Python for Data Scientists
L7 : Program efficiency

Efficiency of programs (Poll)

Since computer are fast and even getting more faster, do you think efficient programs does matter?

A	B
Yes	No

Efficiency of programs

- A program can be implemented in many different ways

- you can solve a problem using only a handful of different algorithms

- would like to separate choices of implementation from choices of more abstract algorithm

How to evaluate efficiency of programs

- measure with a timer
- count the operations
- abstract notation of order of growth

How to evaluate efficiency of programs: measure with a **timer**

use the time module :

```
import  time

def f1(x):
    return x*x/54

t0 = time.perf_counter()
f1(50000000000)
t1 = time.perf_counter()

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

t0= 0.0501374 , t1= 0.0501609

How to evaluate efficiency of programs: measure with a **timer**

- running time **varies between algorithms**
- running time **varies between implementations**
- running time **varies between computers**
- running time is **not predictable** based on small inputs

time varies for different inputs but cannot really express a relationship between inputs and time

How to evaluate efficiency of programs: **count** the operations

```
def f1(x):
    return x*x/54

def f2(x):
    sum = 0
    for i in range(x+1):
        sum += i
    return sum
```

f1(x) → 2 operations

f2(x) → 1 + 4x operations

- 1 assignment
- 4x mathematical operations

How to evaluate efficiency of programs: **count** the operations

- count **depends on algorithm**
- count **depends on implementations**
- count **independent of computers**
- no clear definition of **which operations** to count

count varies for different inputs

How to evaluate efficiency of programs:

- timing and counting evaluate implementations

- timing evaluates machines

→How to **evaluate algorithm**

→How to **evaluate scalability**

→How to **evaluate in terms of input size**

Efficiency in terms of size of input

Example: Function that searches for an element in a list

```
def search_for_elmt(L, e):
    for i in L:
        if i == e:
            return True
    return False
```

- when e is **first element** in the list : BEST CASE
- when e is **not in list :** WORST CASE
- when **look through about half** of the elements in list : AVERAGE CASE

BEST, AVERAGE, WORST CASES

- **best case**: minimum running time over all possible inputs of a given size
- **average case**: average running time over all possible inputs of a given size
- **worst case**: maximum running time over all possible inputs

How to evaluate efficiency of programs: Orders of growth

Aims:

- Evaluate program's efficiency when **input is very big**
- Express the **growth of program's run time** as input size grows
- Put an **upper bound** on growth – as tight as possible
- No need to be precise: **"order of" not "exact"** growth
- Look at **largest factors** in run time (which section of the program will take the longest to run?)

→ tight upper bound on growth, as function of size of input, in worst case

Measuring order of growth: Big OH notation

Big Oh notation measures an **upper bound on the asymptotic growth** : called order of growth

Measuring order of growth: Big OH notation

- **Big Oh or *O()*** is used to describe worst case
  - worst case occurs often and is the bottleneck when a program runs
  - express rate of growth of program relative to the input size
  - evaluate algorithm **NOT** machine or implementation

Big OH notation : O()

```python
def fact_iter(n):
    answer = 1
    while n > 1:
        answer *= n
        n -= 1
    return answer
```

Answer = 1 → 1 operation

While → n(1 + 2 + 2) op

Return → 1 op

number of steps: 2 + 5n

worst case asymptotic complexity:

- ignore additive constants
- ignore mulAplicative constants

→ O(n)

Big OH notation

- Given an expression for the number of operations needed to compute an algorithm, want to know asymptotic behavior as size of problem gets large

- Focus on term that grows most rapidly in a sum of terms

- Ignore multiplicative constants, since want to know how rapidly time required increases as increase size of input

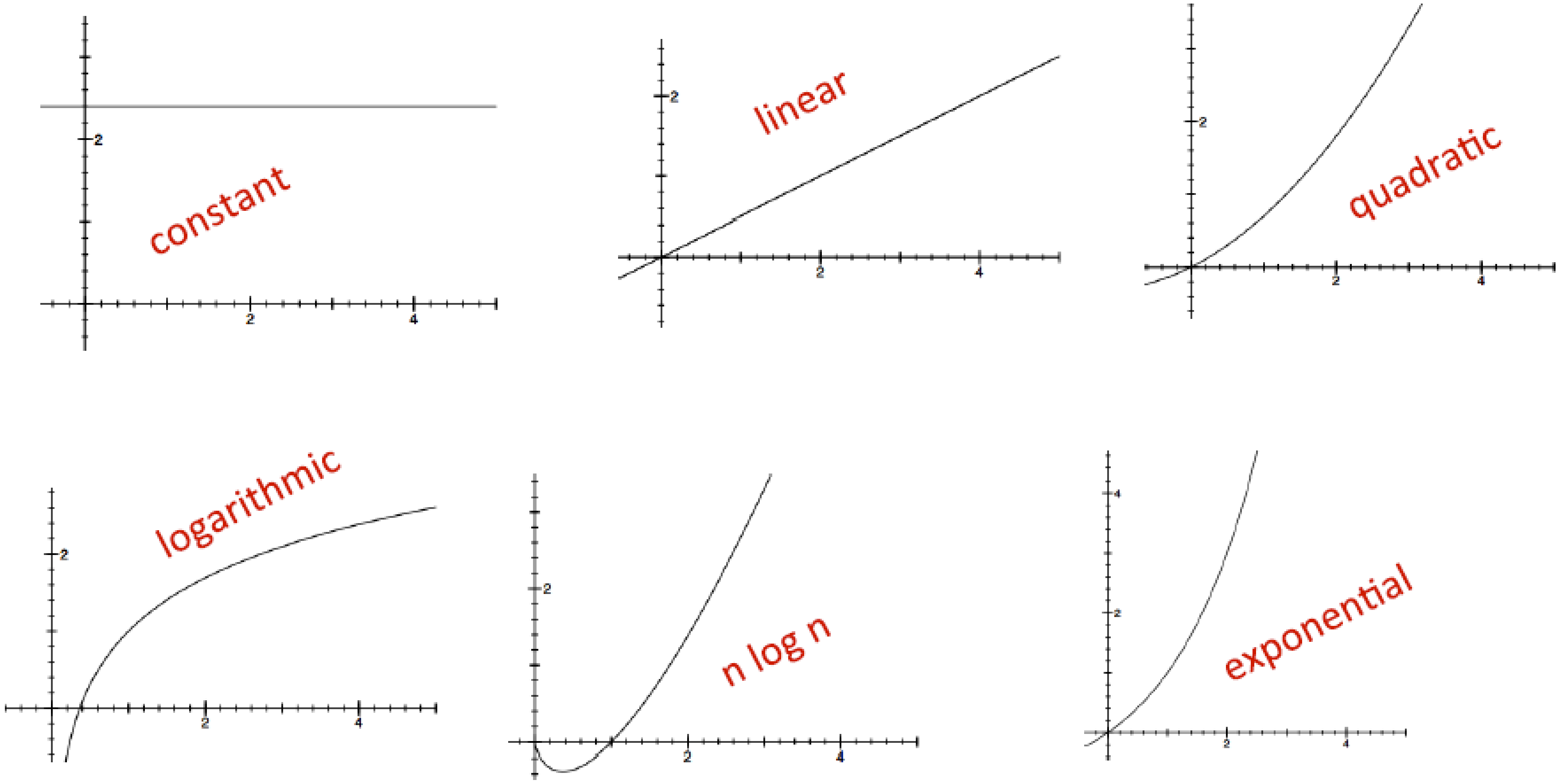
Simplification examples

- drop constants and multiplicative factors
- focus on **dominant terms**

Examples:

$n^2 + 2n + 2$	$O(n^2)$
$n^2 + 100000n + 3^{1000}$	$O(n^2)$
$\log(n) + n + 4$	$O(n)$
$0.0001*n*\log(n) + 300n$	$O(n*\log(n))$
$2n^{30} + 3^n$	$O(3^n)$

Types of orders of growth



Analyze programs and their complexity (Poll)

Addition for O():

- used with **sequential** statements
- O(f(n)) + O(g(n)) is O(f(n) + g(n))

```
for i in range(n):
    print('a')

for j in range(n*n):
    print('b')
```

O(n) + O(n*n) = O(n+n^2) = O(n^2)

Analyze programs and their complexity (Poll)

multiplication for O():

- used with **nested** statements/loops
- O(f(n)) * O(g(n)) is O(f(n) * g(n))

```
for i in range(n):
    for j in range(n*n):
        print('b')
```

O(n)*O(n^2) = O(n*n^2) = O(n^3)

Complexity classes

- *O(1)*
- *O(log n)*
- *O(n)*
- *O(n log n)*
- *O(n^c)*
- *O(c^n)*

Constant complexity

- *O(1)* denotes constant running time

- complexity independent of inputs

Logarithmic complexity

- *O(log n)* denotes logarithmic running time
- complexity grows as log of size of one of its inputs
- example: (next lecture)
  - bisection search
  - binary search of a list

Linear complexity

- *O(n)* denotes linear running time

- Examples:
    - searching a list in sequence to see if an element is present
    - iterative loops

Log-Linear complexity

- *O(n log n)* denotes log-linear running time

- many practical algorithms are log-linear and very commonly used log-linear algorithm is merge sort (next lecture)

Polynomial complexity

- *O(n^c)* denotes polynomial running time (c is a constant)
- most common polynomial algorithms are quadratic, i.e., complexity grows with square of size of input
- commonly occurs when we have nested loops or recursive function calls

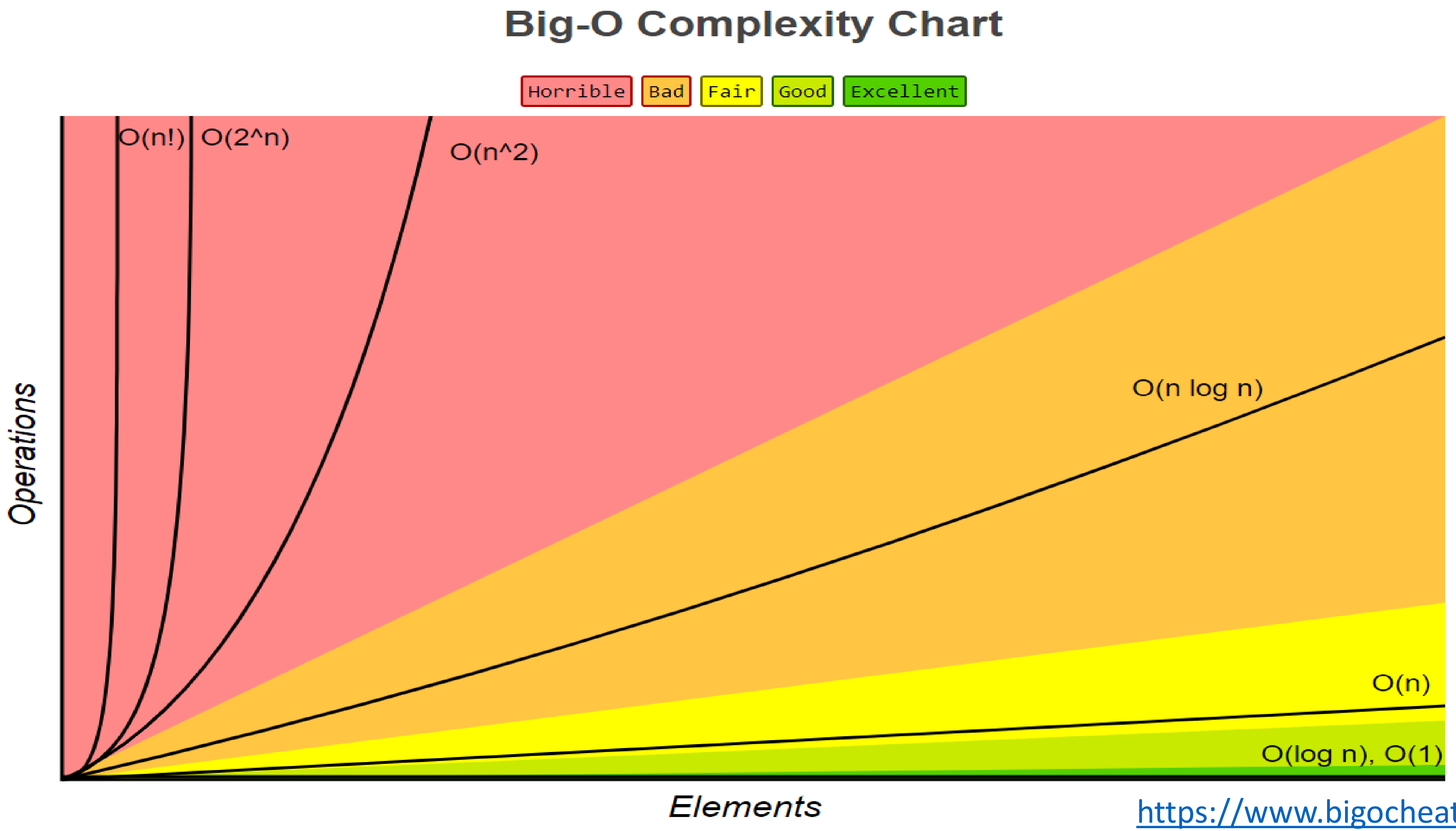
Exponential complexity

- *O(c^n)* denotes exponential running rime (c is a constant being raised to a power based on size of input)

- recursive functions where more than one recursive call for each size of problem

- many important problems are inherently exponential
  - unfortunate, as cost can be high
  - will lead us to consider approximate solutions as may provide reasonable answer more quickly

Complexity classes: ordered low to high



https://www.bigocheatsheet.com/

Complexity growth

CLASS	n=10	= 100	= 1000	= 1000000
O(1)	1	1	1	1
O(log n)	1	2	3	6
O(n)	10	100	1000	1000000
O(n log n)	10	200	3000	6000000
O(n^2)	100	10000	1000000	1000000000000
O(2^n)	1024	12676506 00228229 40149670 3205376	10715086071862673209484250490600 01810561404811705533607443750388 37035105112493612249319837881569 58581275946729175531468251871452 85692314043598457757469857480393 45677748242309854210746050623711 41877954182153046474983581941267 39876755916554394607706291457119 64776865421676604298316526243868 37205668069376	Good luck!!

Recap: Complexity classes

- *O(1)* : code does not depend on size of problem
- *O(log n)* : reduce problem in half each time through process
- *O(n)* : simple iterative or recursive programs
- *O(n log n)* : log-linear running time
- *O(n^c) :*nested loops or recursive calls
- *O(c^n)* : multiple recursive calls at each level

Examples (Poll)

What is the best and the worst cases for this example?

```
def fib_iter(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        a = 0
        b = 1
        for i in range(n-1):
            tmp = a
            a = b
            b = tmp + b
        return b
```

O(1)

O(1)

O(n)

O(1)

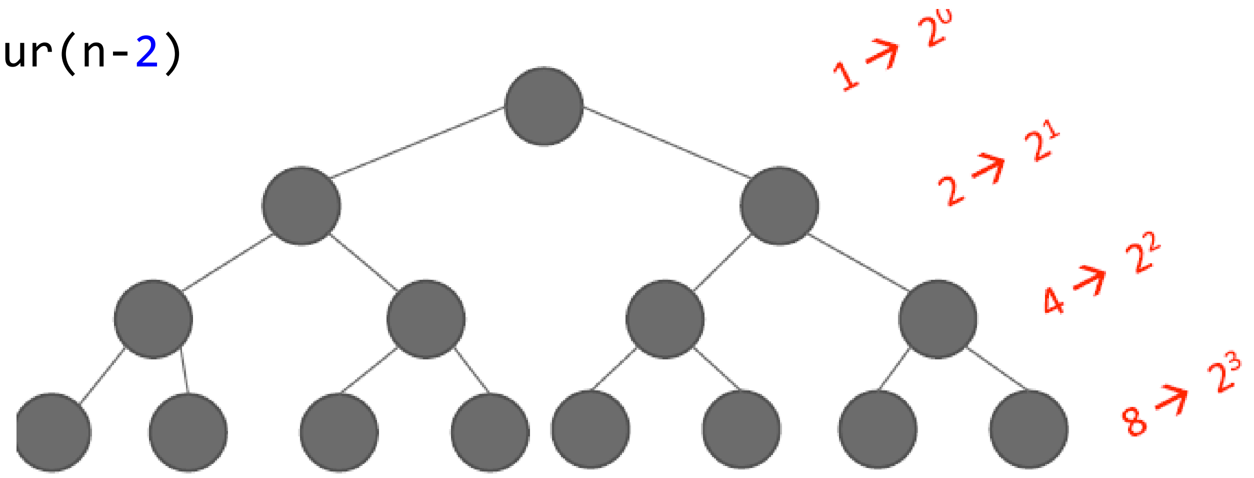
A- O(n)   O(1)
B- O(1)   O(n)
C- None of the above

Examples (Poll)

What is the worst cases for this example?

```python
def fib_recur(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib_recur(n-1) + fib_recur(n-2)
```

A- O(n)
B- O(n^2)
C- O(2^n)

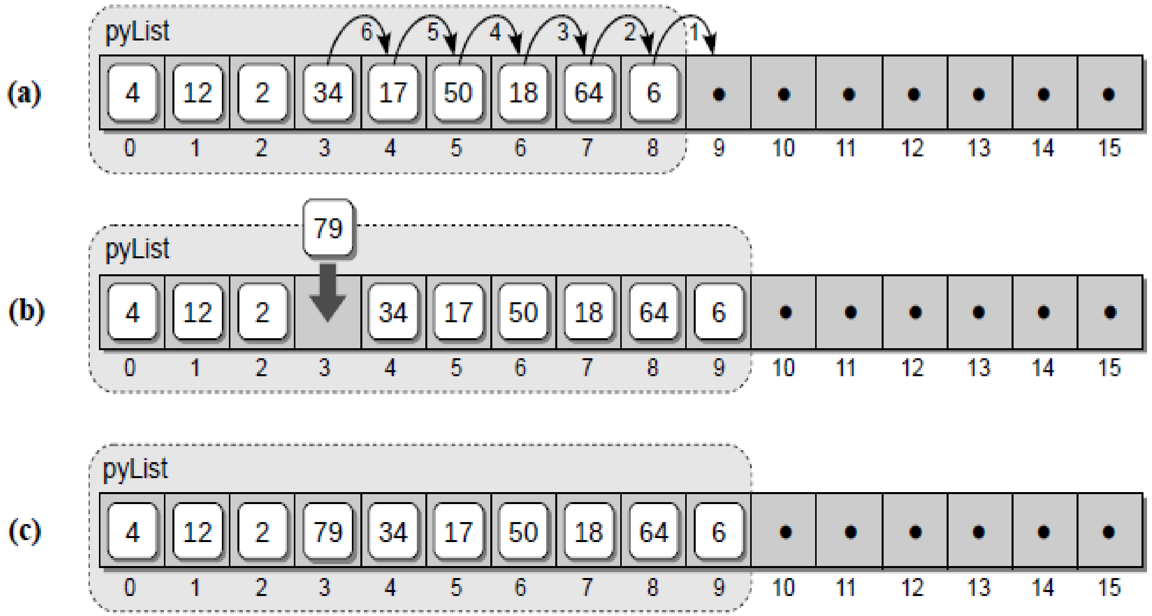


Big-O of Algorithms and lists (Poll)

Expensive Python list operations

- insert(i, x)

→ O(n)



Big-O of Algorithms and lists (Poll)

Expensive Python list operations

- remove(i)

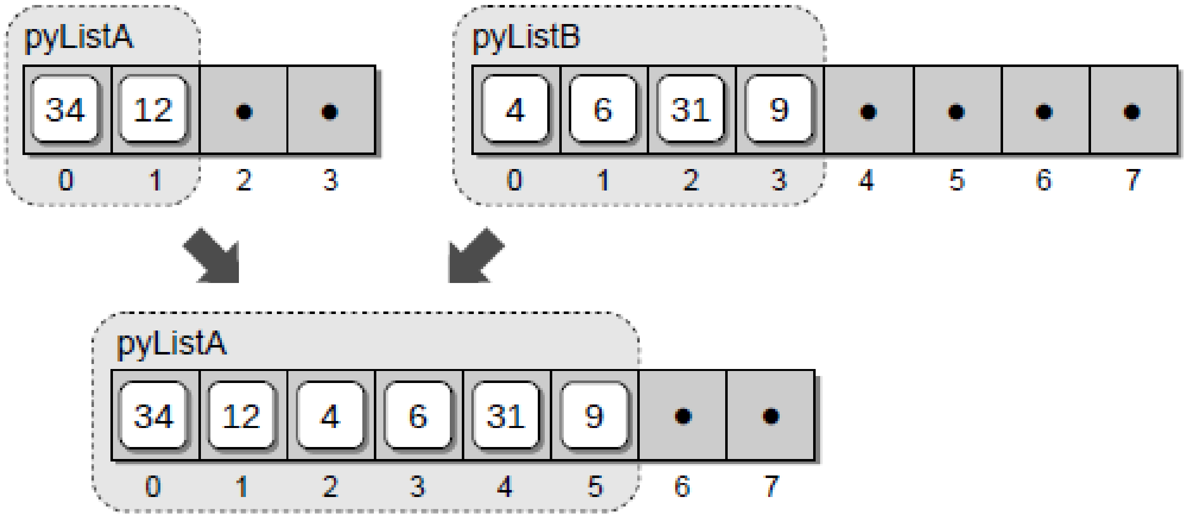
→ O(n)

Big-O of Algorithms and lists (Poll)

Expensive Python list operations

- extend()

  → O(k)



Time complexity of Python Data structure

Operation	Worst case
Copy	O(n)
Append	O(1)
Pop last	O(1)
Remove	O(n)
Get item	O(1)
Set item	O(1)
Iteration	O(n)

Time complexity of Python Data structure

Operation	Worst case
Multiply	O(nk)
X in L	O(n)
Get length	O(1)
Sort	O(nlogn)

Time complexity of Python Data structure

What about the other Data structures?

(Assignment 4)

Summary

- Compare **efficiency of algorithms**
  - notation that describes growth
  - **lower order of growth** is better
  - independent of machine or specific implementation

- Use Big Oh
  - describe order of growth
  - **upper bound**
  - **worst case** analysis