# Extended Introduction to Computer Science
# CS1001.py

## Lecture 7:
## Time Complexity (reminder);
## Merging of sorted lists;
## Higher Order Functions and Lambda Expressions

Instructors: Benny Chor, Amir Rubinstein

Teaching Assistants: Michal Kleinbort, Amir Gilad

School of Computer Science
Tel-Aviv University
Spring Semester 2016-7
http://tau-cs1001-py.wikidot.com

# Lecture 6: Highlights

- Basic algorithms:
  - Binary search
  - Sorting
- Complexity of algorithms
  - The O(…) notation – a formal definition for complexity
  - Worst / best case analysis
  - Tractable and intractable algorithms

# Lecture 7: Plan

- Complexity of algorithms (reminder and summary)
  - another example: merging of sorted lists

- Anonymous functions (lambda expressions), High order functions

# Big O Notation

- We say that a function $f(n)$ is $O(g(n))$ if there is a constant $c$ such that for large enough $n$,

  $|f(n)| \leq c \cdot |g(n)|$

- We denote this as $f(n) = O(g(n))$

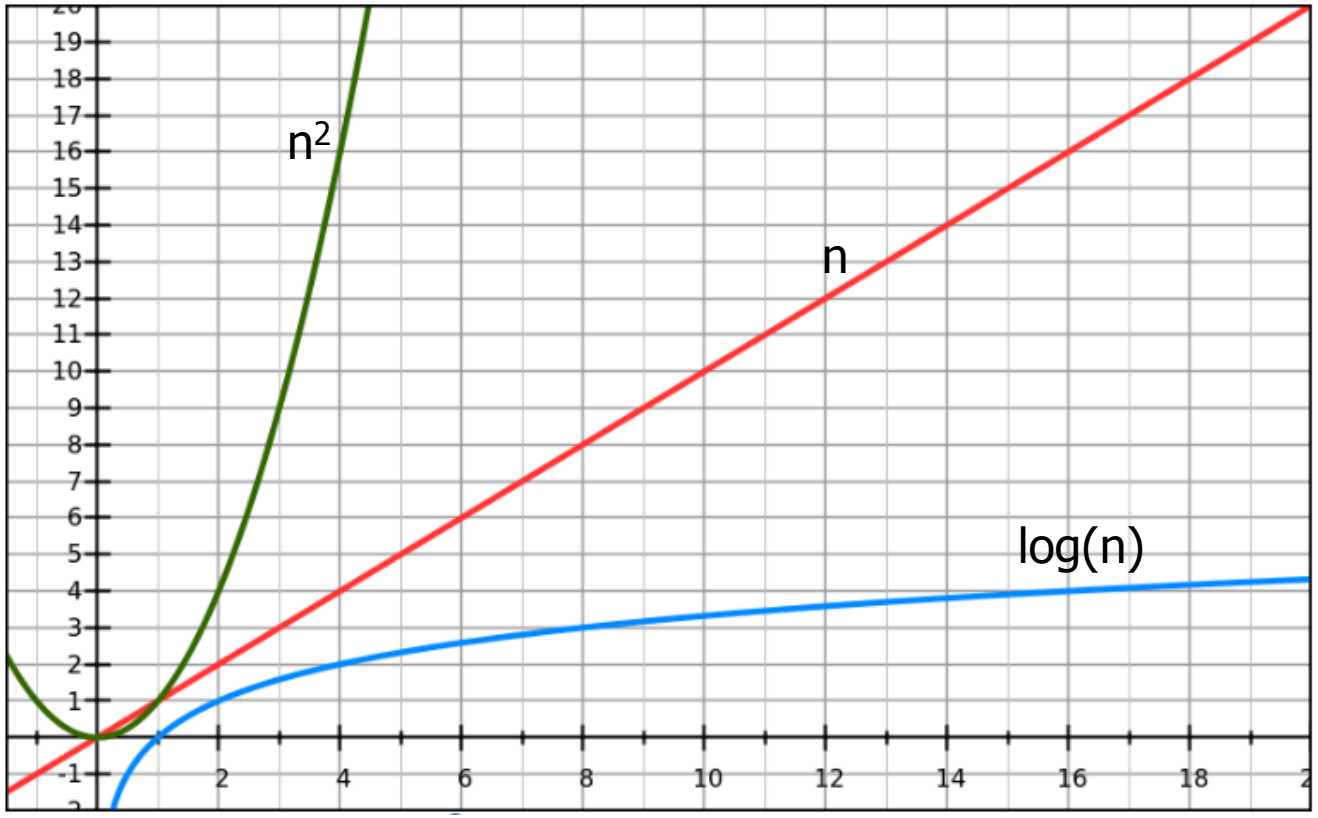
  For example:

  - $5n \cdot \log_2(n) = O(n \log(n))$ [where did the log base disappear?]
  - $2\log_2(n) = O(n)$ [not the tightest possible bound]
  - $1000 \cdot n \cdot \log_2(n) = O(n^2)$ [not the tightest possible bound]
  - $2^{n/100} \neq O(n^{100})$

# Basic algorithms – time complexity summary

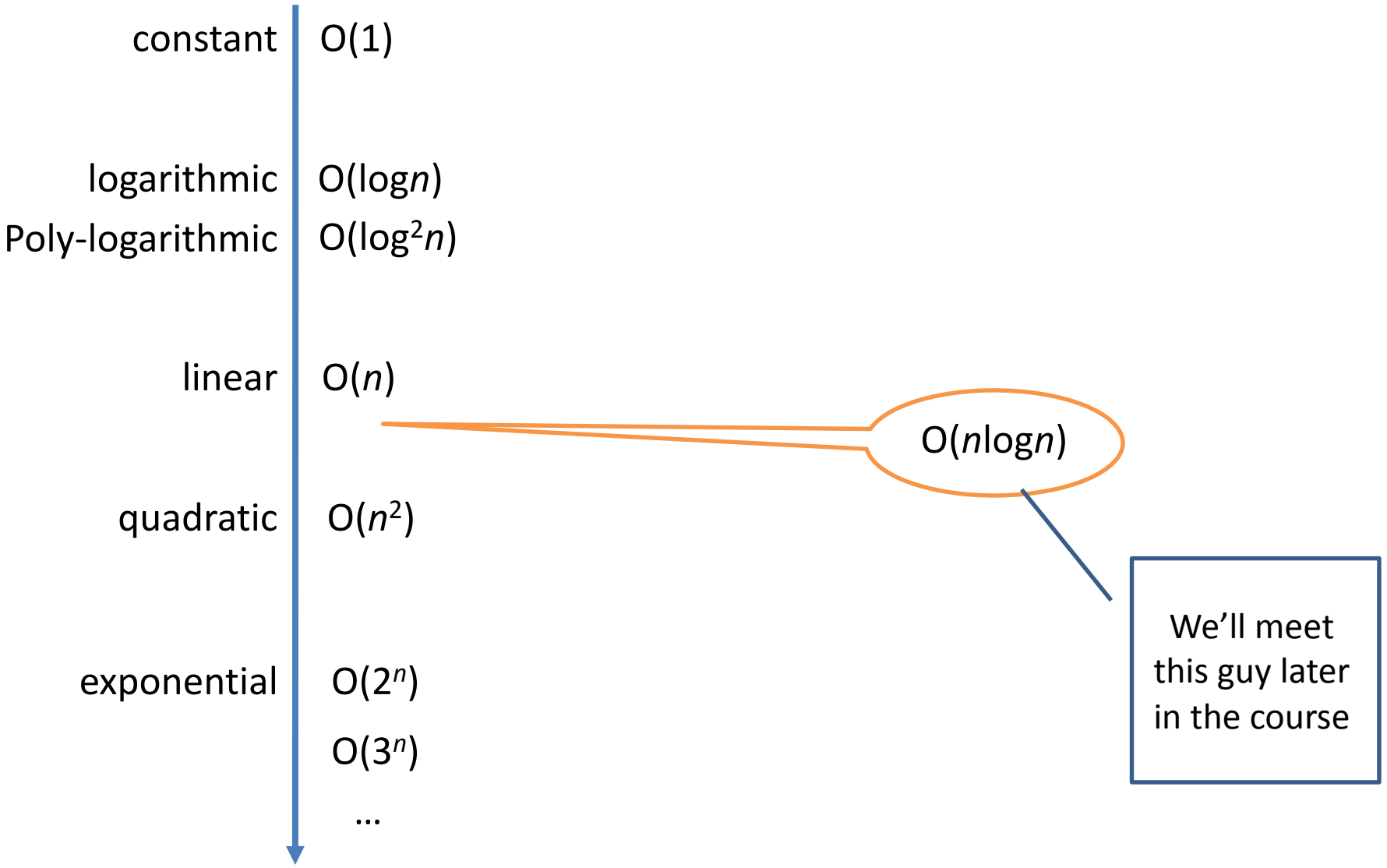
| Algorithm | Time complexity | |
|---|---|---|
| | Best case scenario | Worst case scenario |
| Search - sequential<br>Binary search | O(1)<br>O(1) | O(n)<br>O(logn) |
| Selection sort | $O(n^2)$ | $O(n^2)$ |
| Merge (today) | O(n+m) | O(n+m) |

# Complexity classes




- Logarithmic: input x2 → time + constant
- Linear: input x2 → time x2 (approximately)
- Quadratic: input x2 → time $x2^2$ (approximately)

# Complexity Hierarchy



# Tractability - Basic Distinction:

How would execution time for a very fast, modern processor ($10^{10}$ ops per second, say) vary for a task with the following time complexities and n = input sizes?

| | 10 | 20 | 30 | 40 | 50 | 60 |
|---|---|---|---|---|---|---|
| n | 1.0E-09 seconds | 2.0E-09 seconds | 3.0E-09 seconds | 4.0E-09 seconds | 5.0E-09 seconds | 6.0E-09 seconds |
| $n^2$ | 1.0E-08 seconds | 4.0E-08 seconds | 9.0E-08 seconds | 1.6E-07 seconds | 2.5E-07 seconds | 3.6E-07 seconds |
| $n^3$ | 1.0E-07 seconds | 8.0E-07 seconds | 2.7E-06 seconds | 6.4E-06 seconds | 1.3E-05 seconds | 2.2E-05 seconds |
| $n^5$ | 1.0E-05 seconds | 0.00032 seconds | 0.00243 seconds | 0.01024 seconds | 0.03125 seconds | 0.07776 seconds |
| $2^n$ | 1.02E-07 seconds | 1.05E-04 seconds | 0.107 seconds | 1.833 minutes | 1.303 days | 0.64 years |
| $3^n$ | 5.9E-06 seconds | 0.35 seconds | 5.72 hours | 38.55 years | 22764 centuries | 1.34E+09 centuries |

Modified from Garey and Johnson's classical book

Polynomial time = tractable. Exponential time = intractable.

# Time Complexity - What is tractable in Practice?

- A polynomial-time algorithm is good.
- An exponential-time algorithm is bad.
- $n^{100}$ is polynomial, hence good.
- $2^{n/100}$ is exponential, hence bad.

Yet for input of size n = 4000, the $n^{100}$ time algorithm takes more than $10^{35}$ centuries on the above mentioned machine, while the

$2^{n/100}$ algorithm runs in just under two minutes.

# Time Complexity - Advice

- Trust, but check! Don't just mumble "polynomial-time algorithms are good", "exponential-time algorithms are bad" because the lecturer told you so.
- Asymptotic run time and the O notation are important, and in most cases help clarify and simplify the analysis.
- But when faced with a concrete task on a specific problem size, you may be far away from "the asymptotic".
- In addition, constants hidden in the O notation may have unexpected impact on actual running time.

# Time Complexity – Advice (cont.)

- We will employ both asymptotic analysis and direct measurements of the actual running time.
- For direct measurements, we will use either the time package and the time.clock() function.
- Or the timeit package and the timeit.timeit() function.
- Both have some deficiencies, yet are highly useful for our needs.

# Worst / Best Case Complexity

- In many cases, for the same size of input, the content of the input itself affects the complexity.

- Examples we have seen? Examples in which this is not the case?
  - binary search
  - selection sort

- Note that this statement is completely nonsense:
  "The best time complexity is when n is very small…"

- Often the average complexity is more informative (e.g. when the worst case is rather rare).
  However analyzing it is usually more complicated, and requires some knowledge on the distribution of inputs.
  Assuming distribution is uniform: $T_{average}(n) = \frac{\sum_{I \in Inputs(n)} T(I)}{|Inputs(n)|}$

  examples from our course?
  - Quicksort runs on average in O(nlogn) (also best case)
  - soon: Hash table chains are of length O(n/m) on average

# Merge

# Merge

- The computational problem:
  - Input: two sorted sequences of elements
  - Output: one sorted sequence containing all elements in both sequences

- Possible algorithms?

# Merge - possible algorithms

- Simply concatenate both lists and sort them all

```
def merge_by_sort(A,B):
     """ merging two lists """
    C = A + B
    selection_sort(C)
    return C
```

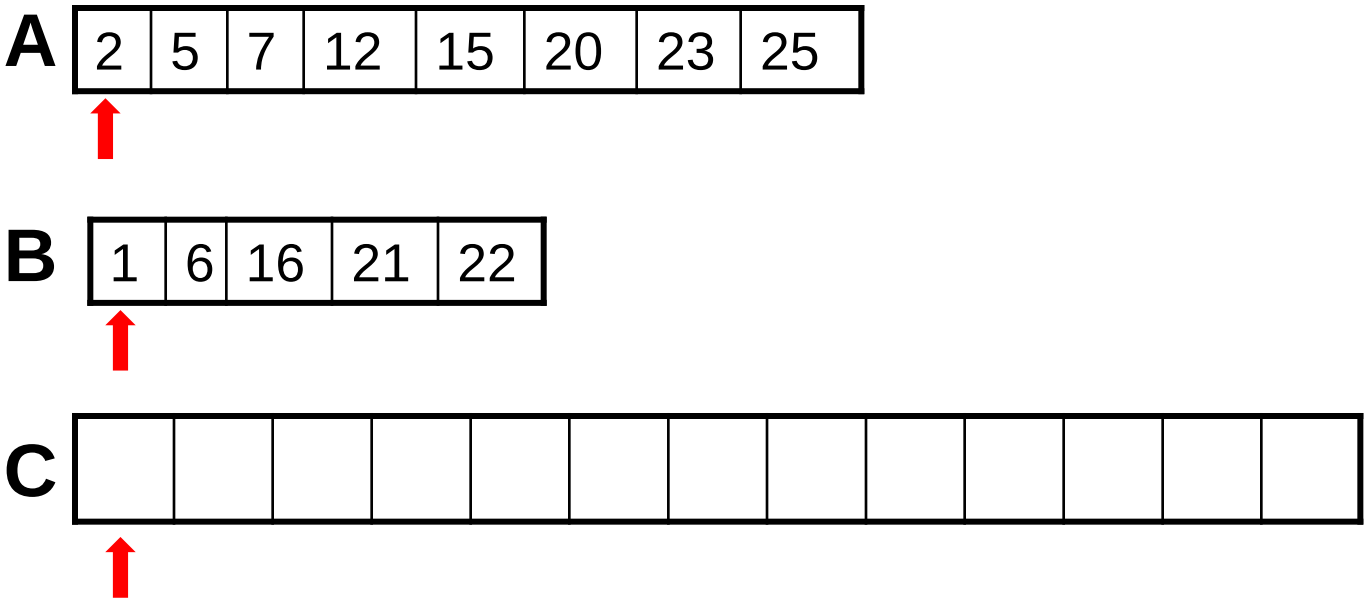
  - However, this solution does not take advantage of the input lists being sorted already.

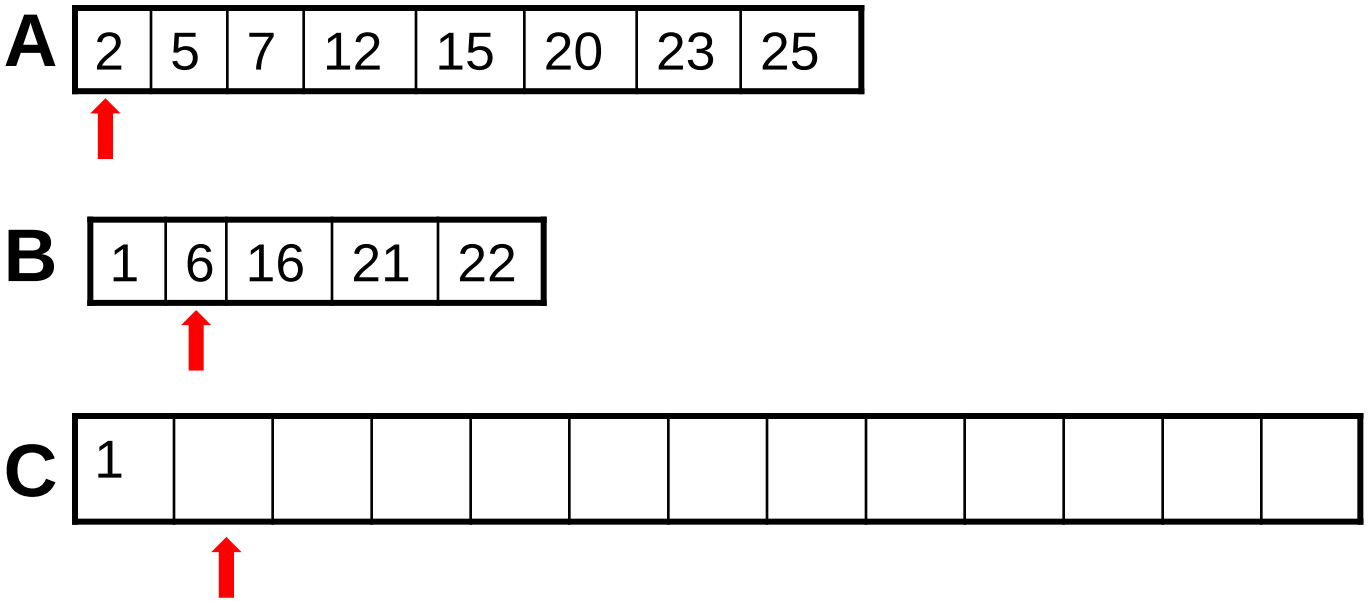
- 3 “running” indices, for input lists (A, B) and the output (C).

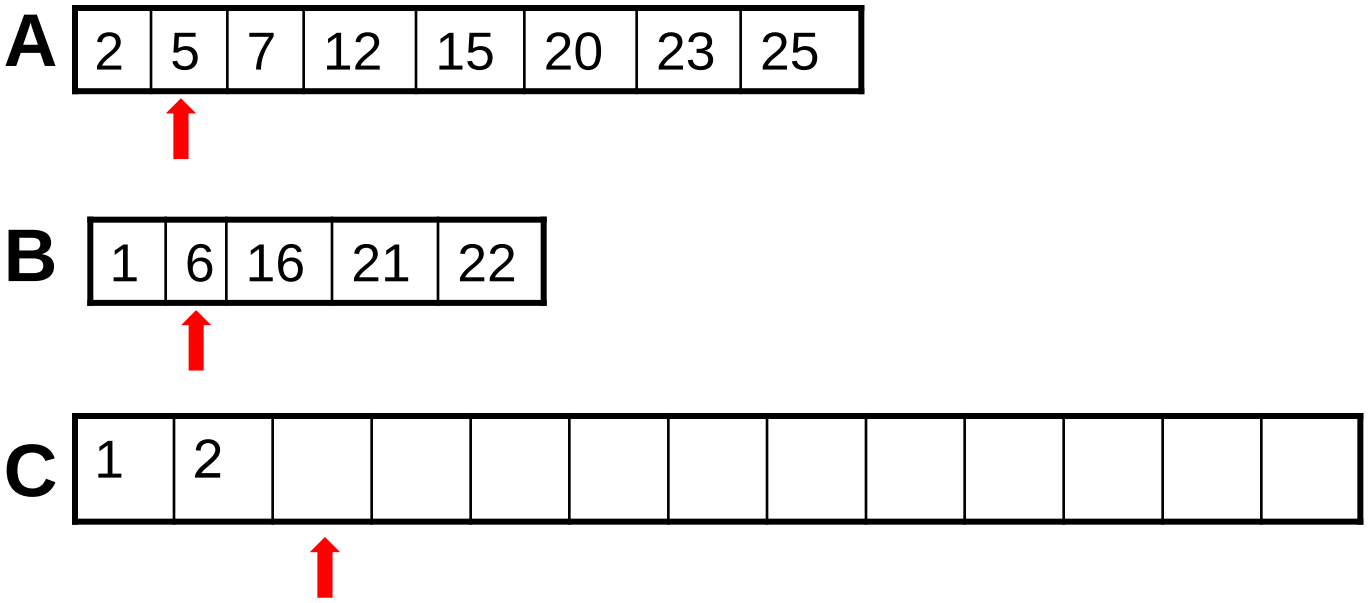
  At each iteration, select the minimal element from A or B and copy it to C.

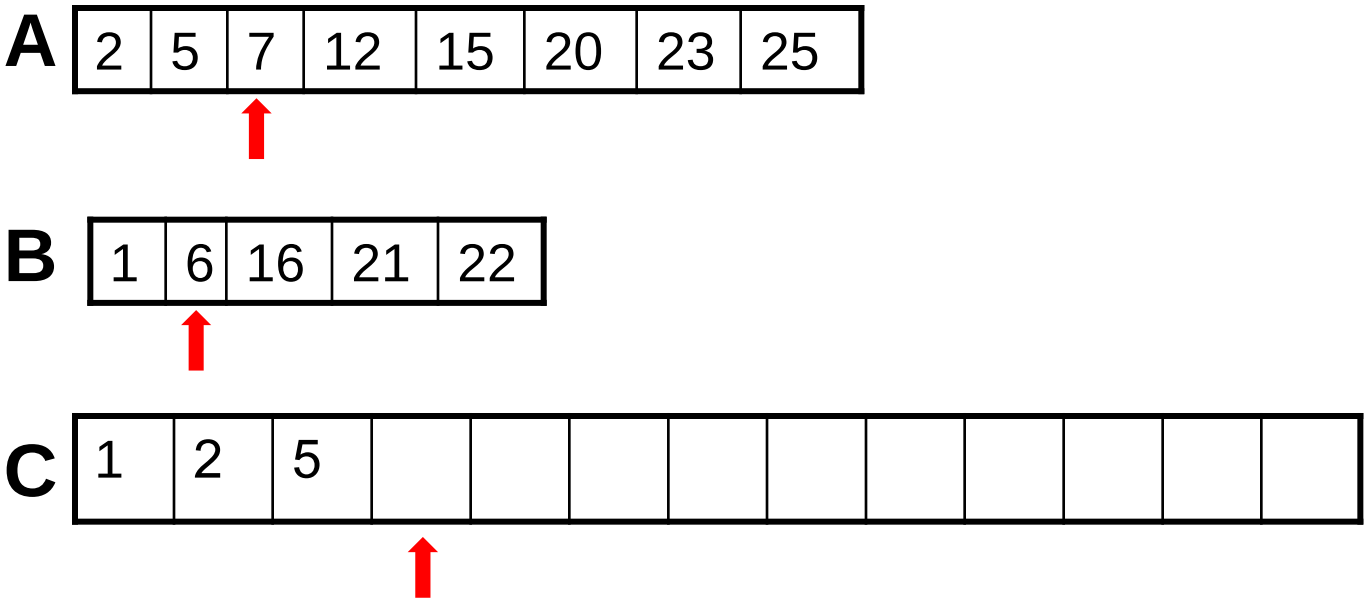
  What happens when one of the lists is completed?

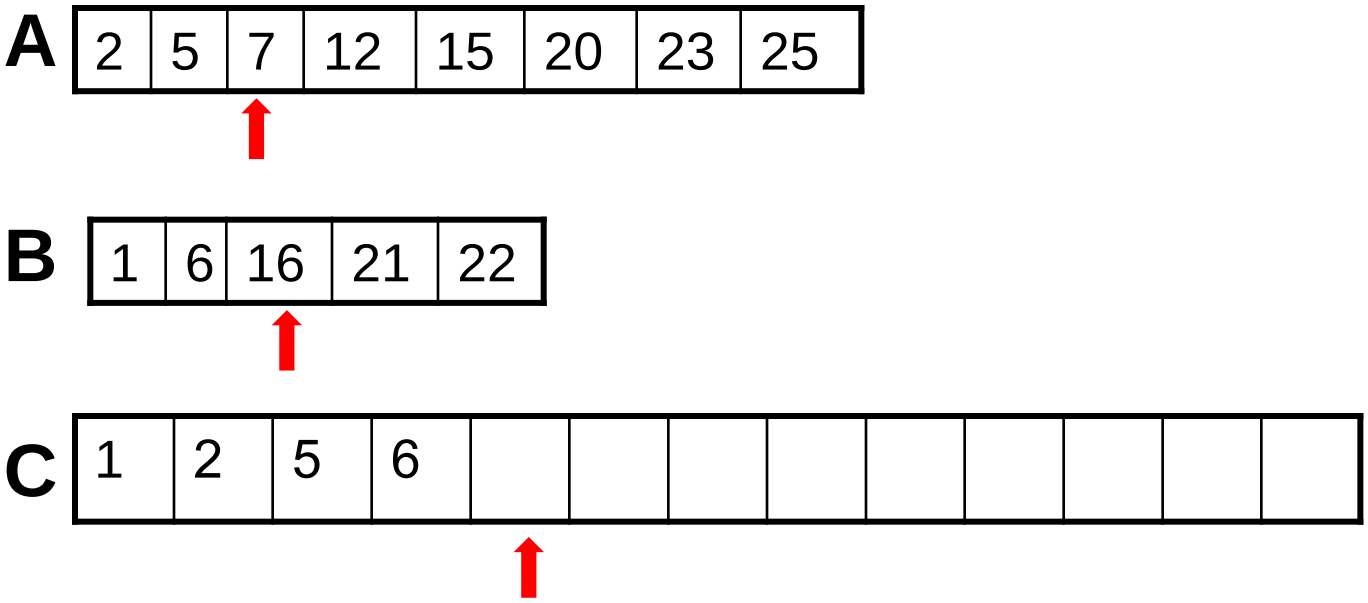
# Example

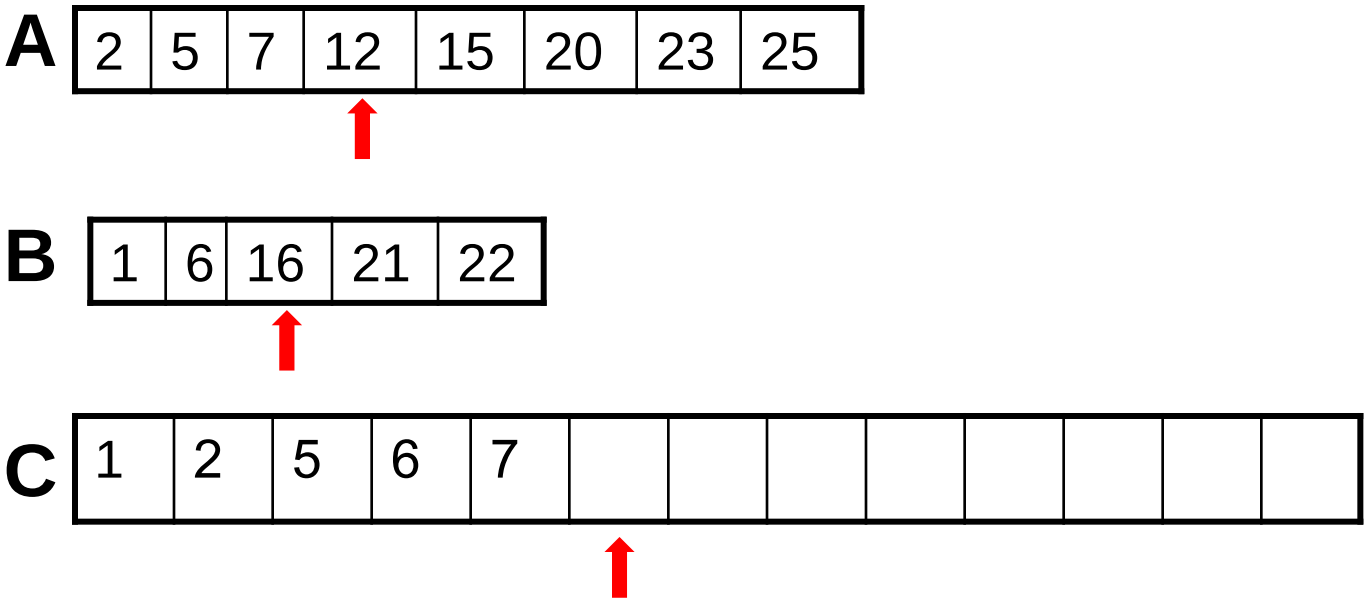


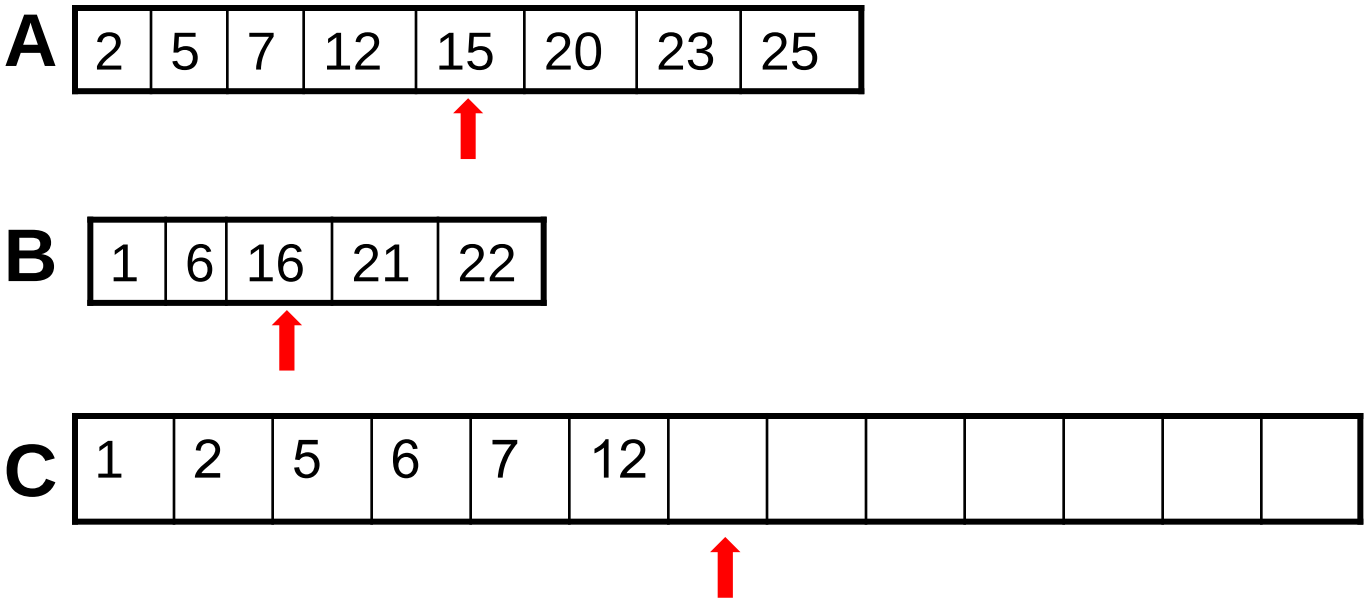

A
2
5
7
12
15
20
23
25
B
1
6
16
21
22
C
1

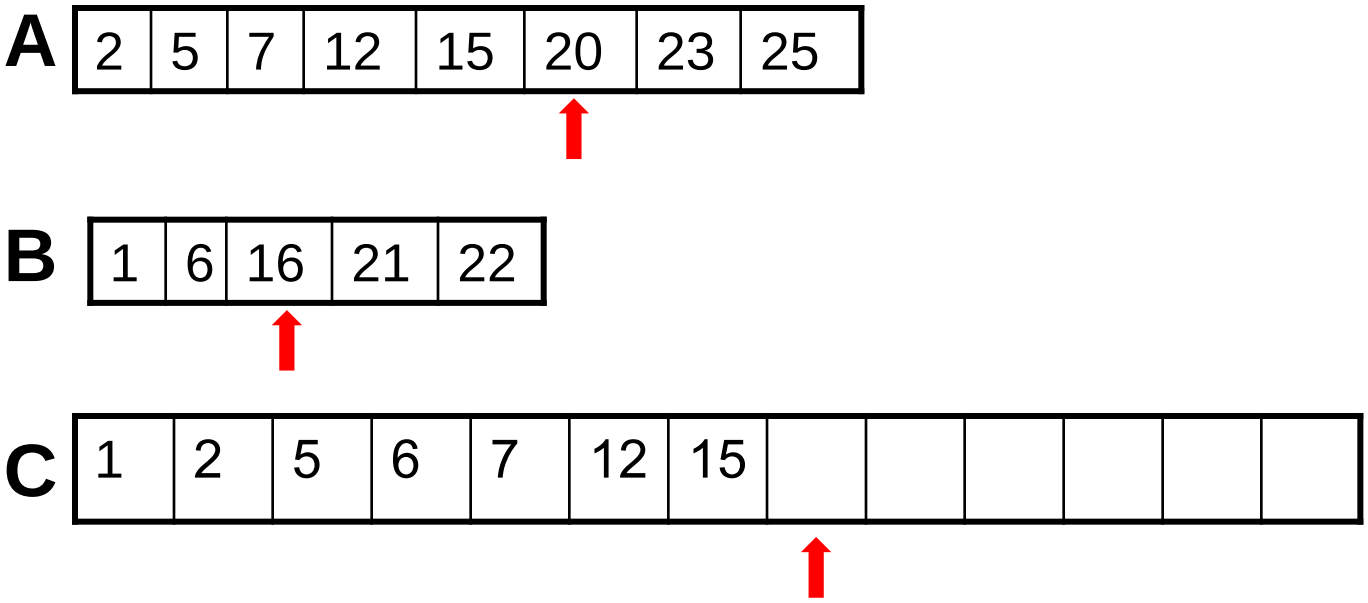

A
2
5
7
12
15
20
23
25
B
1
6
16
21
22
C
1
2

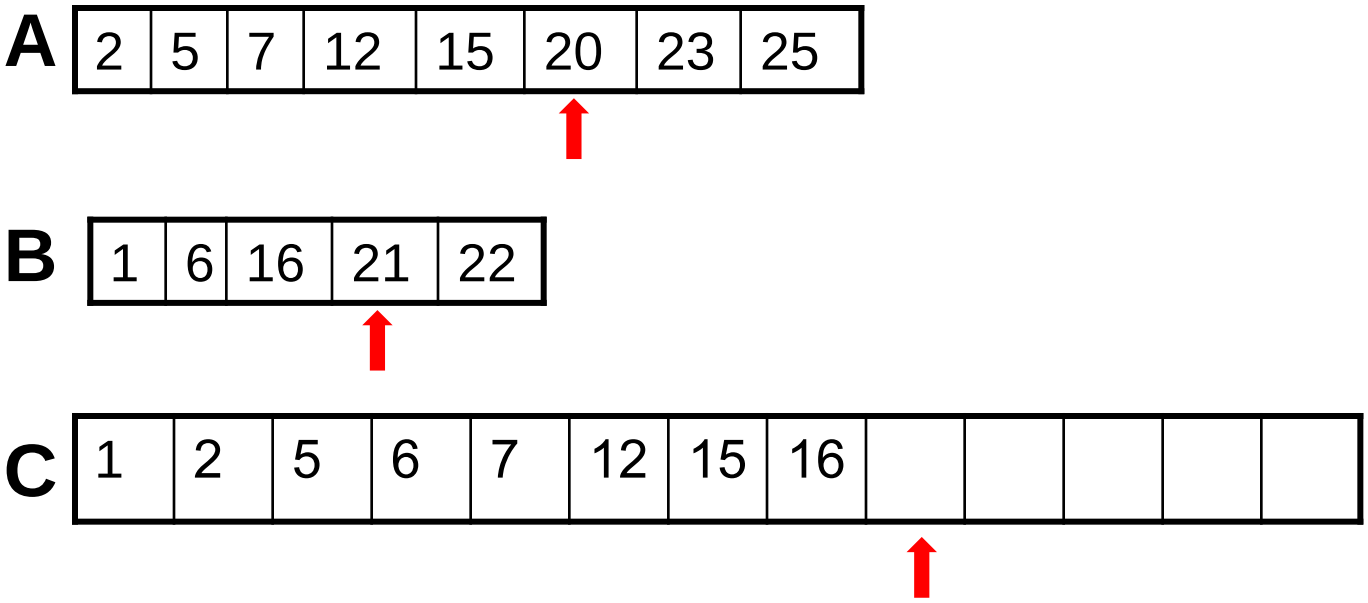

A
2
5
7
12
15
20
23
25
B
1
6
16
21
22
C
1
2
5

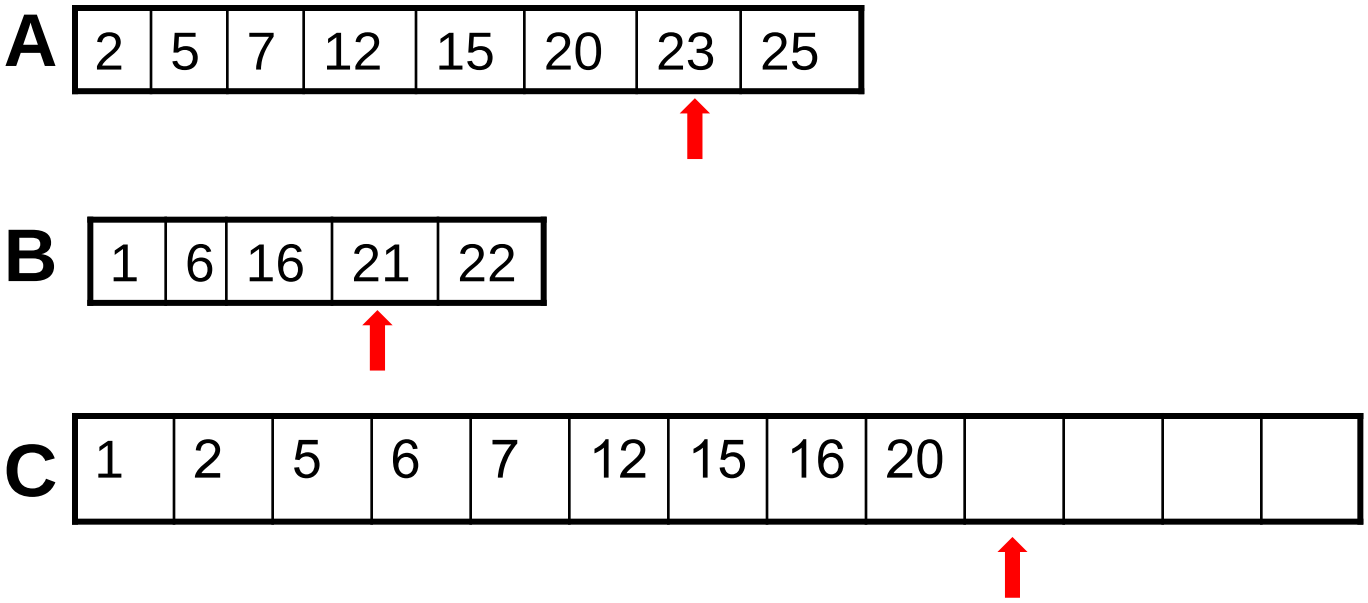

A
2
5
7
12
15
20
23
25
B
1
6
16
21
22
C
1
2
5
6

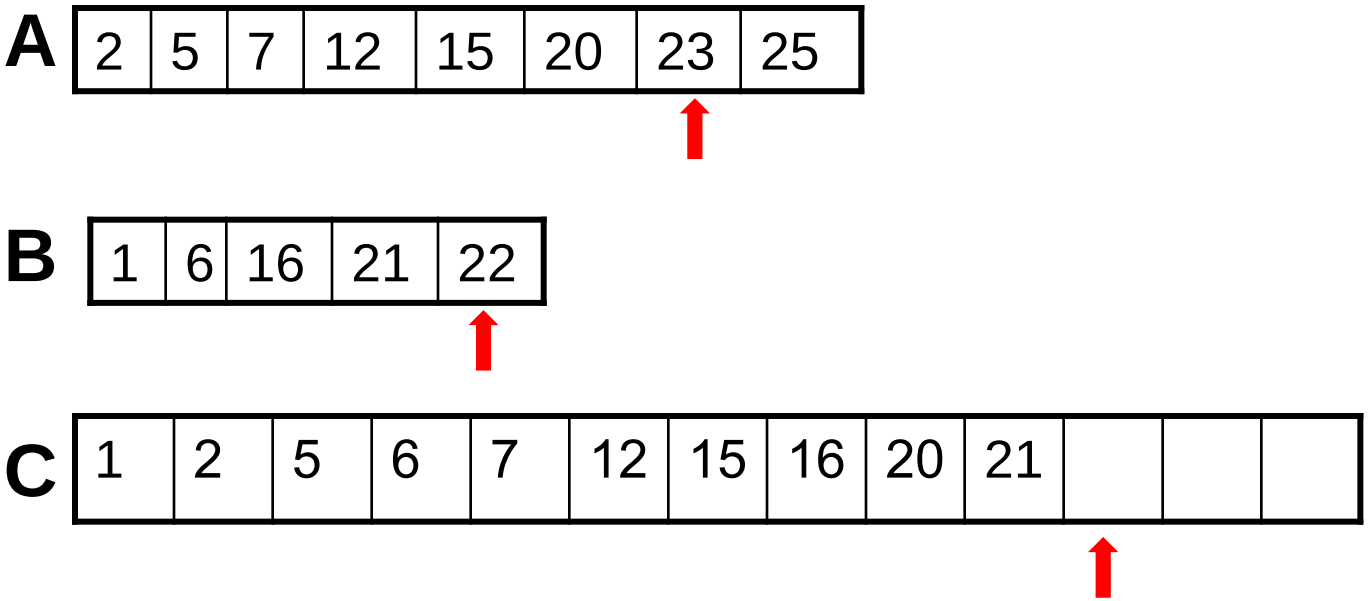

A
2
5
7
12
15
20
23
25
B
1
6
16
21
22
C
1
2
5
6
7

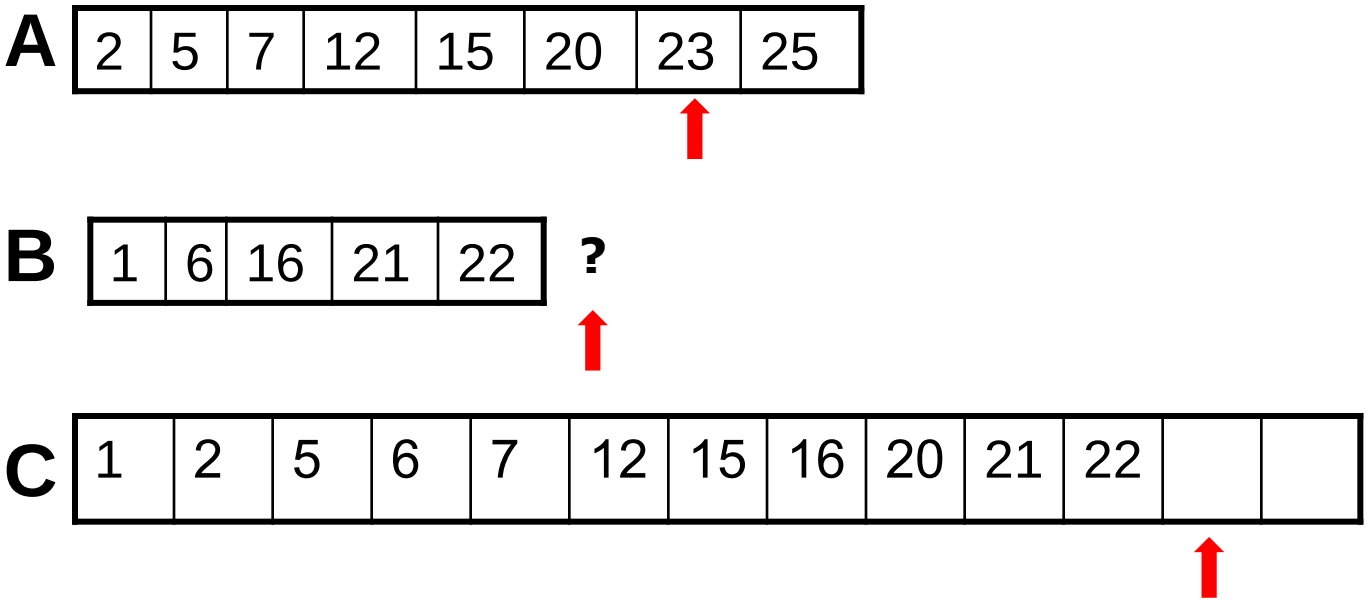

A
2
5
7
12
15
20
23
25
B
1
6
16
21
22
C
1
2
5
6
7
12

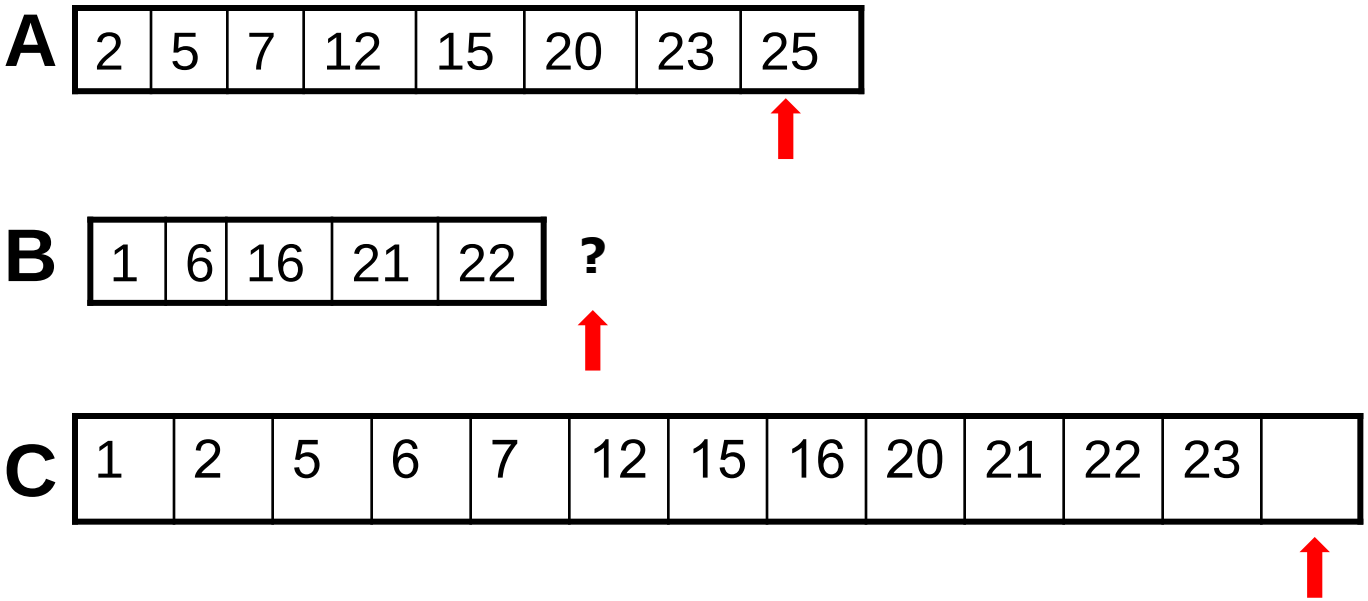

A
2
5
7
12
15
20
23
25
B
1
6
16
21
22
C
1
2
5
6
7
12
15

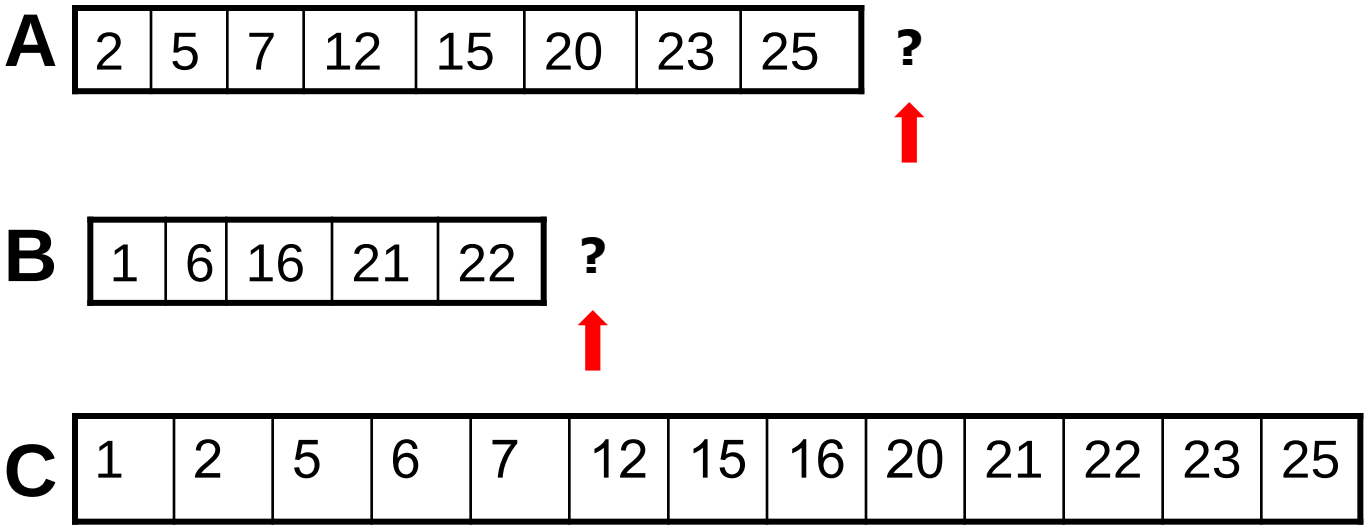

A
2
5
7
12
15
20
23
25
B
1
6
16
21
22
C
1
2
5
6
7
12
15
16


A
2
5
7
12
15
20
23
25
B
1
6
16
21
22
C
1
2
5
6
7
12
15
16
20


A
2 5 7 12 15 20 23 25
B
1 6 16 21 22
C
1 2 5 6 7 12 15 16 20 21


A
2
5
7
12
15
20
23
25
B
1
6
16
21
22
?
C
1
2
5
6
7
12
15
16
20
21
22


A
2
5
7
12
15
20
23
25
B
1
6
16
21
22
?
C
1
2
5
6
7
12
15
16
20
21
22
23


A
2
5
7
12
15
20
23
25
?
B
1
6
16
21
22
?
C
1
2
5
6
7
12
15
16
20
21
22
23
25

# Code

```
def merge(A, B):
    ''' Merge list A of size n and list B of size m
        A and B must be sorted! '''
    n = len(A)
    m = len(B)
    C = ____________________

    a=0; b=0; c=0
    while  a<n  and  b<m: #more element in both A and B
           if A[a] < B[b]:
               C[c] _______________
               a = a+1
           else:
               C[c] = B[b]
               ___________
           c = c+1

    if _____:#A was completed
           while _____________:
               C[c] = B[b]
               b = b+1
               c = c+1
    else:     #B was completed
           _______________:
               ___________
               ___________
               ___________

    return _______________
```

# Code

```
def merge(A, B):
    ''' Merge list A of size n and list B of size m
        A and B must be sorted! '''
    n = len(A)
    m = len(B)
    C = [0 for i in range(n + m)]

    a=0; b=0; c=0
    while  a<n  and  b<m: #more element in both A and B
           if A[a] < B[b]:
               C[c] = A[a]
               a+=1
           else:
               C[c] = B[b]
               b+=1
           c+=1

    if a==n: #A was completed
           while b<m:
               C[c] = B[b]
               b+=1
               c+=1
    else:    #B was completed
           while a<n:
               C[c] = A[a]
               a+=1
               c+=1

    return C
```

```
C[c:] = A[a:]+B[b:] #append remaining elements
```

one of the lists A or B is non-empty

# Merge - analysis

- Again, we will look at the number of iterations.
- So, how many iterations are needed, as a function of the input size?
  - Denote: |A|=n, |B|=m
- Does the answer depend on the content of the lists, or on their length only?
- Compare to merge_by_sort we saw earlier:

```
def merge_by_sort(A,B):
     """ merging two lists """
    C = A + B
    selection_sort(C)
    return C
```

# Merge – Actual Running Times

```
for merge_func in [merge_by_sort, merge]:
  print(merge_func.__name__)
  for n in [1000,2000,4000]:
        lst1 = [random.choice(range(10000)) for i in range(n)]
        lst1 = sorted(lst1)
        lst2 = [random.choice(range(10000)) for i in range(n)]
        lst2 = sorted(lst2)

        t0 = time.clock()    #Stopper go!
        merge_func(lst1,lst2)
        t1 = time.clock()    #Stopper end

        print("n=", n, t1-t0)
```

Note: we chose n=m for simplicity.

```
merge_by_sort
n= 1000 0.23752907143226304
n= 2000 1.019045996069333
n= 4000 3.790595452774026
merge
n= 1000 0.0009043048767365391
n= 2000 0.0017817907024495483
n= 4000 0.0037136004715678794
```

Consistent with the theoretical analysis:
-merge_by_sort is quadratic

- merge is linear

# Merge_by_python_sort

merge_by_sort

n= 1000 0.2375290714322630 4

n= 2000 1.019045996069333

n= 4000 3.790595452774026

merge_by_python_sort

n= 1000 0.0001748825618896 8876

n= 2000 0.0004243556094420 9527

n= 4000 0.0007657397797764531

merge

n= 1000 0.0009043048767365391

n= 2000 0.0017817907024495483

n= 4000 0.0037136004715678794

```
def merge_by_python_sort(A,B):
    return sorted(A+B)
```

Python's sort function

So, python's sorted does a farely nice job!

However we will not get into the interiors of sorted (at least not now).

# And Now to Something Completely Different:
# Lambda Expressions

# Short digression: Expression vs. Statement

- An expression is anything that "has a value". Anything that can be the right hand side of an assignments (e.g., res = … ).
- Expressions are combined to larger expressions by operators

```
>>> 3+4
5
>>> x>y
False
>>> x>y and “A” in “Amir”
False
>>> min([1,2,3,4,5])
1
>>> [x**2 for x in [1,2,3] if x-1 != 0]    #list comprehension
[4,9]
>>> "equal" if x==4 else "not equal“    #if expression
'equal'
```

- A statement is a code segment that performs some action, and does not “have a value”: assignments, loops, conditionals, functions definition.
- Lambda expressions, which we will now see, can include only expressions.

# λ Expressions and λ Calculus

The λ (lambda) calculus was invented by Alonzo Church (1903-1995). Church was one of the great mathematicians and logicians of the twentieth century. Lambda calculus was one of several attempts to capture a mathematical notion of computing, long before actual computers existed.

(photo from Wikipedia)

You will meet him (Church) at least once more in your studies, when you get to the famous Church thesis in the Computational Models course.

# λ Expressions, cont

- In the current context, we will concentrate on λ expressions: These have the form $\lambda\ x_1\ x_2\ \ldots\ x_k$ : expression
- Such a λ expression represents an “anonymous function."
- The arguments are the $x_1\ x_2\ \ldots\ x_k$ ($k \geq 0$)
- The expression on the right is the body of the function, which is to be executed with the actual values supplied upon calling the function.
- This construct does not have any explicit name. Thus it is an “anonymous'' function.
- The function can be applied and executed like any other function.

# λ Expressions: A few Examples

- Given the three coordinates x,y,z in Euclidean 3D space, the length of the vector going from (0,0,0) to (x,y,z) is $\sqrt{x^2+y^2+z^2}$ .

- In Python, this can be defined as following:

```
def euclid(x,y,z):
    return (x**2 + y**2 + z**2)**0.5

>>> type(euclid)
<class 'function'>
>>> euclid
<function euclid at 0x00000000037BF158>
>>> hex(id(euclid))
'0x37bf158'
>>> euclid(2,3,4)
5.385164807134504
```

$$\sqrt{2^2+3^2+4^2} = \sqrt{29} \approx 5.38$$

- So `euclid` is an object of type function, placed in the computer's memory just like any other object.

# λ Expressions: A few Examples

- So we have:

```
def euclid(x,y,z):
    return (x**2 + y**2 + z**2)**0.5
```

- An alternative way to define this function would be:

```
>>> euclid2 = lambda x,y,z: (x**2 + y**2 + z**2)**0.5

>>> type(euclid2)
<class 'function'>
>>> type(lambda x,y,z: (x**2 + y**2 + z**2)**0.5)
<class 'function'>
>>> lambda x,y,z: (x**2 + y**2 + z**2)**0.5
<function <lambda> at 0x170b150>
```

- The Python Shell  informed that this is a function.

# λ Expressions: A few Examples

```
>>> euclid2 = lambda x,y,z: (x**2 + y**2 + z**2)**0.5
```

- This function can subsequently be applied:

```
>>> euclid2(2,3,4)
5.385164807134504
```

- We can also use a function without a name (notice the extra parenthesis):

```
>>> (lambda x,y,z: (x**2 + y**2 + z**2)**0.5)(2 ,3 ,4)
5.385164807134504
```

# Why Should We Care ?

We saw we could define

```
>>> euclid = lambda x,y,z: (x**2 + y**2 + z**2)**0.5
```

and invoke it by calling, e.g.

```
>>> euclid(2,3,4)
```

But then

```
def euclid(x,y,z):
    return (x**2 + y**2 + z**2)**0.5
```

does exactly the same.

- So what are these weird, cumbersome λ expression good for in our context?

  1) A nice way to define short, simple functions

  2) A way to define anonymous functions, usually in higher order functions, which we will now see.

- We will see examples that show the usefulness of λ expressions.

# The Derivative as a "High Order Operator"

As you surely recall, the derivative of $f$ at point $x$ is defined by

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

- We want to define a Python function diff that receives a function, $f$, as its input argument and produces the function $f'$ (the derivative of $f$) as its returned value.
- We cannot compute the limit as $h \to 0$, so we compute the value at some small $h$.
- So, this diff is a "high order function'': if uses another function as input and/or output. However, for Python this is nothing unusual.

```
def diff(f):
    h=0.001
    return (lambda x: (f(x+h)-f(x))/h)
```

# The Derivative as a “High Order Operator”

On input f , a real value function, diff(f) returns another Python function, which is (a numeric approximation to) the derivative of f.

```
>>> diff(lambda x: x**3)    # derivative of f(x)=x**3
<function <lambda > at 0x300d978 >  # outcome of diff

>>> diff(lambda x: x**3)(5) # apply the resulting
75.01500100002545                  function on 5
```

# Highly Variable Functions

So far, we assumed that h=0.001 is small enough for our needs. This may be OK in most cases, but for highly variable functions, the outcome may be very inaccurate.

As a specific (and somewhat artificial) example, consider the function sin_by_million (x) = $\sin(10^6 \cdot x)$, its derivative is $10^6 \cdot \cos(10^6 \cdot x)$ so at point x=0 its value is $10^6 \cdot \cos(0) = 10^6$, so diff(sin_by_million)(0) should be approximately $10^6$

```
def sin_by_million(x):
     return math.sin(10**6*x)
>>> diff(sin_by_million)(0)
826.8795405320026 #???
```

# Implementation and Default Parameters' Values

826.8795405320026 is, ahhm, not even close to $10^6$=1,000,000. The reason for this discrepancy is that h=0.001 is usually small enough, but it is way too big for sin_by_million.

We already saw that Python provides a mechanism of default values for parameters.

This let us use a predetermined value as a default parameter, yet use different values when we deem it necessary. We recommend you explicitly specify the original parameter name when setting such a different value (even though in some cases, for example with a single default parameter, this is not required by Python)

```
def diff_param(f, h=0.001):
    #when h not specified, default h=0.001 is used
    return (lambda x: (f(x+h)-f(x))/h)
```

# Short digression: Optional and Named Parameters in Python

- Optional parameter: a function can define default value for a parameter, which is used when a call omits this parameter .
- Arguments may be passed in any order by using named parameters. They can be combined with positional parameters - positional parameters first.

```
def f(a, b=1, c=2):
    return a + 2*b + 3*c
```

```
>>> f(4 ,5 ,6)
32
>>> f(4 ,5)
20
>>> f(4)
12
```

```
>>> f(4, b=5)
20
>>> f(4, c=3)
15
>>> f(c=3, a=4)
15
>>> f(b=5, a=4, c =6)
32
```

# Generalized Implementation and Default Parameters

```
def diff_param(f, h=0.001):
    #when h not specified, default h=0.001 is used
    return (lambda x: (f(x+h)-f(x))/h)
```

We can now apply this mechanism with different degrees of resolution.

```
>>> diff_param(sin_by_million)(0)
826.8795405320026  # no h specified - default h used
>>> diff_param(sin_by_million ,h =0.001)(0)
826.8795405320026  # parameter equals the default h
>>> diff_param(sin_by_million ,h =0.00001)(0)
-54402.11108893698 # smaller and smaller h
>>> diff_param(sin_by_million ,h =0.0000001)(0)
998334.1664682814  # better and better accuracy for derivative
>>> diff_param(sin_by_million ,h =0.000000001)(0)
999999.8333333416
>>> diff_param(sin_by_million ,h =0.000000000001)(0)
999999.9999998333  # indeed almost 1000000 , as expected
```

Recall that real numbers (type float in Python) have a limit on accuracy. A value that is too small will be interpreted as zero.