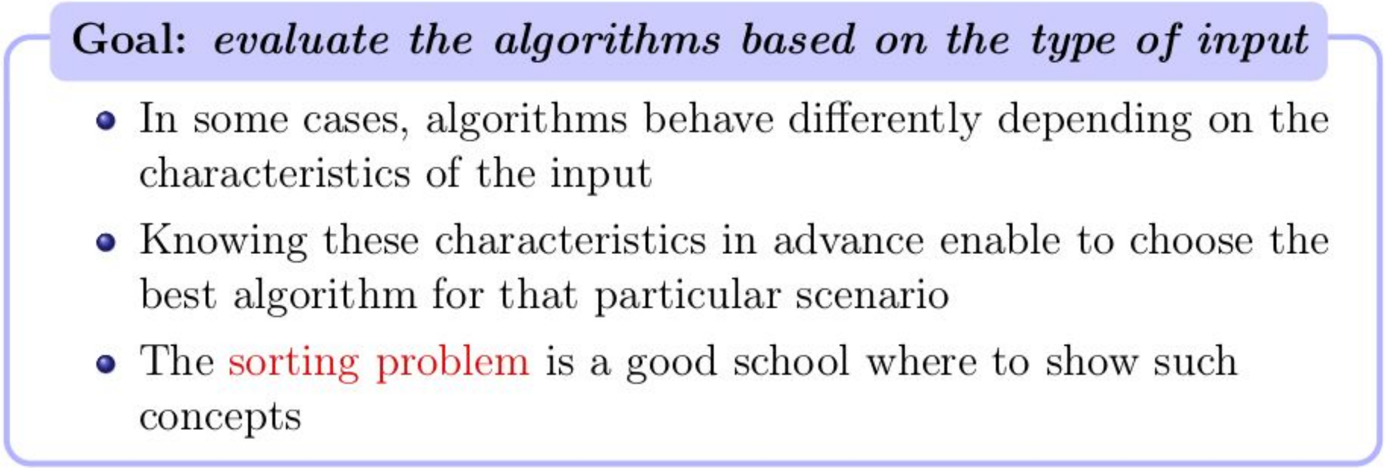
How much does it cost to multiply two numbers of 2^k · k bits?

$\Theta(2^{2k} \cdot k^2)$

What is the complexity of the factorial (n=2^k multiplications)?

$\Theta(2^{3k} \cdot k^2)$ = $\Theta(n^3 (\log n)^2)$

Sorting algorithms

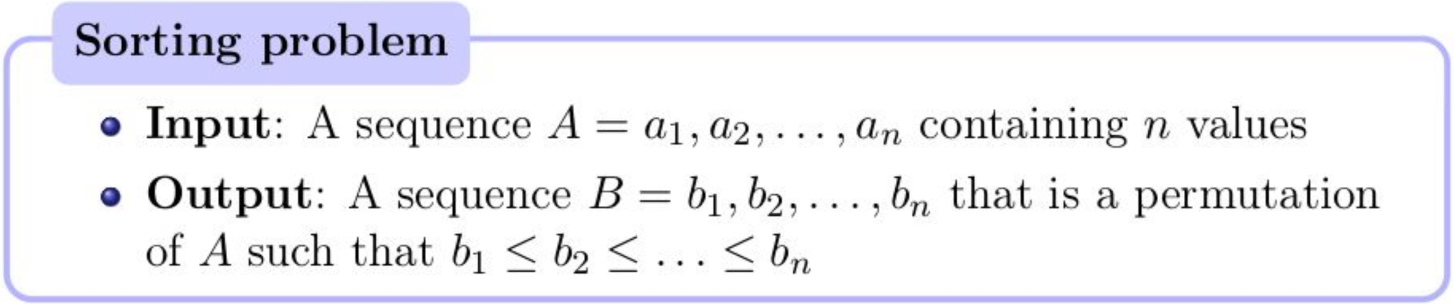


Goal: ***evaluate the algorithms based on the type of input***

- In some cases, algorithms behave differently depending on the characteristics of the input
- Knowing these characteristics in advance enable to choose the best algorithm for that particular scenario
- The sorting problem is a good school where to show such concepts

sorting algorithms are already implemented (in general, no need to reinvent the wheel) , but they are a great training ground.

Sorting



Sorting problem

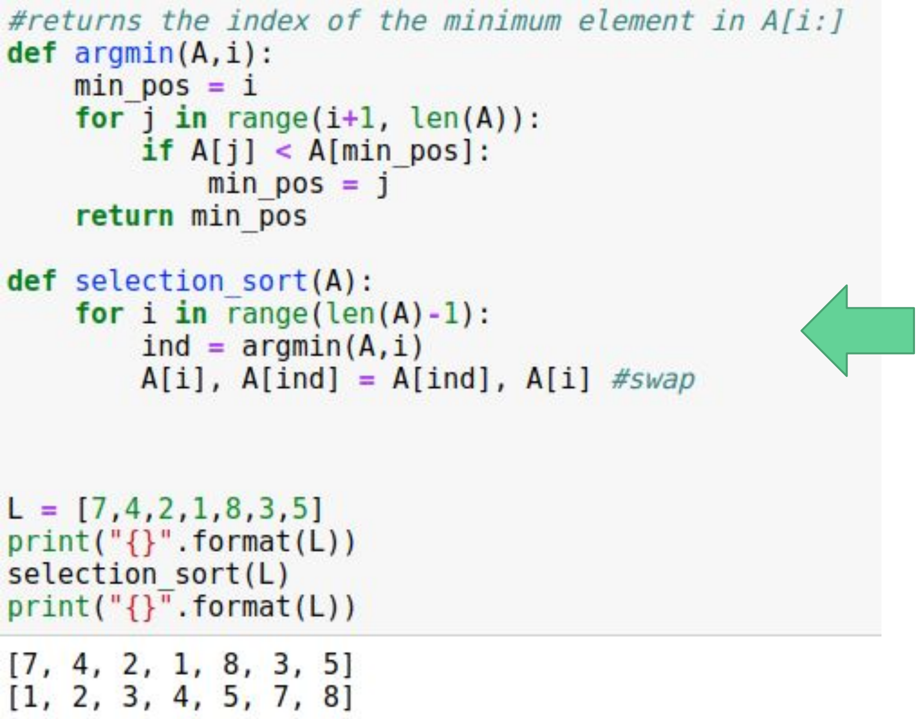
- **Input**: A sequence $A = a_1, a_2, \ldots, a_n$ containing n values
- **Output**: A sequence $B = b_1, b_2, \ldots, b_n$ that is a permutation of A such that $b_1 \leq b_2 \leq \ldots \leq b_n$

Naive approach:

- Search for the minimum, put it in the correct position, reduce the problem to the n – 1 elements that are left and continue until the sequence is finished
- This is called **selection sort**

Selection Sort

Search for the minimum, put it in the correct position, reduce the problem to the n − 1 elements that are left and continue until the sequence is finished



```python
#returns the index of the minimum element in A[i:]
def argmin(A,i):
    min_pos = i
    for j in range(i+1, len(A)):
        if A[j] < A[min_pos]:
            min_pos = j
    return min_pos

def selection_sort(A):
    for i in range(len(A)-1):
        ind = argmin(A,i)
        A[i], A[ind] = A[ind], A[i] #swap



L = [7,4,2,1,8,3,5]
print("{}".format(L))
selection_sort(L)
print("{}".format(L))
```

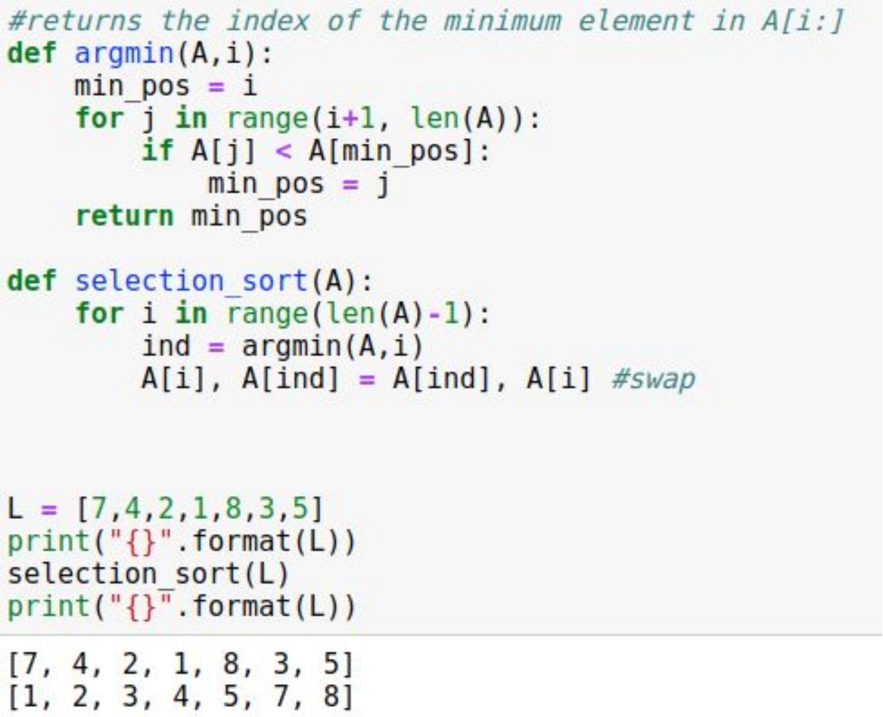
```
[7, 4, 2, 1, 8, 3, 5]
[1, 2, 3, 4, 5, 7, 8]
```

argmin(A,i) returns the index of the minimum element in A[i:]

This function repeatedly searches the **minimum in A[i :]**, and **swaps** it with the element in **A[i]**
i: 0,..,n-1 since the last value is already in the right position

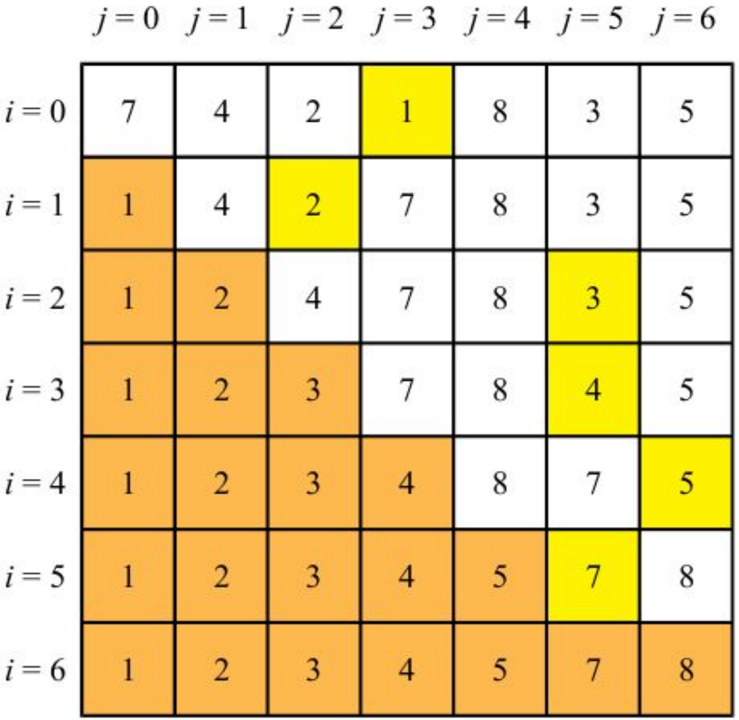
Selection Sort

Search for the minimum, put it in the correct position, reduce the problem to the n − 1 elements that are left and continue until the sequence is finished



```
#returns the index of the minimum element in A[i:]
def argmin(A,i):
    min_pos = i
    for j in range(i+1, len(A)):
        if A[j] < A[min_pos]:
            min_pos = j
    return min_pos

def selection_sort(A):
    for i in range(len(A)-1):
        ind = argmin(A,i)
        A[i], A[ind] = A[ind], A[i] #swap



L = [7,4,2,1,8,3,5]
print("{}".format(L))
selection_sort(L)
print("{}".format(L))
```

```
[7, 4, 2, 1, 8, 3, 5]
[1, 2, 3, 4, 5, 7, 8]
```

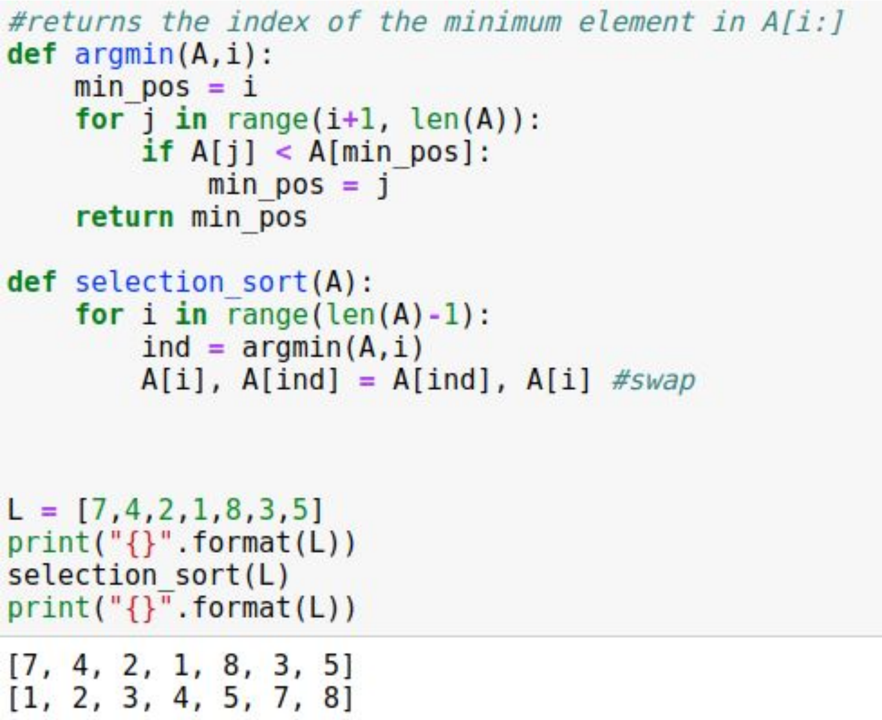


	$j=0$	$j=1$	$j=2$	$j=3$	$j=4$	$j=5$	$j=6$
$i=0$	7	4	2	1	8	3	5
$i=1$	1	4	2	7	8	3	5
$i=2$	1	2	4	7	8	3	5
$i=3$	1	2	3	7	8	4	5
$i=4$	1	2	3	4	8	7	5
$i=5$	1	2	3	4	5	7	8
$i=6$	1	2	3	4	5	7	8

How much does this cost?

Selection Sort

Search for the minimum, put it in the correct position, reduce the problem to the n − 1 elements that are left and continue until the sequence is finished



```
#returns the index of the minimum element in A[i:]
def argmin(A,i):
    min_pos = i
    for j in range(i+1, len(A)):
        if A[j] < A[min_pos]:
            min_pos = j
    return min_pos

def selection_sort(A):
    for i in range(len(A)-1):
        ind = argmin(A,i)
        A[i], A[ind] = A[ind], A[i] #swap



L = [7,4,2,1,8,3,5]
print("{}".format(L))
selection_sort(L)
print("{}".format(L))
```

```
[7, 4, 2, 1, 8, 3, 5]
[1, 2, 3, 4, 5, 7, 8]
```

How much does this cost?

How many comparisons in argmin(A, i)?

len(A) − 1 − i = n − 1 − i

How many comparisons in selection_sort(A)?

$$= \sum_{i=0}^{n-2}(n-1-i)$$

$$= (n-1) + (n-2) + \ldots + 2 + 1$$

$$= \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} = n^2 - n/2$$

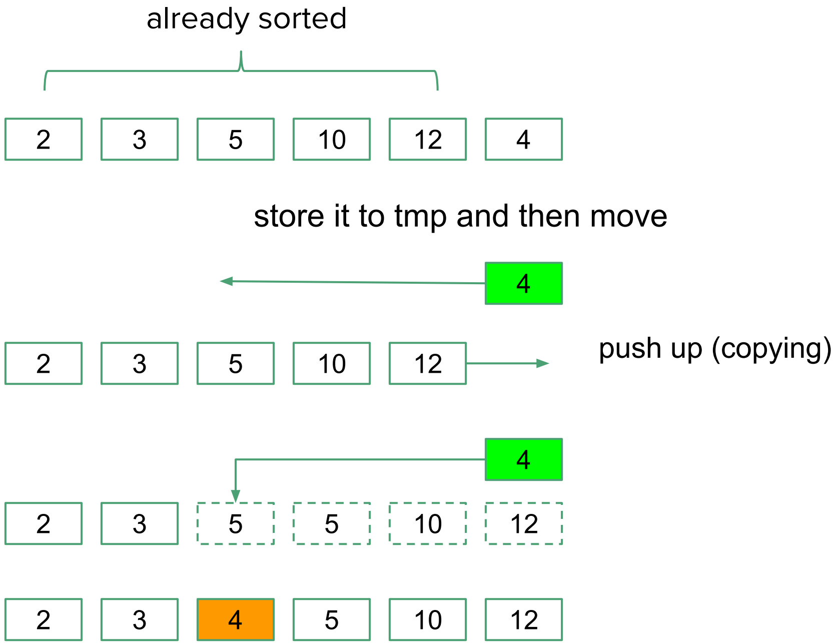
Complexity is **θ(n^2)** in worst, average, best case (the algorithm works in the same way regardless of the input).

Insertion Sort

The idea of **insertion sort** is to build a sorted list **step by step**. In each step, one element is placed in its correct position on the left-side part of the array.

At each iteration (i):

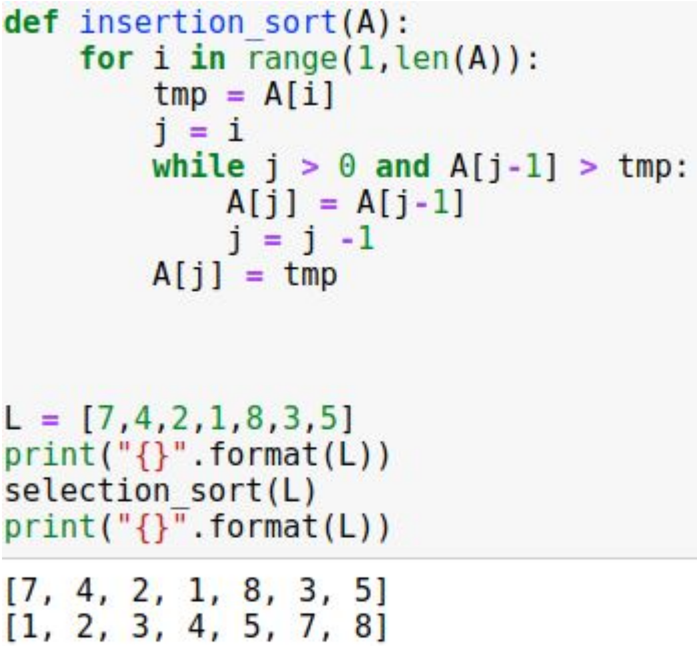
```
A[i] → temp
for j: i-1,...,0
        if A[j] > A[i]
        copy A[j] →A[j+1]
A[j] ← A[i].
```



Efficient algorithm to sort small sets of elements ~100s (small constants)
It is “in-place” there is no need to copy the list (saves memory!)

Insertion Sort

The idea of **insertion sort** is to build a sorted list **step by step**. In each step, one element is placed in its correct position on the left-side part of the array.



```
def insertion_sort(A):
    for i in range(1,len(A)):
        tmp = A[i]
        j = i
        while j > 0 and A[j-1] > tmp:
            A[j] = A[j-1]
            j = j -1
        A[j] = tmp



L = [7,4,2,1,8,3,5]
print("{}".format(L))
selection_sort(L)
print("{}".format(L))
```

```
[7, 4, 2, 1, 8, 3, 5]
[1, 2, 3, 4, 5, 7, 8]
```

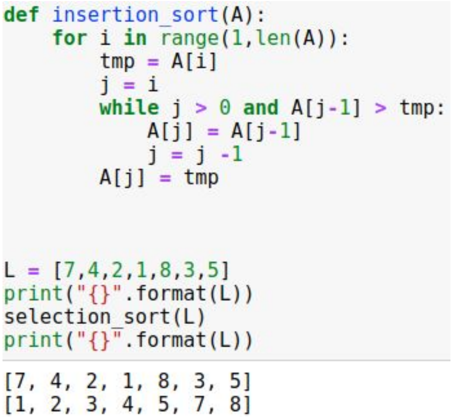
The first element is assumed to be a sorted list (with one element).

The first current element is the second in the list. Range starts from 1!

The current element is placed in a TMP variable, and the values before in the list are copied up until they are lower than TMP.

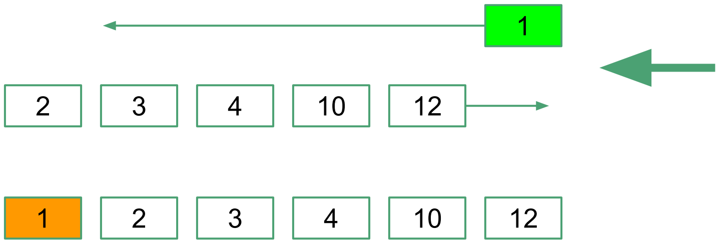
When I find a value that is lower than TMP, I place the value there

Insertion Sort



```
def insertion_sort(A):
    for i in range(1,len(A)):
        tmp = A[i]
        j = i
        while j > 0 and A[j-1] > tmp:
            A[j] = A[j-1]
            j = j -1
        A[j] = tmp


L = [7,4,2,1,8,3,5]
print("{}".format(L))
selection_sort(L)
print("{}".format(L))
```

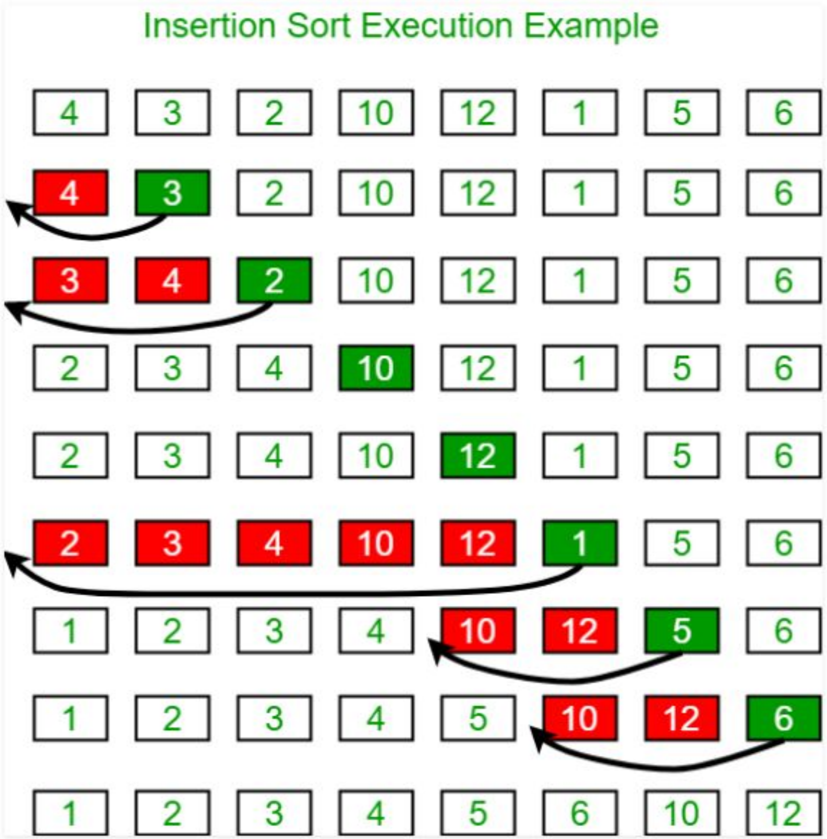
```
[7, 4, 2, 1, 8, 3, 5]
[1, 2, 3, 4, 5, 7, 8]
```



1

2 3 4 10 12

1 2 3 4 10 12



Insertion Sort Execution Example

4 3 2 10 12 1 5 6

4 3 2 10 12 1 5 6

3 4 2 10 12 1 5 6

2 3 4 10 12 1 5 6

2 3 4 10 12 1 5 6

2 3 4 10 12 1 5 6

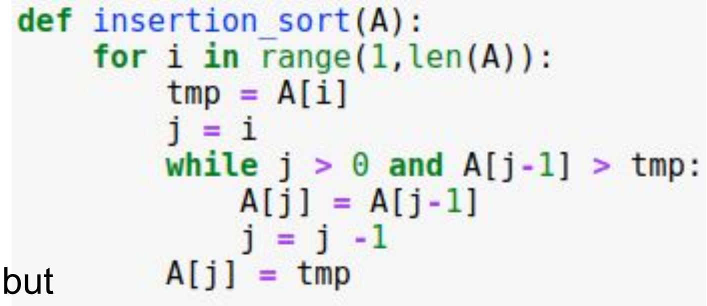
1 2 3 4 10 12 5 6

1 2 3 4 5 10 12 6

1 2 3 4 5 6 10 12

[https://www.geeksforgeeks.org]

Insertion Sort: complexity



```
def insertion_sort(A):
    for i in range(1,len(A)):
        tmp = A[i]
        j = i
        while j > 0 and A[j-1] > tmp:
            A[j] = A[j-1]
            j = j -1
        A[j] = tmp
```

The cost does not depend only on the **size of the input** but also on <u>how the values are sorted</u>

What is the cost if the list is already sorted?

- the for is executed (n operations), never gets into the while: **θ(n)**

What is the cost if the list is sorted in reverse order?

- the for is executed (n operations), for each, all elements have to be pushed up (n operations): **θ(n^2)**

What is the cost on average? (informally, half list sorted)

- the for is executed (n operations), for each, half of the elements have to be pushed up (n operations): **θ(n^2)**

Merge Sort

IDEA: Sorting two sublists already sorted is fast!

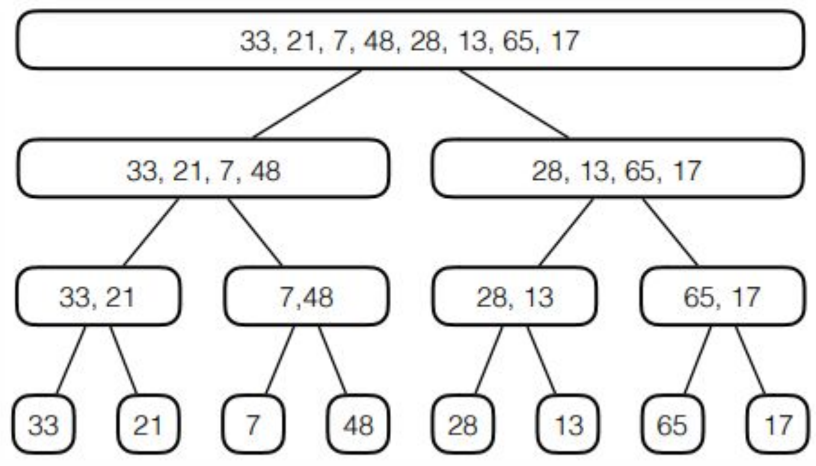
MergeSort is based on the divide-et-impera technique

Divide: Break (virtually) the sequence of n elements in two sub-sequences

Impera: Call MergeSort recursively on both sub-sequences (note that sub-lists of one element are sorted lists!)

Combine: Join (merge) the two sorted sub-sequences

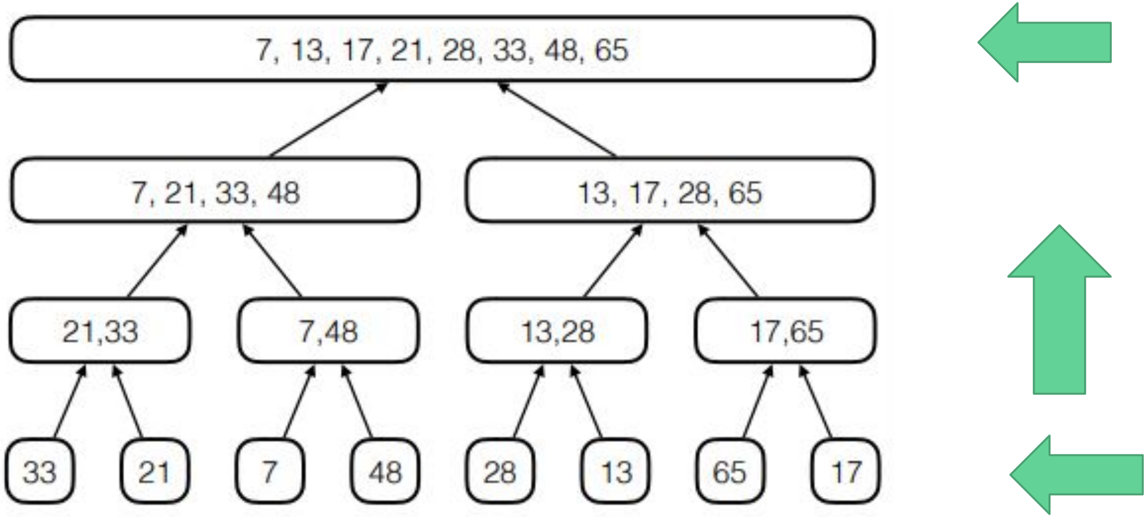
Merge Sort



Keep dividing

1 sized lists are **ordered**!

Merge Sort



7, 13, 17, 21, 28, 33, 48, 65

7, 21, 33, 48

13, 17, 28, 65

21,33

7,48

13,28

17,65

33

21

7

48

28

13

65

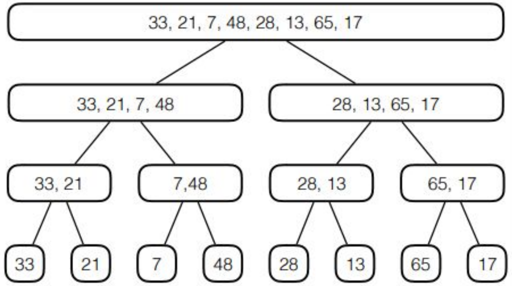
17

1 List only: solution

merge sorted lists
into bigger sorted
lists

1 element only: sorted!

Merge Sort



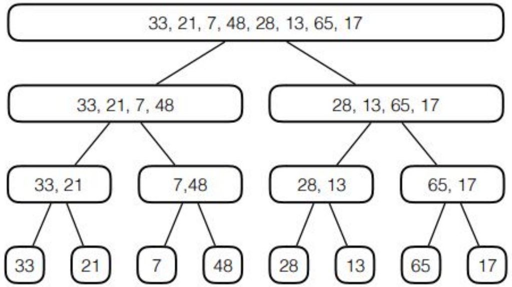
Merge sort requires three methods:

1. `merge`: **gets two sorted lists and produces a sorted list** with all the elements. Builds the return list by getting the minimum element of the two lists, "removing" it from the corresponding list and appending it to the list with the result. "removal" can be done by using two indexes pointing to the smallest elements of each of the two (sub)lists and incrementing the index of the minimum of the two (i.e. the element that is also copied to the result list);

2. `recursiveMergeSort`: gets **an unordered (sub)list**, the **index of the beginning** of the list and the index of the end of the list and recursively **splits it in two halves until it reaches lists with length 0 or 1**, at that point **it starts merging pairs of sorted lists to build the result (with `merge`)**;

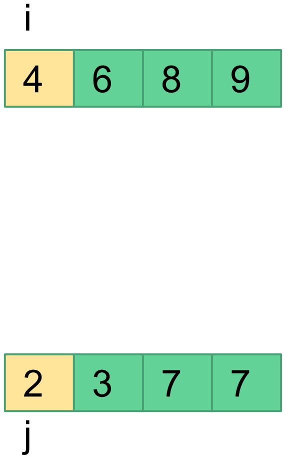
3. `mergeSort` gets an **unordered list and applies the** `recursiveMergeSort` method to it starting **from position 0 to *len*−1**.

Merge sort: implementation

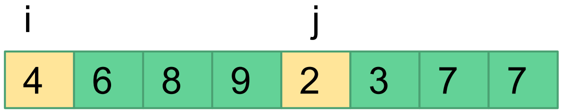


The Merge method:

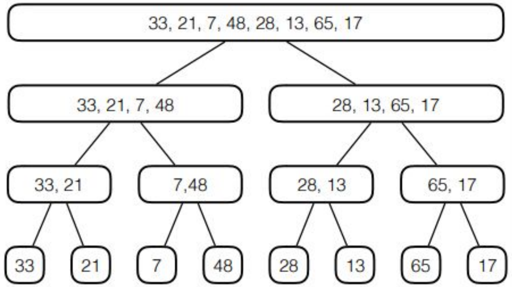
gets two sorted lists and produces a sorted list with all the elements. Builds the return list by getting the minimum element of the two lists, "removing" it from the corresponding list and appending it to the list with the result (using two indexes pointing to the minimum of each list).



note: the two lists can be sublists of one list:

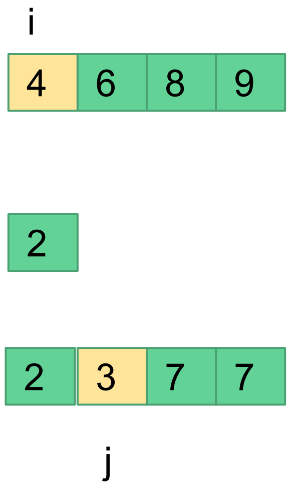


Merge sort: implementation



The Merge method:

gets two sorted lists and produces a sorted list with all the elements. Builds the return list by getting the minimum element of the two lists, "removing" it from the corresponding list and appending it to the list with the result (using two indexes pointing to the minimum of each list).



i

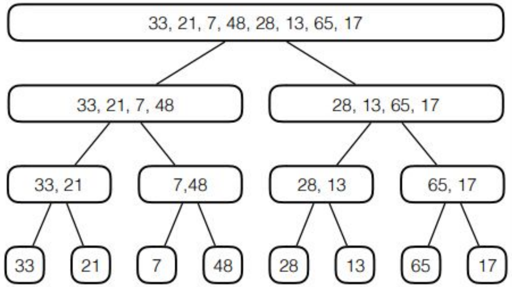
4	6	8	9

2

2	3	7	7

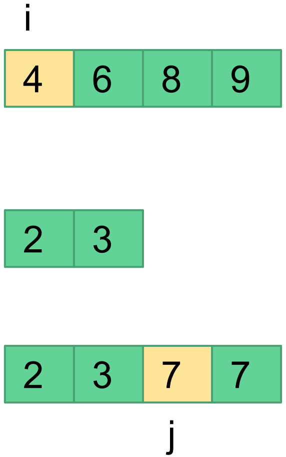
j

Merge sort: implementation

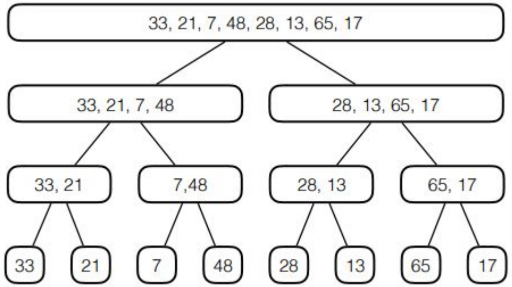


The Merge method:

gets two sorted lists and produces a sorted list with all the elements. Builds the return list by getting the minimum element of the two lists, “removing” it from the corresponding list and appending it to the list with the result (using two indexes pointing to the minimum of each list).

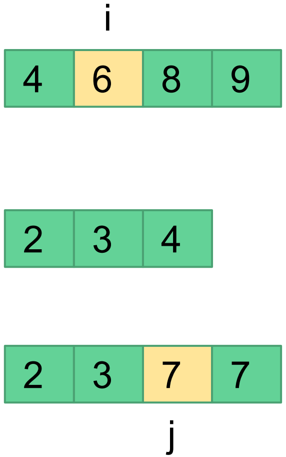


Merge sort: implementation

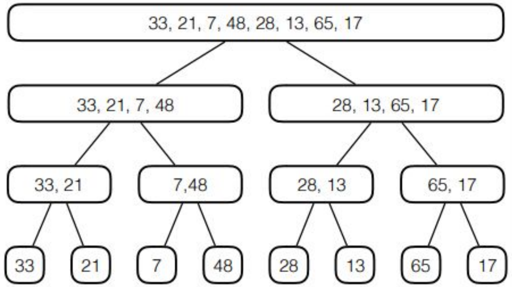


The Merge method:

gets two sorted lists and produces a sorted list with all the elements. Builds the return list by getting the minimum element of the two lists, “removing” it from the corresponding list and appending it to the list with the result (using two indexes pointing to the minimum of each list).

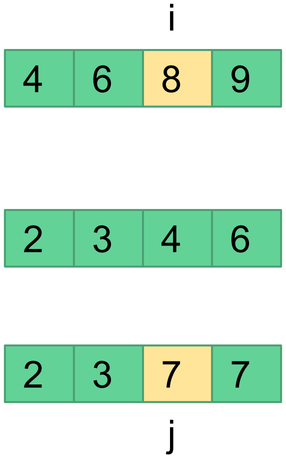


Merge sort: implementation



The Merge method:

gets two sorted lists and produces a sorted list with all the elements. Builds the return list by getting the minimum element of the two lists, "removing" it from the corresponding list and appending it to the list with the result (using two indexes pointing to the minimum of each list).

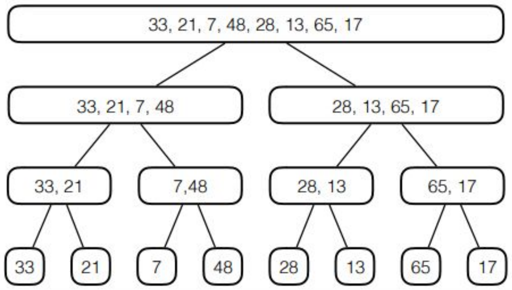


4	6	8	9

2	3	4	6

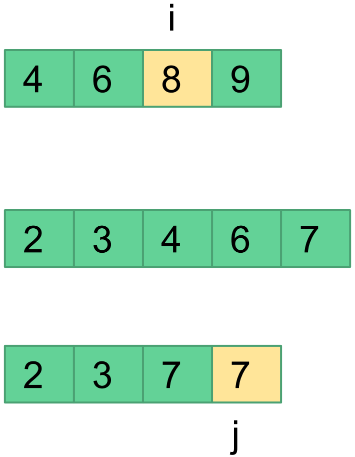
2	3	7	7

Merge sort: implementation



The Merge method:

gets two sorted lists and produces a sorted list with all the elements. Builds the return list by getting the minimum element of the two lists, “removing” it from the corresponding list and appending it to the list with the result (using two indexes pointing to the minimum of each list).



i

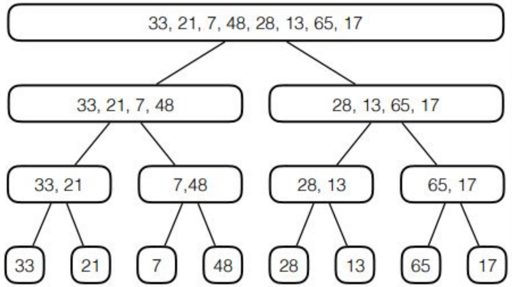
4	6	8	9

2	3	4	6	7

2	3	7	7

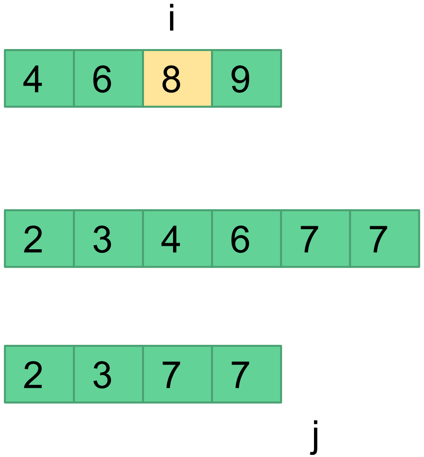
j

Merge sort: implementation

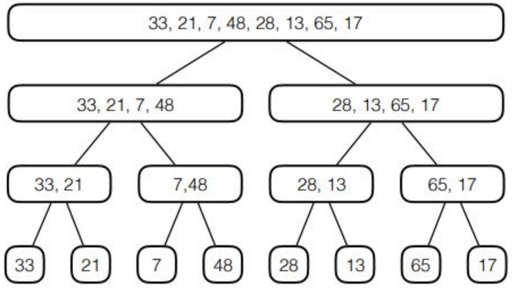


The Merge method:

gets two sorted lists and produces a sorted list with all the elements. Builds the return list by getting the minimum element of the two lists, “removing” it from the corresponding list and appending it to the list with the result (using two indexes pointing to the minimum of each list).



Merge sort: implementation



The Merge method:

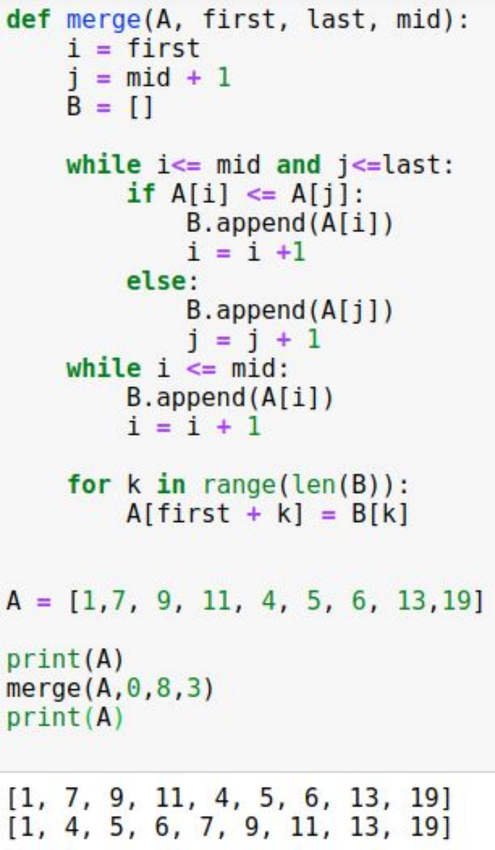
gets two sorted lists and produces a sorted list with all the elements. Builds the return list by getting the minimum element of the two lists, “removing” it from the corresponding list and appending it to the list with the result (using two indexes pointing to the minimum of each list).

i

2	3	4	6	7	7	8	9

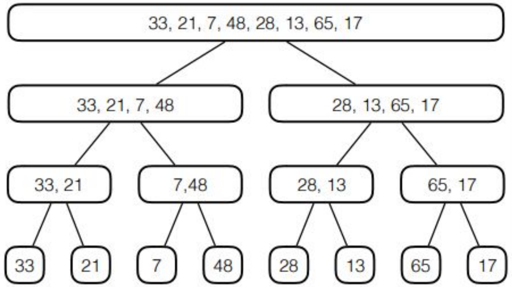
j

Merge sort: merge



```
def merge(A, first, last, mid):
    i = first
    j = mid + 1
    B = []

    while i<= mid and j<=last:
        if A[i] <= A[j]:
            B.append(A[i])
            i = i +1
        else:
            B.append(A[j])
            j = j + 1
    while i <= mid:
        B.append(A[i])
        i = i + 1

    for k in range(len(B)):
        A[first + k] = B[k]


A = [1,7, 9, 11, 4, 5, 6, 13,19]

print(A)
merge(A,0,8,3)
print(A)
```

```
[1, 7, 9, 11, 4, 5, 6, 13, 19]
[1, 4, 5, 6, 7, 9, 11, 13, 19]
```



Variables **i** and **j** are used to scan the values of the two sublists (mid is the mid-point between the two sorted sublists).

The first **while** loop compares the elements when both A[i:mid+1] and A[j:last+1] are not empty, add the smaller to B, and increase either i or j

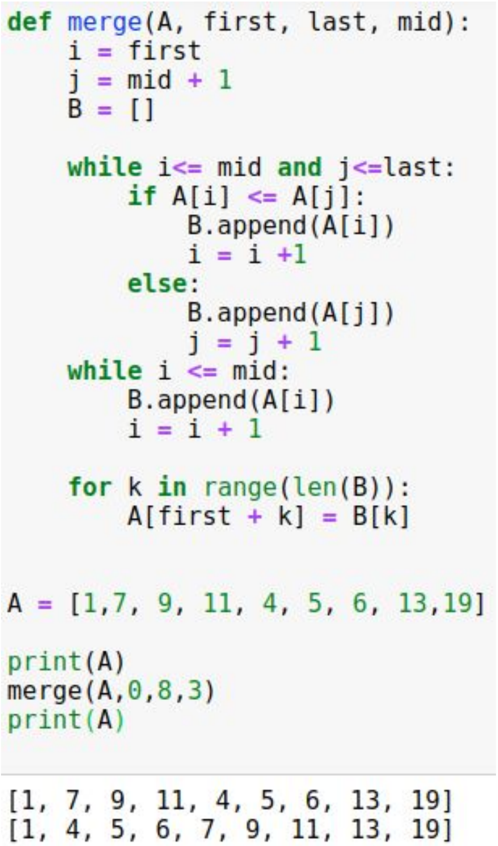
The second while loop moves the remaining elements in **A[i:mid+1]** to **B**

The elements in **B** are smaller than those in **A[j:last+1]** They are moved back into **A[first:first+len(B)]**

Merge sort. Cost of merge

What is the computational cost of Merge ()?

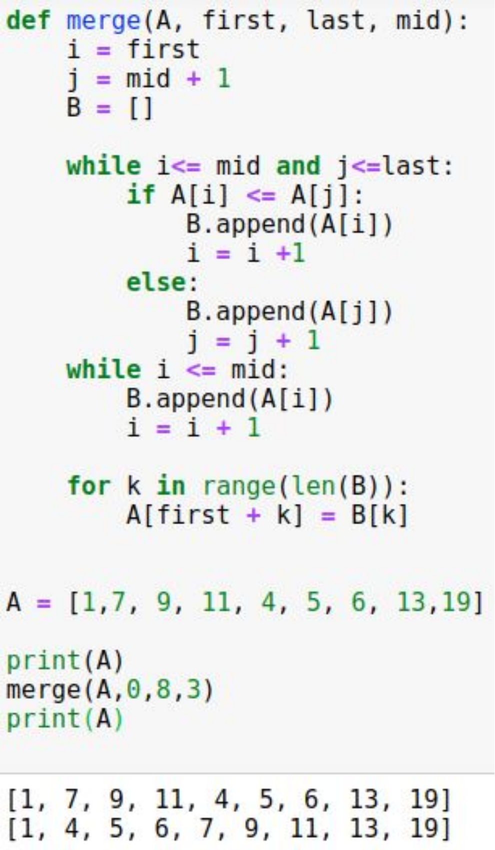
Every time we place one element (and we perform one comparison), so in total n comparisons ⇒ **O(n)**



```python
def merge(A, first, last, mid):
    i = first
    j = mid + 1
    B = []

    while i<= mid and j<=last:
        if A[i] <= A[j]:
            B.append(A[i])
            i = i +1
        else:
            B.append(A[j])
            j = j + 1
    while i <= mid:
        B.append(A[i])
        i = i + 1

    for k in range(len(B)):
        A[first + k] = B[k]


A = [1,7, 9, 11, 4, 5, 6, 13,19]

print(A)
merge(A,0,8,3)
print(A)
```

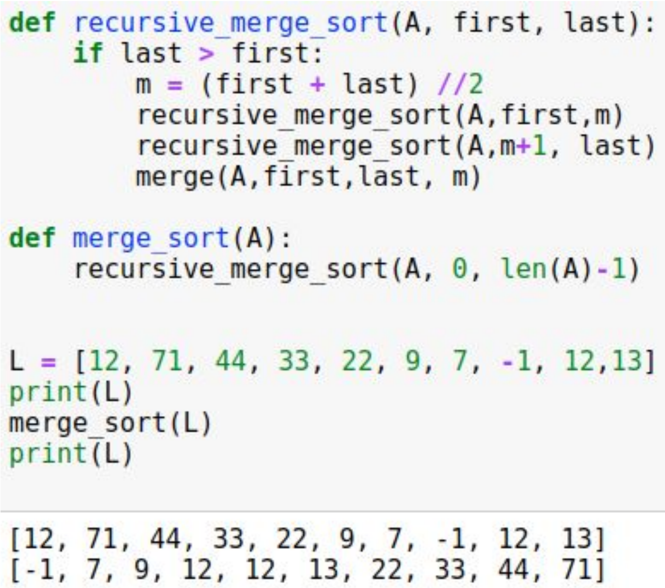
```
[1, 7, 9, 11, 4, 5, 6, 13, 19]
[1, 4, 5, 6, 7, 9, 11, 13, 19]
```

Merge sort. Complete code



```
def merge(A, first, last, mid):
    i = first
    j = mid + 1
    B = []

    while i<= mid and j<=last:
        if A[i] <= A[j]:
            B.append(A[i])
            i = i +1
        else:
            B.append(A[j])
            j = j + 1
    while i <= mid:
        B.append(A[i])
        i = i + 1

    for k in range(len(B)):
        A[first + k] = B[k]


A = [1,7, 9, 11, 4, 5, 6, 13,19]

print(A)
merge(A,0,8,3)
print(A)
```

```
[1, 7, 9, 11, 4, 5, 6, 13, 19]
[1, 4, 5, 6, 7, 9, 11, 13, 19]
```



```
def recursive_merge_sort(A, first, last):
    if last > first:
        m = (first + last) //2
        recursive_merge_sort(A,first,m)
        recursive_merge_sort(A,m+1, last)
        merge(A,first,last, m)

def merge_sort(A):
    recursive_merge_sort(A, 0, len(A)-1)


L = [12, 71, 44, 33, 22, 9, 7, -1, 12,13]
print(L)
merge_sort(L)
print(L)
```

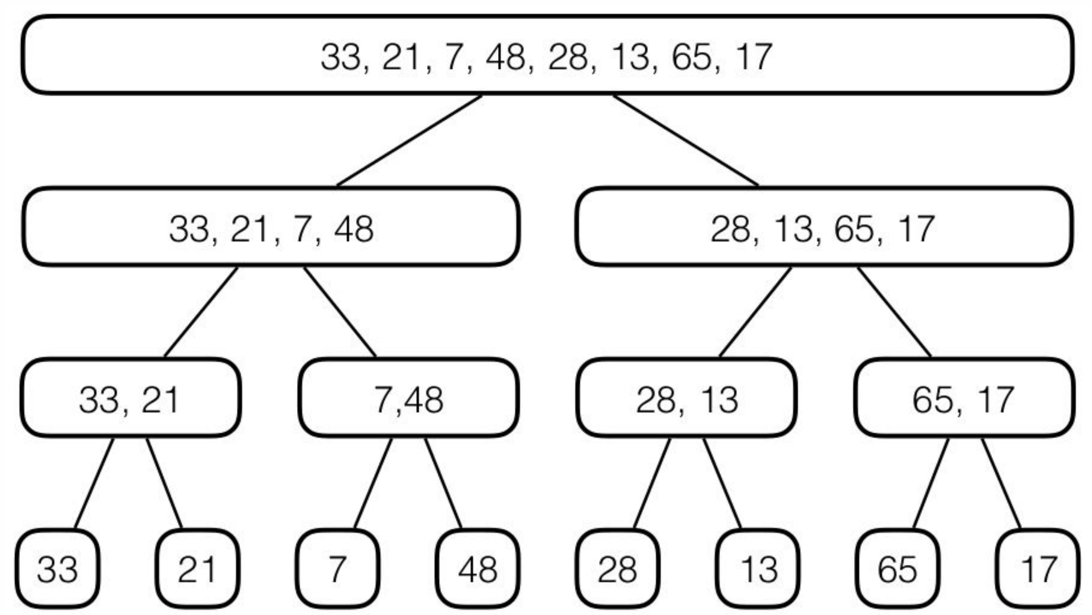
```
[12, 71, 44, 33, 22, 9, 7, -1, 12, 13]
[-1, 7, 9, 12, 12, 13, 22, 33, 44, 71]
```

If we have some elements (last > first):

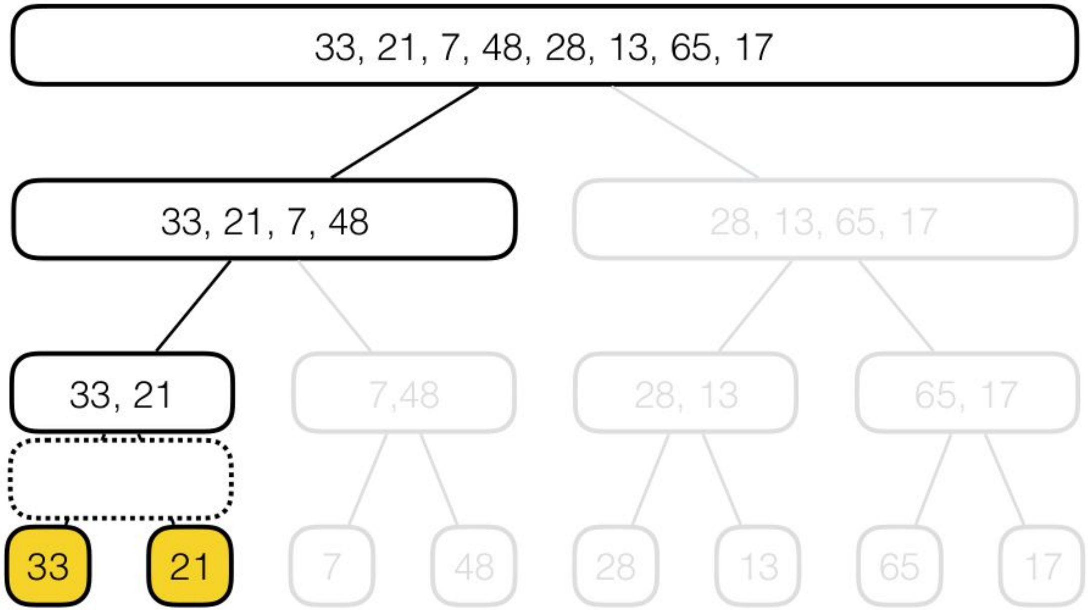
- sort from first to m
- sort from m+1 to last
- merge the two sublists

Merge sort.

call
merge_sort



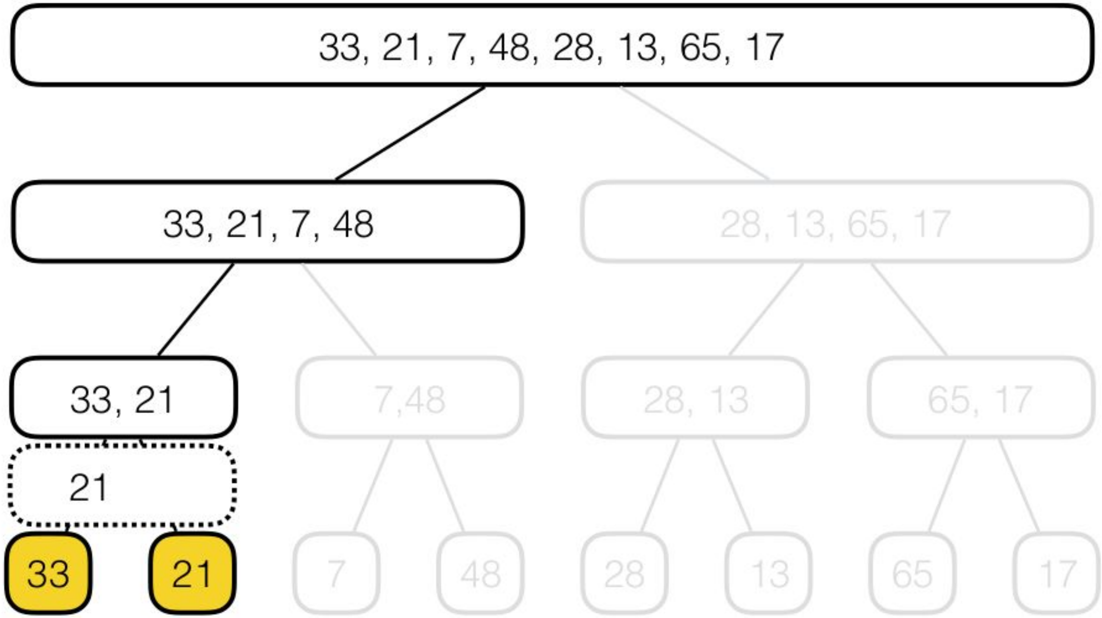
Merge sort.



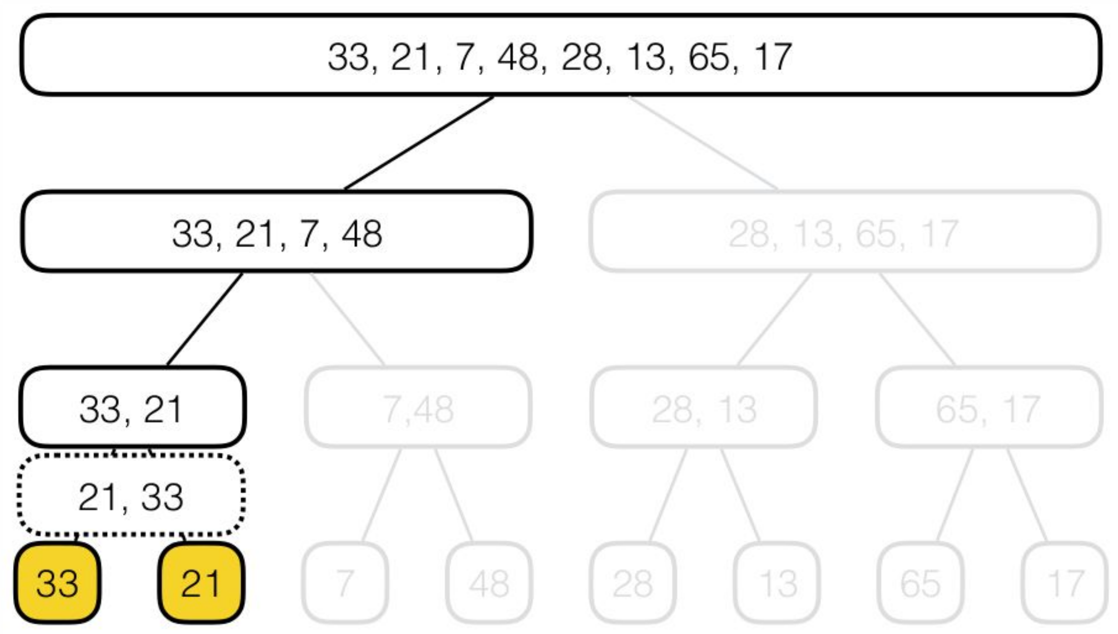
call
merge_sort

call merge_sort
on 33 (sorted)
call merge_sort
on 21 (sorted)
merge the two!

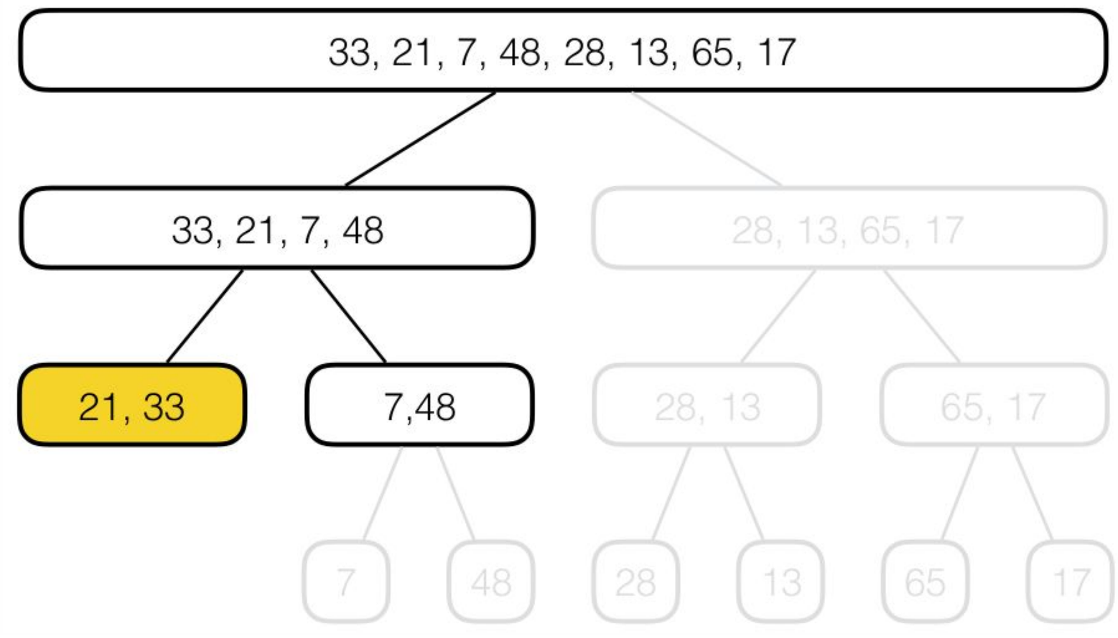
Merge sort.


33, 21, 7, 48, 28, 13, 65, 17
33, 21, 7, 48
28, 13, 65, 17
33, 21
7,48
28, 13
65, 17
21
33
21
7
48
28
13
65
17

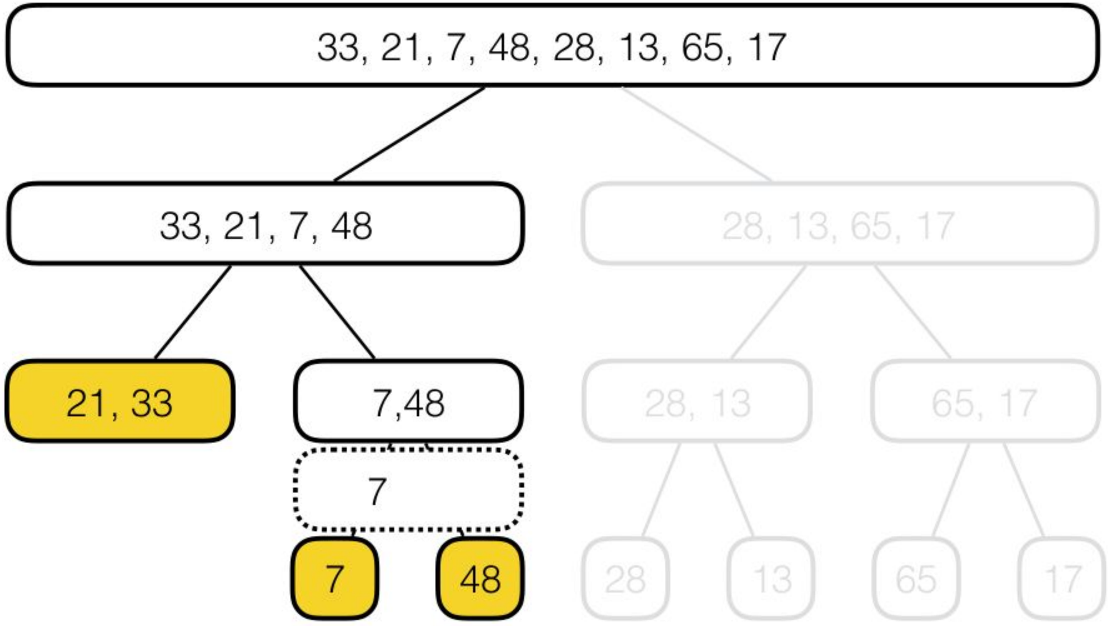
Merge sort.


33, 21, 7, 48, 28, 13, 65, 17
33, 21, 7, 48
28, 13, 65, 17
33, 21
7,48
28, 13
65, 17
21, 33
33
21
7
48
28
13
65
17

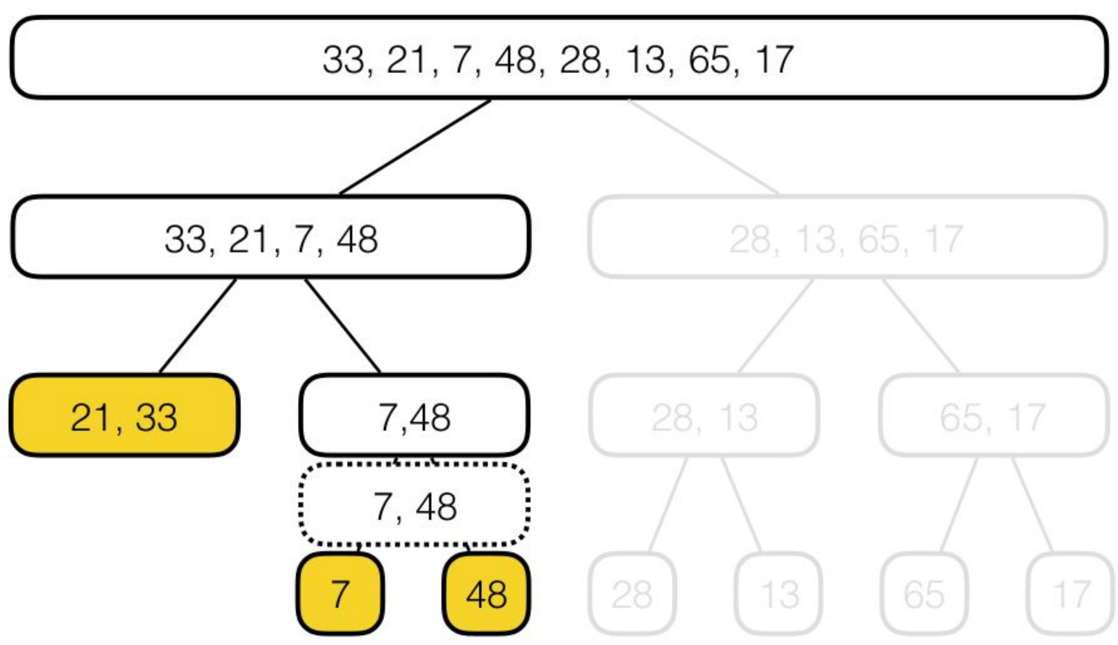
Merge sort.


33, 21, 7, 48, 28, 13, 65, 17
33, 21, 7, 48
28, 13, 65, 17
21, 33
7,48
28, 13
65, 17
7
48
28
13
65
17

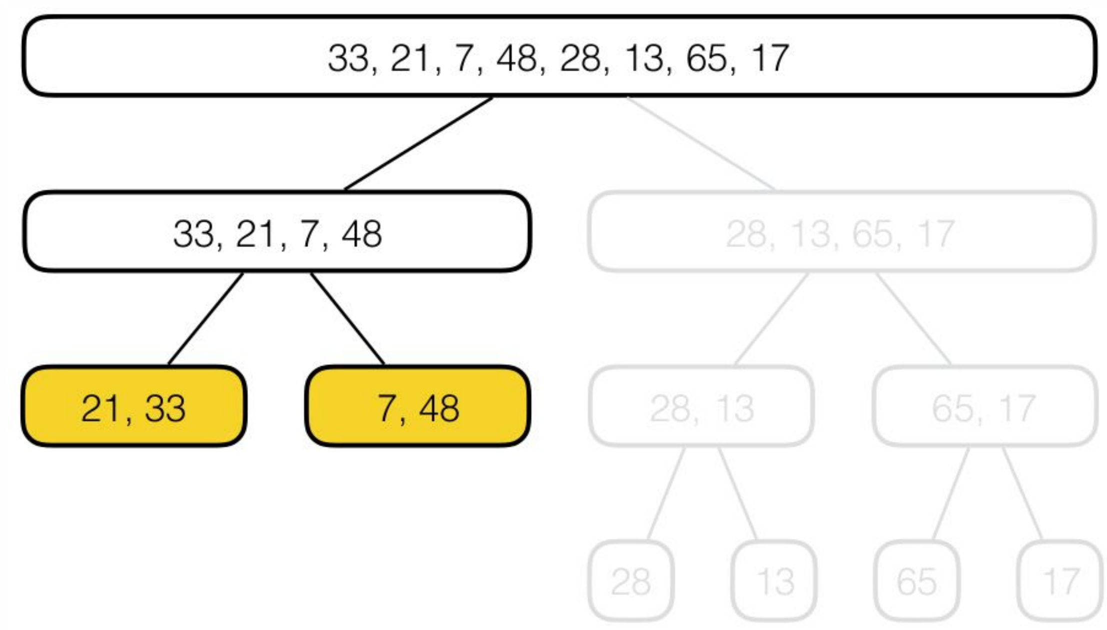
Merge sort.


33, 21, 7, 48, 28, 13, 65, 17
33, 21, 7, 48
28, 13, 65, 17
21, 33
7,48
28, 13
65, 17
7
7
48
28
13
65
17

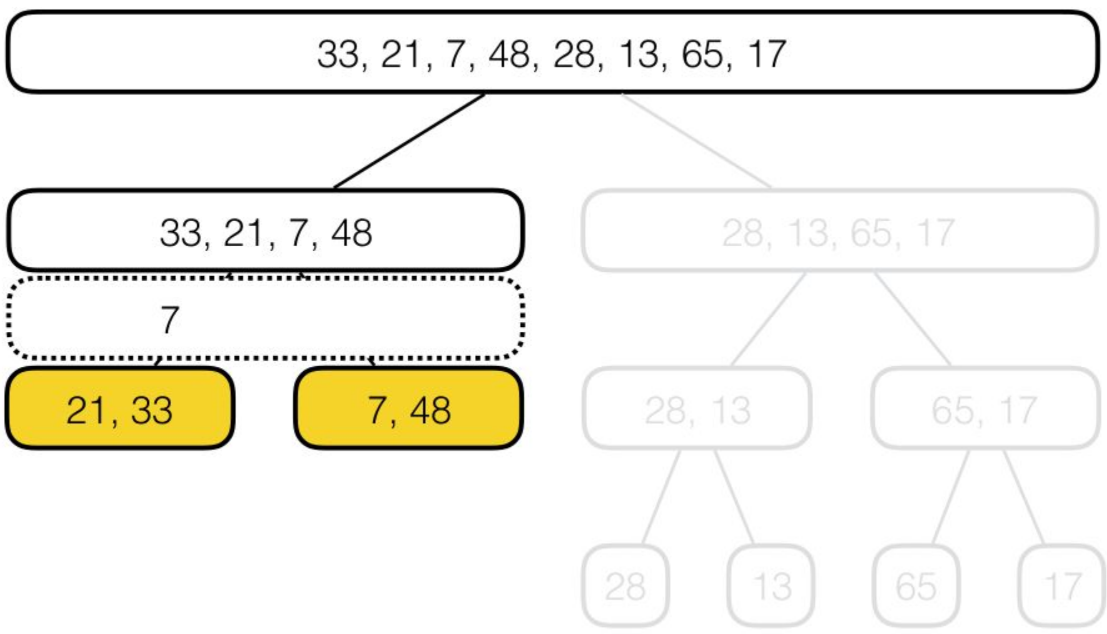
Merge sort.


33, 21, 7, 48, 28, 13, 65, 17
33, 21, 7, 48
28, 13, 65, 17
21, 33
7,48
28, 13
65, 17
7, 48
7
48
28
13
65
17

Merge sort.

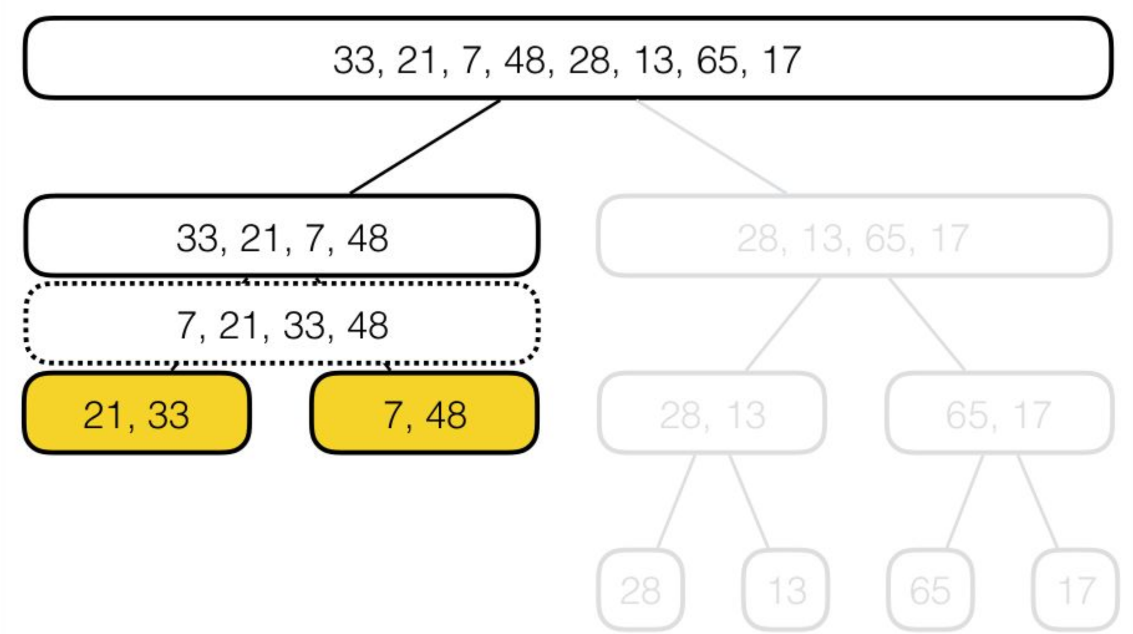

33, 21, 7, 48, 28, 13, 65, 17
33, 21, 7, 48
28, 13, 65, 17
21, 33
7, 48
28, 13
65, 17
28
13
65
17

Merge sort.



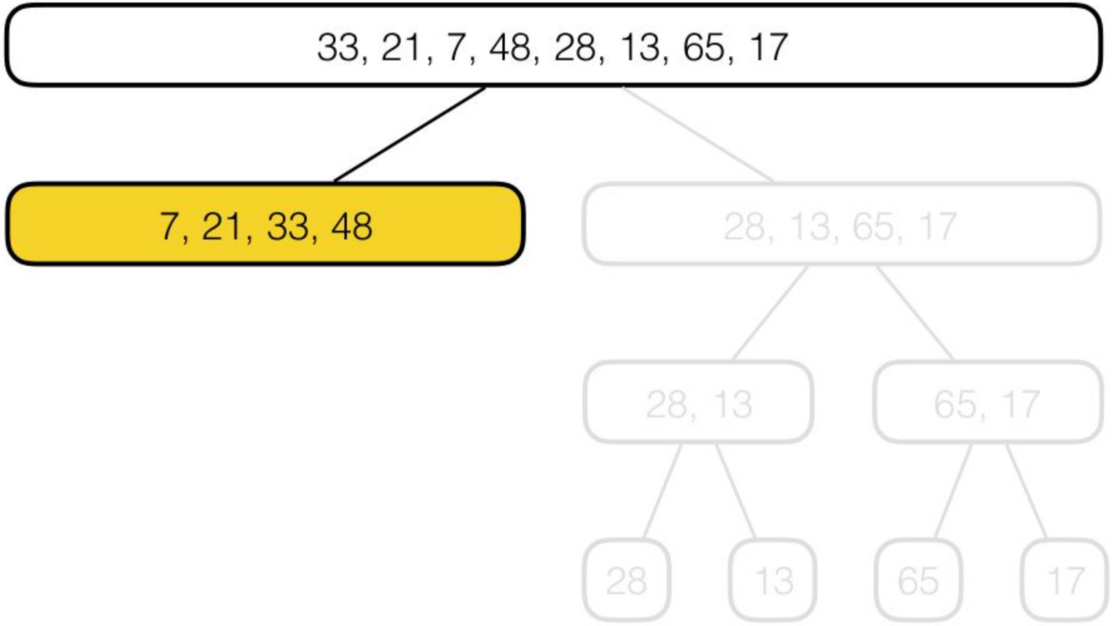
33, 21, 7, 48, 28, 13, 65, 17
33, 21, 7, 48
28, 13, 65, 17
7
21, 33
7, 48
28, 13
65, 17
28
13
65
17

Merge sort.



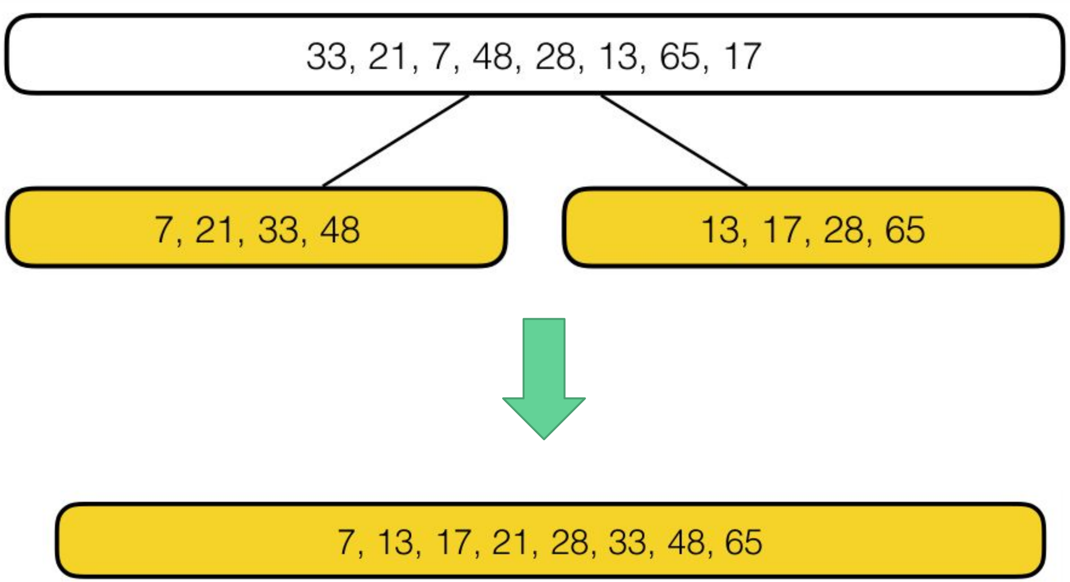
33, 21, 7, 48, 28, 13, 65, 17
33, 21, 7, 48
28, 13, 65, 17
7, 21, 33, 48
21, 33
7, 48
28, 13
65, 17
28
13
65
17

Merge sort.



call merge_sort

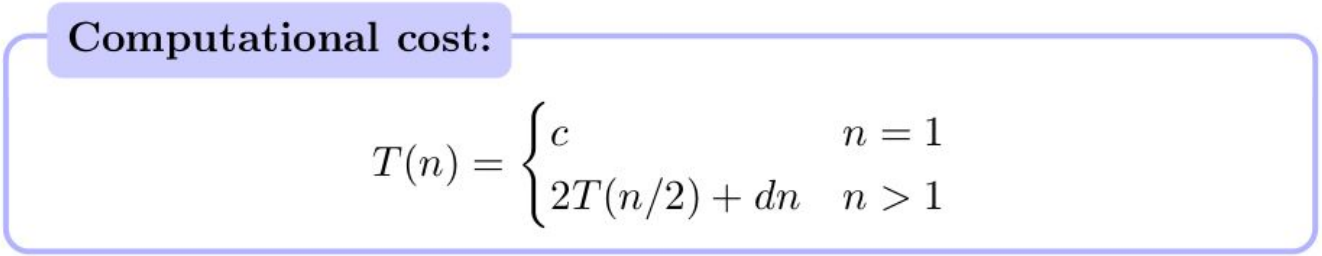
Merge sort.



Merge sort. Cost

Simplifying assumptions:

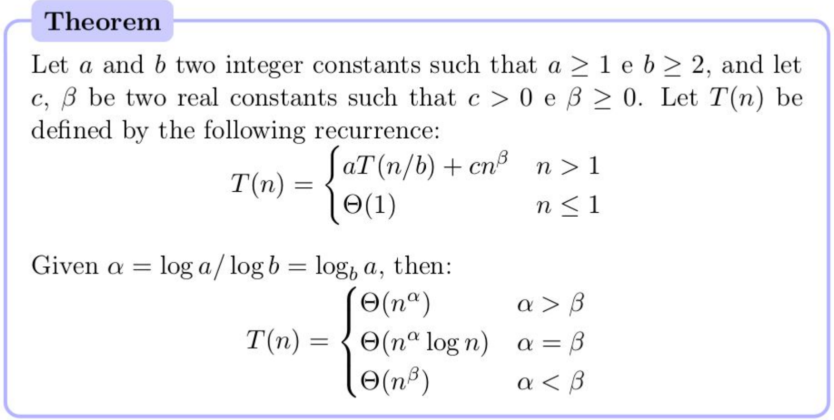
- n = 2^k , i.e. the number of subdivisions is equal to k = log n;
- All the subsequences have size that are exact powers of 2



Computational cost:

$$T(n) = \begin{cases} c & n = 1 \\ 2T(n/2) + dn & n > 1 \end{cases}$$

Merge sort. Cost



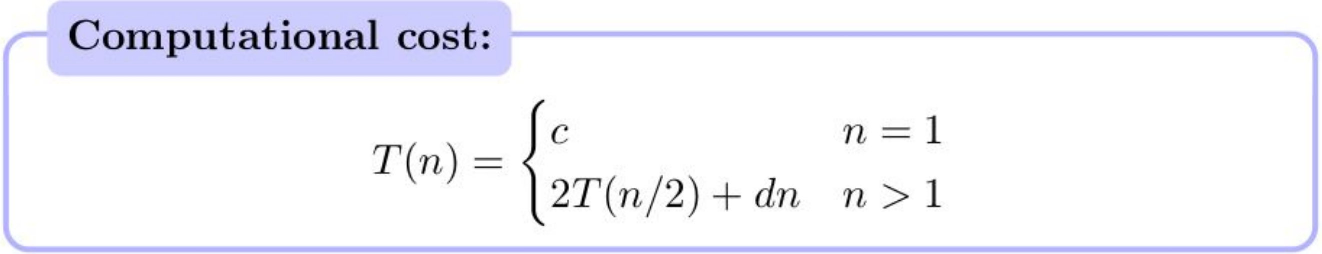
Theorem

Let a and b two integer constants such that $a \geq 1$ e $b \geq 2$, and let c, β be two real constants such that $c > 0$ e $\beta \geq 0$. Let $T(n)$ be defined by the following recurrence:

$$T(n) = \begin{cases} aT(n/b) + cn^{\beta} & n > 1 \\ \Theta(1) & n \leq 1 \end{cases}$$

Given $\alpha = \log a / \log b = \log_b a$, then:

$$T(n) = \begin{cases} \Theta(n^{\alpha}) & \alpha > \beta \\ \Theta(n^{\alpha} \log n) & \alpha = \beta \\ \Theta(n^{\beta}) & \alpha < \beta \end{cases}$$



Computational cost:

$$T(n) = \begin{cases} c & n = 1 \\ 2T(n/2) + dn & n > 1 \end{cases}$$

α = log2 2 = 1

β = 1

→ **θ(n log n)**

No matter the type of input!

Quick sort

- This algorithm is based on a **divide-et-impera** strategy

We will see that…

- Average case: **O(n log n)**, Worst case: **O(n^2)**
- Average case vs worst case: the **multiplicative factor** of QuickSort is better than MergeSort and QuickSort is in-place (does not need a tmp list)
- It is possible to use “heuristic” techniques to avoid the worst case hence it is often preferred to other algorithms

[more info: R. Sedgewick, "Implementing Quicksort Programs". Communications of the ACM, 21(10):847-857, 1978. http://portal.acm.org/citation.cfm?id=359631]

Quick sort: divide step

Input

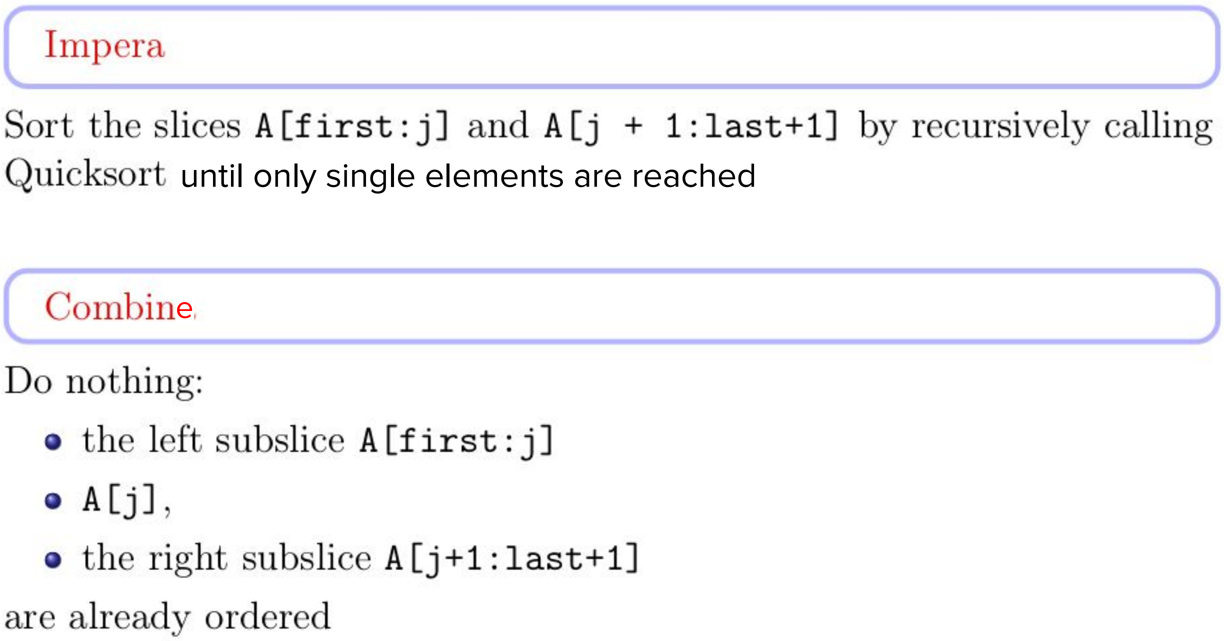
- Sequence A containing n values
- Indexes $\texttt{first}$, $\texttt{last}$ such that $0 \leq \texttt{first} \leq \texttt{last} < n$

Divide

- Select a value $p \in \texttt{A}[\texttt{first} : \texttt{last} + 1]$ called pivot
- Move all the elements in slice $\texttt{A}[\texttt{first} : \texttt{last} + 1]$ in a way that
  $$\forall i \in \texttt{A}[\texttt{first} : \texttt{j}] : \texttt{A}[\texttt{i}] \leq p$$
  $$\forall i \in \texttt{A}[\texttt{j} + 1 : \texttt{last}] : \texttt{A}[\texttt{i}] \geq p$$
- Index j is computed in a way that satisfies such condition
- The pivot is moved in position A[j]

at each iteration the
pivot is put in its right place

Quick sort: impera and combine steps



Impera

Sort the slices `A[first:j]` and `A[j + 1:last+1]` by recursively calling Quicksort until only single elements are reached

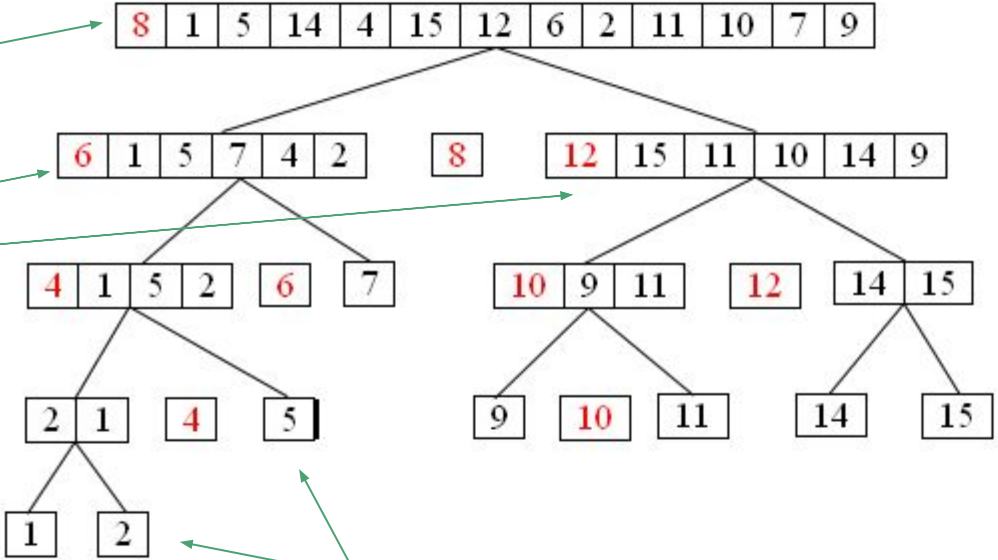
Combine

Do nothing:

- the left subslice `A[first:j]`
- `A[j]`,
- the right subslice `A[j+1:last+1]`

are already ordered

Quick sort



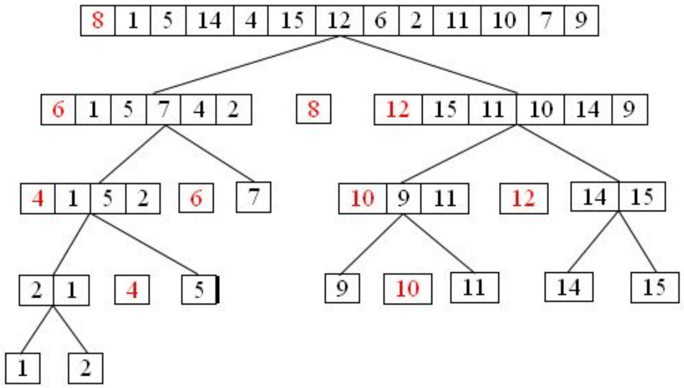
pivot

recursively process
this sublists

single elements

NOTE: it is not great idea to pick the first element as pivot!!!

Quick sort



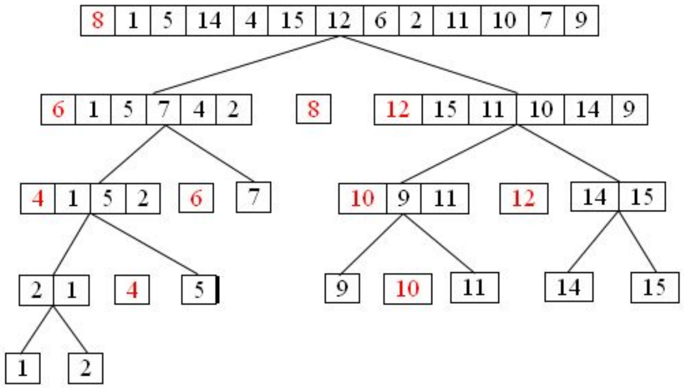
The algorithm makes use of the following methods:

1. pivot : gets the list, a start and end index, sets the **first element** as **pivot** and reorders all the elements in the list from start to end in such a way that **all the elements to the left of the pivot (i.e. having index lower) are smaller than the pivot and all the elements to the right (i.e. with index higher) are bigger than the pivot**. The function **returns the index of the pivot**;

2. recursiveQuickSort: **gets an unordered (sub)list**, with start and end positions, **finds the pivot** and **recursively applies the same procedure to the sublists to the left and right of the pivot** (if sublist has size > 1);

3. quickSort: **gets an unordered list and applies the recursive quick sort procedure to it**.

Pivot method



Pivot partitions the list in two: lower than 6 and higher than 6
Pivot called on this list:

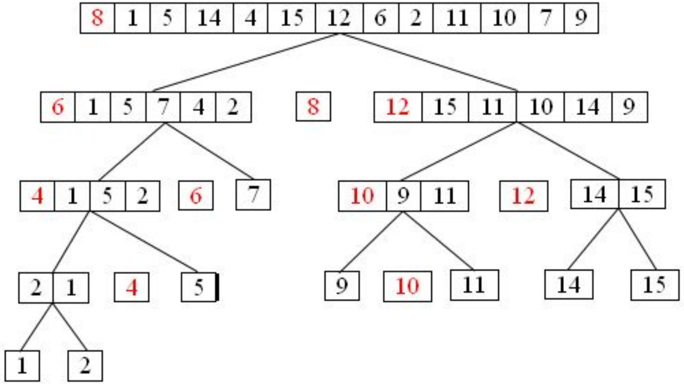
6	1	5	9	7	3	8	9

Two indexes:

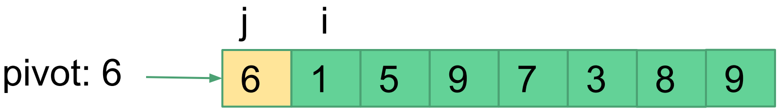
i goes through all the elements

j always points to the last element smaller than pivot

Pivot method



Pivot called on this list:



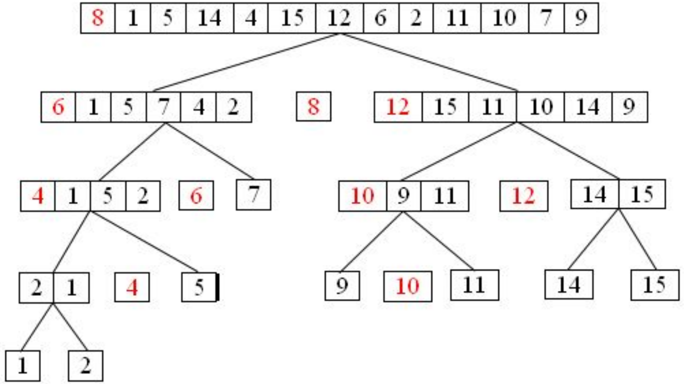
6	1	5	9	7	3	8	9

Two indexes:

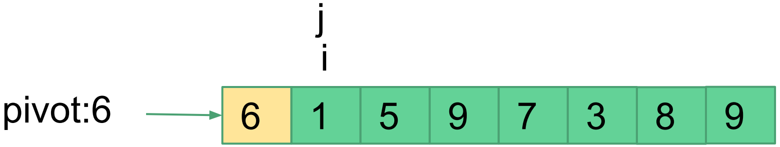
i goes through all the elements

j always points to the last element smaller than pivot

Pivot method



Pivot called on this list:



j
i

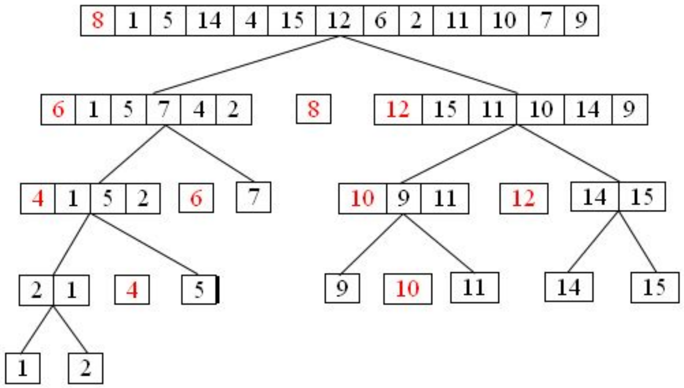
pivot:6

6	1	5	9	7	3	8	9

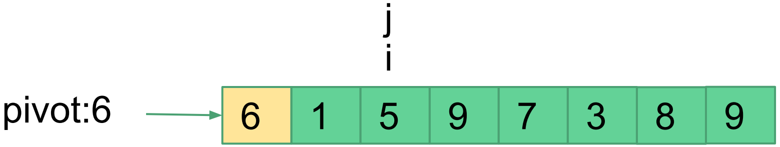
6 > 1
increment j
swap L[i], L[j]

i is incremented at all iterations

Pivot method



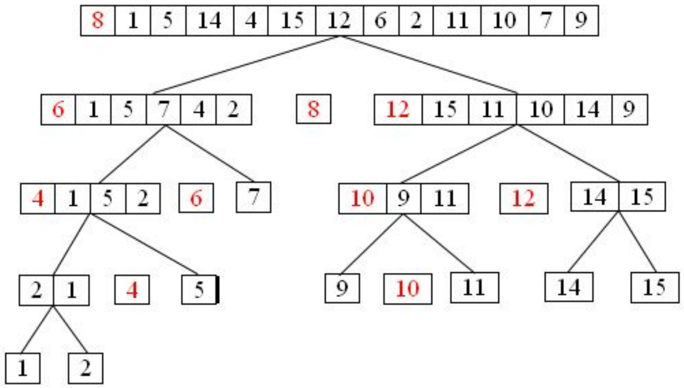
Pivot called on this list:



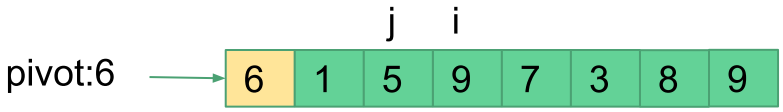
6 > 5
increment j
swap L[i], L[j]

i is incremented at all iterations

Pivot method



Pivot called on this list:



j i

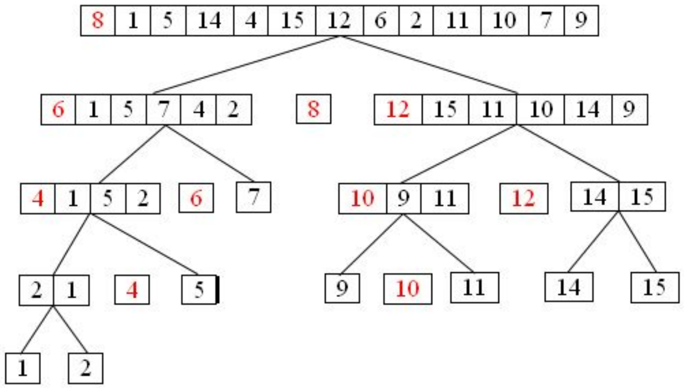
pivot:6

6	1	5	9	7	3	8	9

6 < 9
do nothing

i is incremented at all iterations

Pivot method



Pivot called on this list:

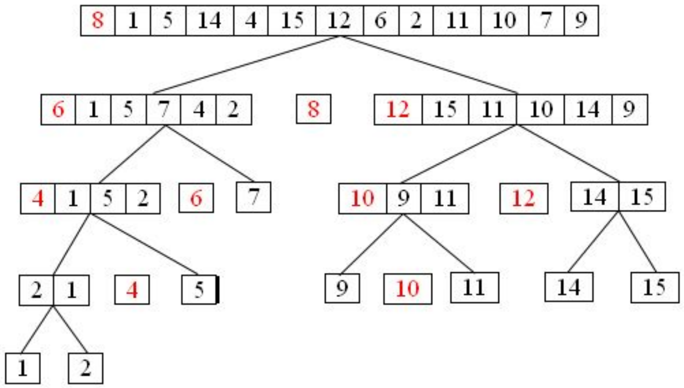
pivot:6

		j		i			
6	1	5	9	7	3	8	9

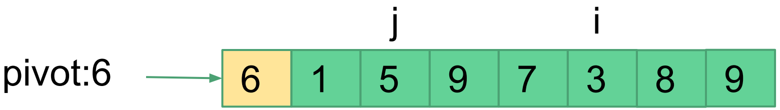
6 < 7
do nothing

i is incremented at all iterations

Pivot method



Pivot called on this list:



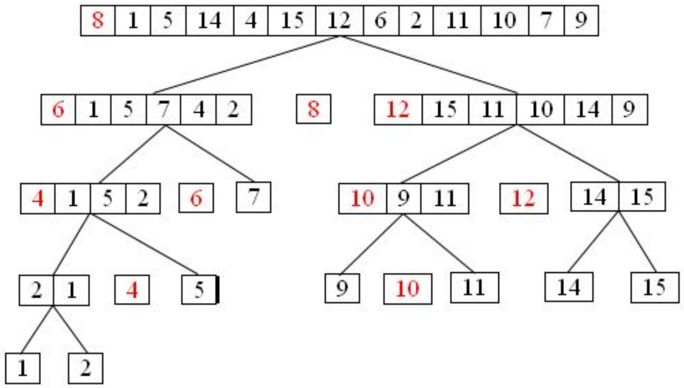
6	1	5	9	7	3	8	9

6 > 3
increment j

j always points to the last element smaller than pivot

i is incremented at all iterations

Pivot method



Pivot called on this list:

pivot:6 →

			j		i		
6	1	5	9	7	3	8	9

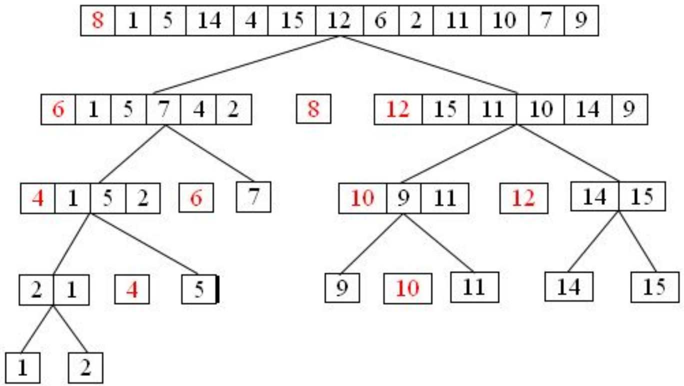
6 > 3
increment j
swap L[i],L[j]

j always points to the last element smaller than pivot

			j		i		
6	1	5	3	7	9	8	9

i is incremented at all iterations

Pivot method



Pivot called on this list:

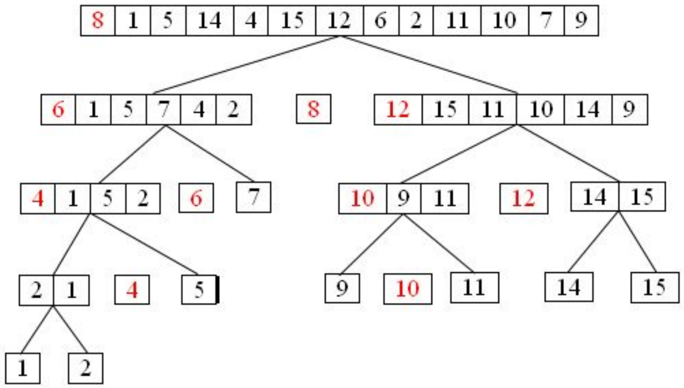
pivot:6 →

			j		i		
6	1	5	3	7	9	8	9

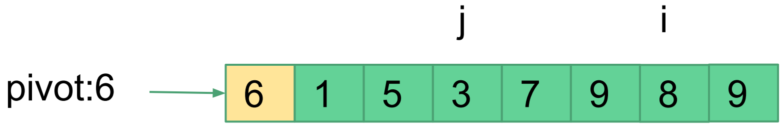
6 < 9
do nothing

i is incremented at all iterations

Pivot method



Pivot called on this list:


j

i

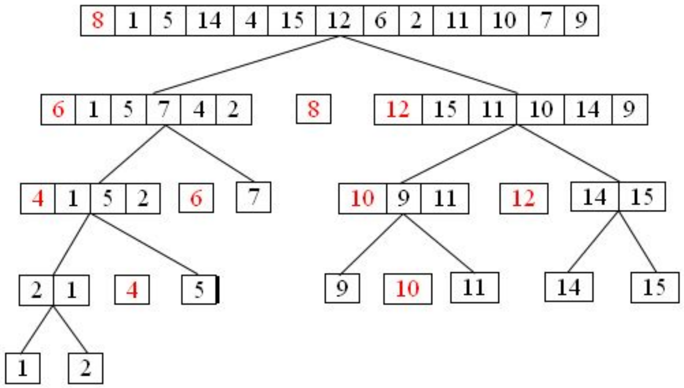
pivot:6

6	1	5	3	7	9	8	9

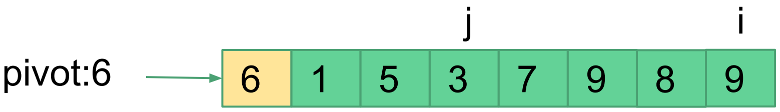
6 < 8
do nothing

i is incremented at all iterations

Pivot method



Pivot called on this list:

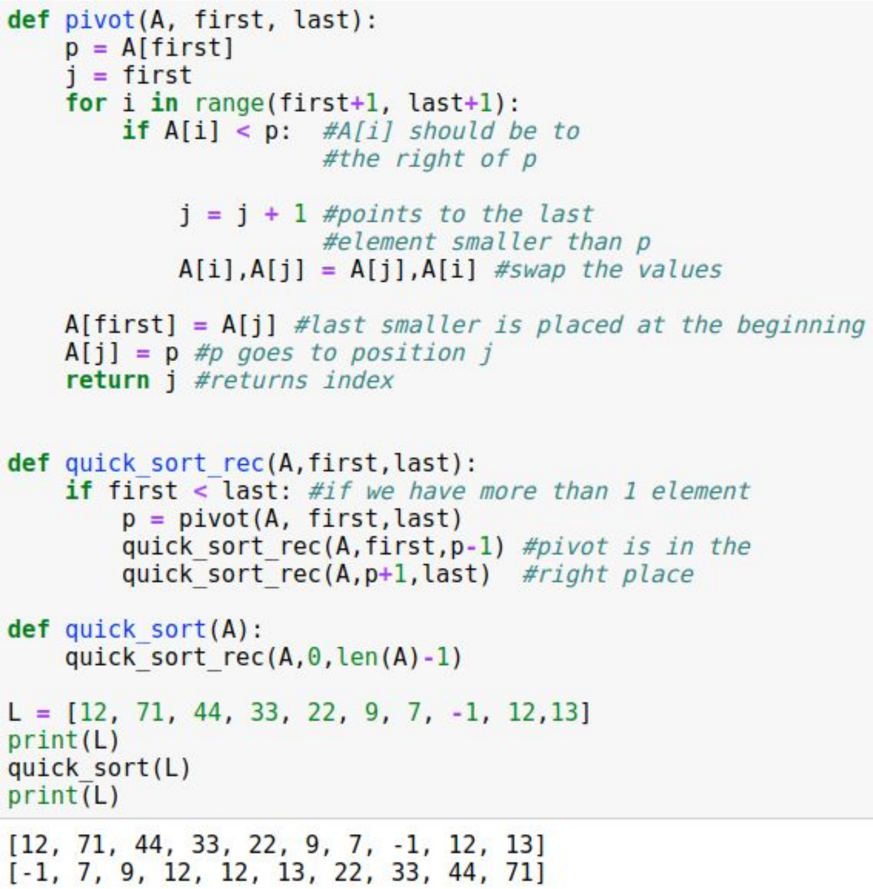


6 < 9
do nothing
END!
swap L[0], L[j]
return j

3	1	5	6	7	9	8	9

i is incremented at all iterations

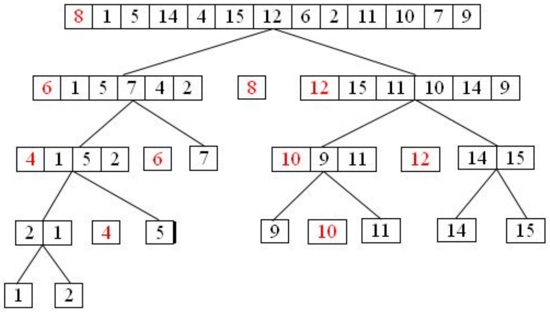
Quick sort: the code



```python
def pivot(A, first, last):
    p = A[first]
    j = first
    for i in range(first+1, last+1):
        if A[i] < p:  #A[i] should be to
                      #the right of p

            j = j + 1 #points to the last
                      #element smaller than p
            A[i],A[j] = A[j],A[i] #swap the values

    A[first] = A[j] #last smaller is placed at the beginning
    A[j] = p #p goes to position j
    return j #returns index


def quick_sort_rec(A,first,last):
    if first < last: #if we have more than 1 element
        p = pivot(A, first,last)
        quick_sort_rec(A,first,p-1) #pivot is in the
        quick_sort_rec(A,p+1,last)  #right place

def quick_sort(A):
    quick_sort_rec(A,0,len(A)-1)

L = [12, 71, 44, 33, 22, 9, 7, -1, 12,13]
print(L)
quick_sort(L)
print(L)
```

```
[12, 71, 44, 33, 22, 9, 7, -1, 12, 13]
[-1, 7, 9, 12, 12, 13, 22, 33, 44, 71]
```

Quick sort: complexity



Cost of `pivot()`

- $\Theta(n)$ [it is only one for loop]

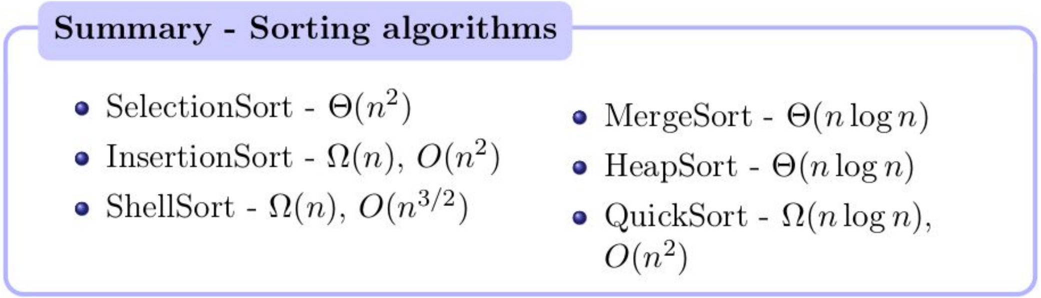
Cost of Quicksort: depends on the partitioning

- Worst partitioning
  - Given a list of size n, the list is subdivided in two sublist of size 0 and $n-1$
  - $T(n) = T(n-1) + T(0) + \Theta(n) = T(n-1) + \Theta(n) = \Theta(n^2)$
  - Question: When do you get the worst case?
- Best partitioning
  - Given a list of size n, the list is subdivided in two sublist of size $n/2$
  - $T(n) = 2T(n/2) + \Theta(n) = \Theta(n \log n)$

worst case, when list is already sorted (theorem not seen)

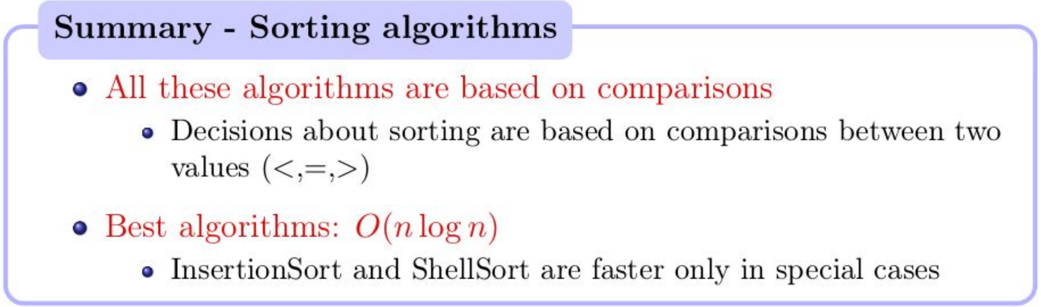
Pick pivot as random value to heuristically reduce the probability of worst case!

Sorting methods: based on comparisons (?!?)



Summary - Sorting algorithms

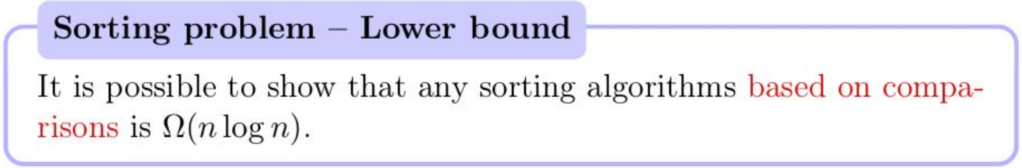
- SelectionSort - $\Theta(n^2)$
- InsertionSort - $\Omega(n)$, $O(n^2)$
- ShellSort - $\Omega(n)$, $O(n^{3/2})$
- MergeSort - $\Theta(n \log n)$
- HeapSort - $\Theta(n \log n)$
- QuickSort - $\Omega(n \log n)$, $O(n^2)$



Summary - Sorting algorithms

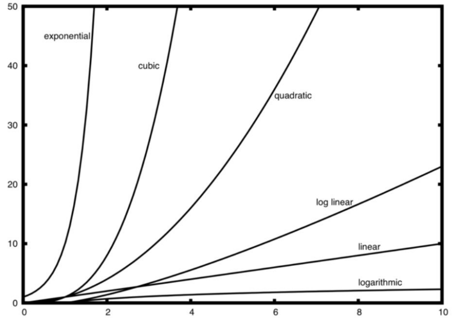
- All these algorithms are based on comparisons
  - Decisions about sorting are based on comparisons between two values (<,=,>)
- Best algorithms: $O(n \log n)$
  - InsertionSort and ShellSort are faster only in special cases

common feature of all methods seen so far



Sorting problem – Lower bound

It is possible to show that any sorting algorithms based on comparisons is $\Omega(n \log n)$.



Counting Sort: NOT based on comparisons

Assumption

The numbers to be sorted are included in a range [0 . . . k − 1]

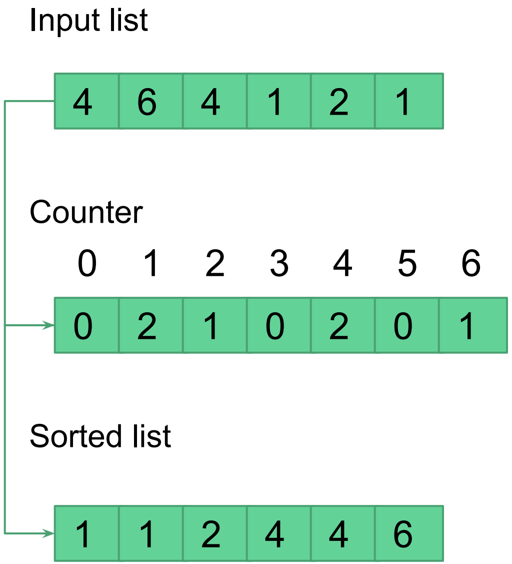
Idea

- Build a list B with k entries, where B[i] contains the number of times that i is contained in A (**B is a list of counters**).
- Put back together the elements in A, using the counters in B (**adding elements have value N > 0, N times**)

Possible improvements

The interval might not be limited to [0 . . . k − 1]; any **known** interval [i, j] can work. In such case, you must subtract i from each number.

First, find the minimum value and subtract all the other values to it (remembering to add them back at the end)



Input list

4	6	4	1	2	1

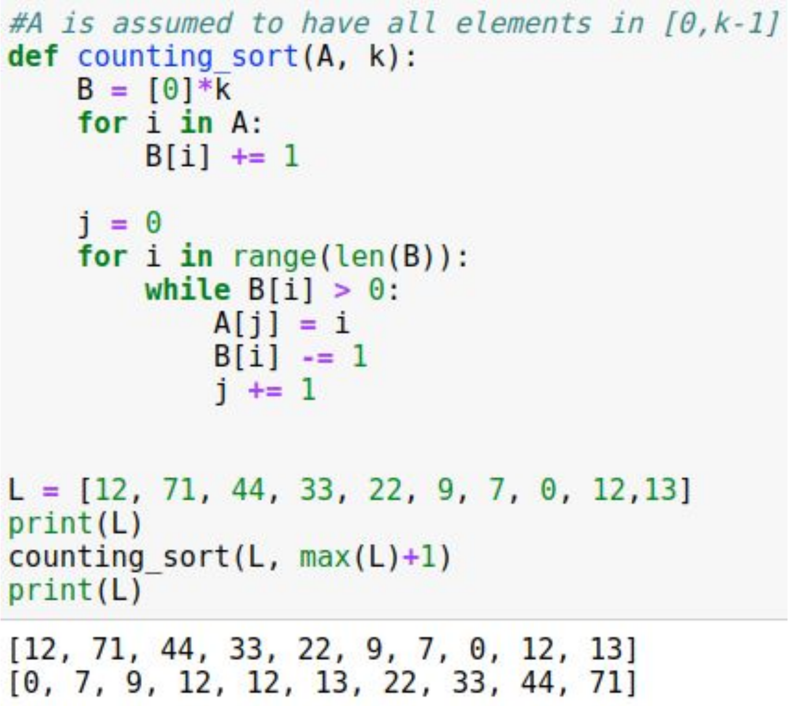
Counter

0	1	2	3	4	5	6
0	2	1	0	2	0	1

Sorted list

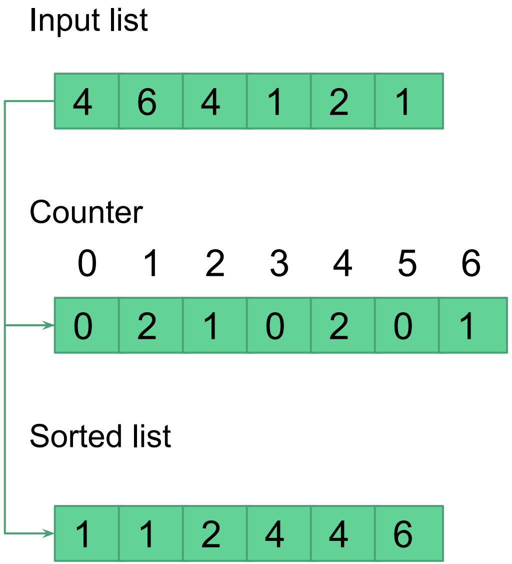
1	1	2	4	4	6

Counting Sort: code



```python
#A is assumed to have all elements in [0,k-1]
def counting_sort(A, k):
    B = [0]*k
    for i in A:
        B[i] += 1

    j = 0
    for i in range(len(B)):
        while B[i] > 0:
            A[j] = i
            B[i] -= 1
            j += 1


L = [12, 71, 44, 33, 22, 9, 7, 0, 12,13]
print(L)
counting_sort(L, max(L)+1)
print(L)
```

```
[12, 71, 44, 33, 22, 9, 7, 0, 12, 13]
[0, 7, 9, 12, 12, 13, 22, 33, 44, 71]
```



Input list

4 6 4 1 2 1

Counter

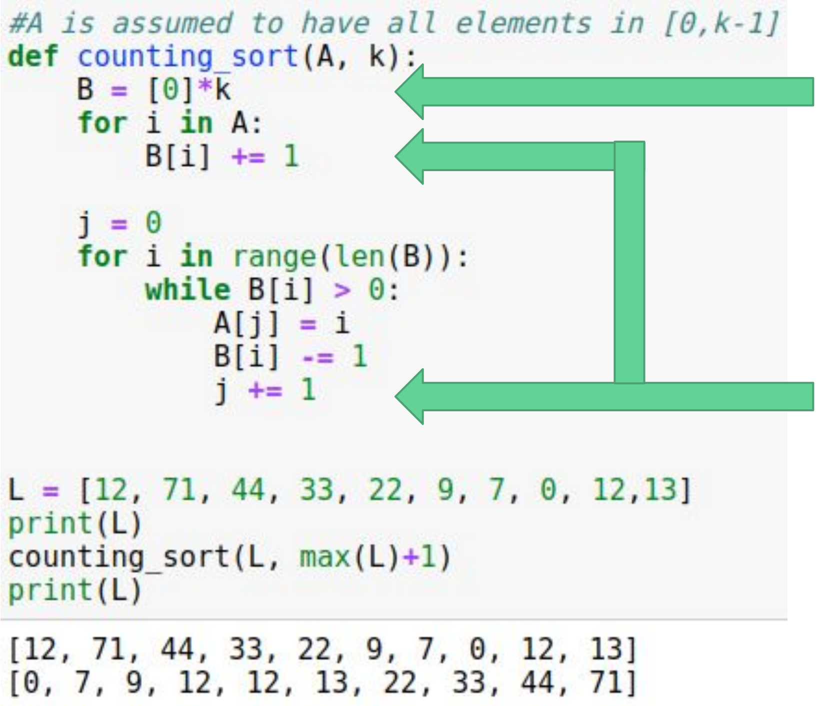
0 1 2 3 4 5 6

0 2 1 0 2 0 1

Sorted list

1 1 2 4 4 6

Counting Sort: complexity



```
#A is assumed to have all elements in [0,k-1]
def counting_sort(A, k):
    B = [0]*k
    for i in A:
        B[i] += 1

    j = 0
    for i in range(len(B)):
        while B[i] > 0:
            A[j] = i
            B[i] -= 1
            j += 1


L = [12, 71, 44, 33, 22, 9, 7, 0, 12,13]
print(L)
counting_sort(L, max(L)+1)
print(L)
```

```
[12, 71, 44, 33, 22, 9, 7, 0, 12, 13]
[0, 7, 9, 12, 12, 13, 22, 33, 44, 71]
```

Size of A: **n**

adds k zeros to B : **θ(k)**

executed once for each element in A: **θ(n)**

Overall, complexity: **θ(n + k)**

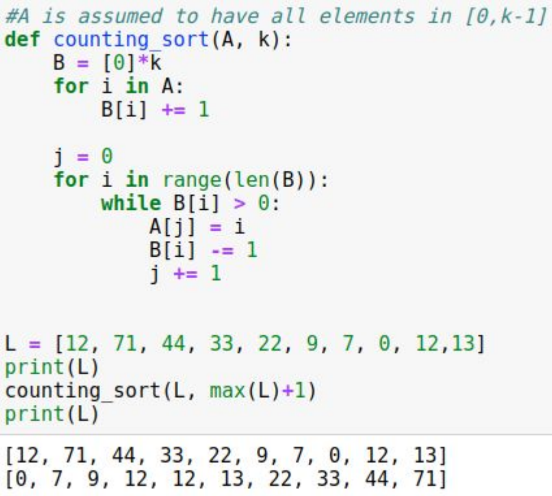
Counting Sort: complexity

Complexity of CountingSort

- $\Theta(n+k)$
- If k is $O(n)$, then the complexity of Counting Sort is $\Theta(n)$

Counting Sort and lower bounds for sorting

- Counting Sort is not based on comparisons
- If k is $\Omega(n^3)$, this algorithm is worse than any other algorithm seen so far

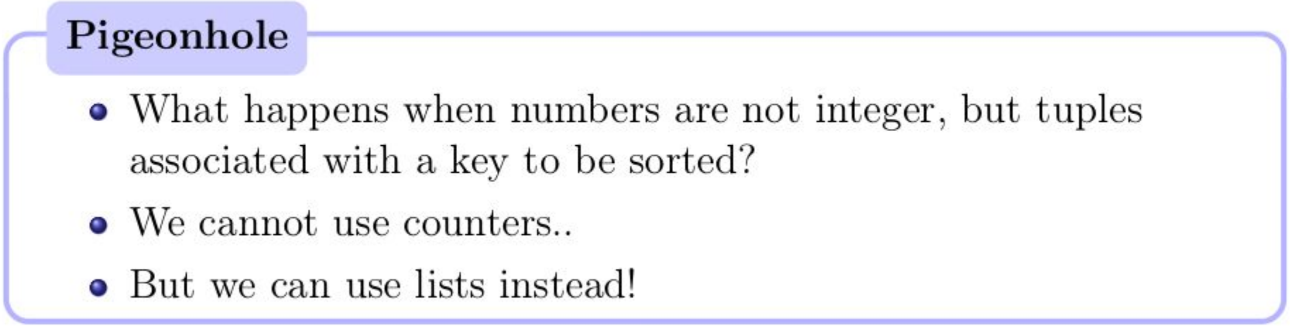


```
#A is assumed to have all elements in [0,k-1]
def counting_sort(A, k):
    B = [0]*k
    for i in A:
        B[i] += 1

    j = 0
    for i in range(len(B)):
        while B[i] > 0:
            A[j] = i
            B[i] -= 1
            j += 1


L = [12, 71, 44, 33, 22, 9, 7, 0, 12,13]
print(L)
counting_sort(L, max(L)+1)
print(L)
```

```
[12, 71, 44, 33, 22, 9, 7, 0, 12, 13]
[0, 7, 9, 12, 12, 13, 22, 33, 44, 71]
```

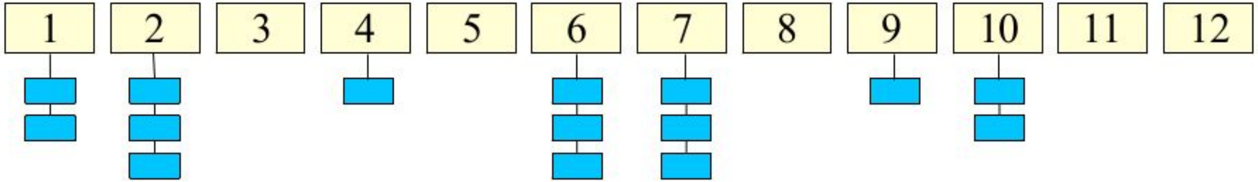
Counting Sort

This can be used to sort every object that can be associated to a sortable key (can use lists of objects)



Pigeonhole

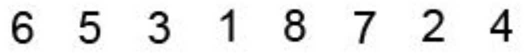
- What happens when numbers are not integer, but tuples associated with a key to be sorted?
- We cannot use counters..
- But we can use lists instead!



Exercise: Bubble Sort

Compare two consecutive elements, if the two are in inverted order swap them, move up one place and continue until you arrive at the end of the list.

Restart from the beginning and continue until no more swaps are needed.



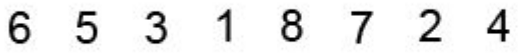
[from wikipedia]

Exercise: Bubble Sort

Sketch of the code:

```
procedure BubbleSort(A:list)
  swapFlag ← true
  while swapFlag do
    swapFlag ← false
    for i ← 0 to length(A)-2  do
      if A[i] > A[i+1] then
        swap( A[i], A[i+1] )

               swapFlag ← true
```

Optimization: at the end of an iteration one element is placed in its final position (at the end of the list).



[from wikipedia]