

1.3

Complexity

Lesson Plan

- [10] Ice Breaker
- [20] Intro to Complexity and Formal Definition
- [10] Big-O for Algorithms
- [15] Complexity of List and String operations
- [10] Space complexity
- [30] Complexity Practice

Icebreaker

List Methods

Adding Elements to a List

```
lst.append(x)
lst.insert(i, x)
lst.extend(lst2)
lst += lst2
```

Removing Elements from a List

```
lst.remove(x)
lst.pop()
lst.pop(i)
lst.clear()
```

Modifying a List

```
lst.sort()  
lst.reverse()
```

Checking Elements of a List

```
lst.index(x)  
lst.count(x)  
min(lst)  
max(lst)
```

Initializing a List

```
lst = []  
lst = [0, 1, 2, 3]  
lst = [0] * n
```

List Comprehensions

List Comprehensions - great for building lists!

```
def square(nums):  
    squares = []  
    for num in nums:  
        squares.append(num**2)  
    return squares
```

```
def square(nums):  
    return [num**2 for num in nums]
```

Generalized List Comprehension

```
[value for element in collection if condition]
```

Generalized List Comprehension Example

```
def is_valid_email_address(address):  
    return address.endswith('techexchange.in') and \  
        len(address) > len('@techexchange.in') and \  
        '/' not in address  
  
emails = ['DANIEL@techexchange.in', 'daniel@google.com', 'mikayla@techexchange.in']  
valid_emails = [ ] for email in emails [ ]
```

Tuples & Unpacking

Unpacking Tuples

circle = (2, 3, 4)

You can access elements in a tuple like you would in a list...

... but it's often clearer to unpack a tuple this way

Use an underscore when you don't care about the value!

x = circle[0]
y = circle[1]
r = circle[2]

x, y, r = circle

_, _, r = circle
area = math.pi * r**2

Iterating through a list of Tuples

```
people = [('Mikayla', 'Scorpio'), ('Patrick', 'Leo')]  
  
for person, sign in people:  
    print(person + ' is a ' + sign)
```

```
Mikayla is a Scorpio  
Patrick is a Leo
```

We've seen tuples before...

```
goats = ['serena', 'lionel', 'michael']  
for i, goat in enumerate(goats):  
    print(i, goat)
```

```
(0, 'serena')  
(1, 'lionel')  
(2, 'michael')
```

Tuples are Immutable!

```
point = (3, 4)  
point[0] = 1
```

```
Traceback (most recent call last):  
  File "main.py", line 5, in <module>  
    point[0] = 1  
TypeError: 'tuple' object does not support item assignment
```

Intro to Complexity

Time Complexity

The **amount of time** it takes to run a program

in terms of the **size of the input** to the program

Example

```
def sum(lst):
    sum = 0
    for item in lst:
        sum += item
    return sum
```

In this example, the **size of the input** is the length of the list, which we'll call **n**

The **amount of time** it takes to run the program depends on how big **n** is

Which of these takes longer to run?

- `sum([1, 2, 3, 4, 5, 6])`
- `sum([1, 2, 3])`

“**size of the input**” depends on the problem and may involve more than one variable

Time Complexity

Count “elementary operations”:

- Number of elementary operations is **n + 1**
- If every operation took 1 second, this program would take **n + 1** seconds to run
- We're only interested in the **dominant term**: **n**
 - For large input sizes **n**, the extra +1 doesn't make much of a difference

```
def sum(lst):
    sum = 0                # 1 operation
    for item in lst:        # loops n == len(lst) times
        sum += item        # 1 operation
    return sum
```

Practice: Count the Elementary Operations

Chat waterfall:

How many “elementary operations” are in this program **in terms of n**, where **n == len(lst)**?

```
def sum(lst):
    sum = 0
    i = 0
    while i < len(lst):
        sum += lst[i]
        i += 1
    return sum
```

Practice: Count the Elementary Operations

There are **2n + 2** elementary operations.

For large enough **n**, it's close enough to say there are about **n** operations.

```
def sum(lst):
    sum = 0                # 1 operation
    i = 0                  # 1 operation
    while i < len(lst):    # loops n == len(lst) times
        sum += lst[i]      # 1 operation
        i += 1             # 1 operation
    return sum
```

Worst-Case Analysis

What's a word where this function runs quickly?

What is a word which would be the worst case for this algorithm?

```
def contains_a(word):  
    for letter in word:  
        if letter == 'a':  
            return True  
    return False
```

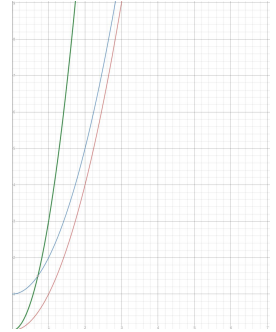
Big-O

Big-O expresses the notion of:

"has the same growth as"
(as n gets large)

$n^2 + 1$ is order n^2
 $n^2 + 1$ is $O(n^2)$

$3n^2$ is order n^2
 $3n^2$ is $O(n^2)$



Formal Definition

Formal Definition

A function $f(n)$

is said to be $O(g(n))$ as $n \rightarrow \infty$ if

there exists some constants $M > 0$ and c such that

$$|f(n)| \leq M * g(n) \text{ for all } n \geq c$$

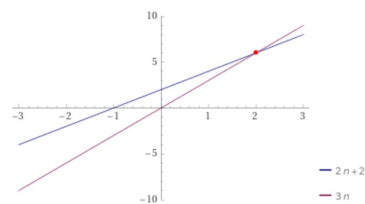
Formal Definition: Example

The function $2n + 2$

is said to be $O(n)$ as $n \rightarrow \infty$ because

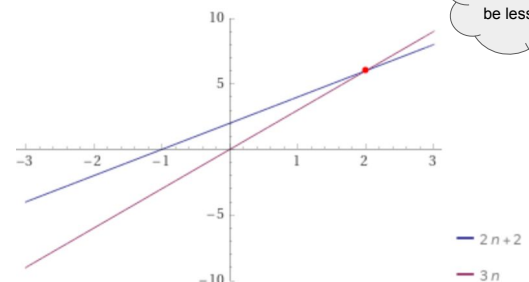
there exists constants 3 and 2 such that

$$|2n + 2| \leq 3 * n \text{ for all } n \geq 2$$



Formal Definition: Example

$$2n + 2 \leq 3n \text{ for all } n \geq 2$$



After their paths cross at $n = 2$, $2n+2$ will always be less than $3n$

Big-O for Algorithms

Big-O for Algorithm Time Complexity

An **algorithm** is $O(g(n))$
if in the **worst case**
the number of **elementary operations**
as a function of the **input size, n**
is $O(g(n))$

- Examples:
- "The sum function is $O(n)$ "
 - "Binary search is $O(\log n)$ "
 - "Insertion sort is $O(n^2)$ "
 - "Merge sort is $O(n \log n)$ "

Sometimes we call this out explicitly:

- "Quicksort is on average $O(n \log n)$ but worst case $O(n^2)$ "

Working with Big-O

In practice, use following rules

- If $f(x)$ is a sum of several terms, if there is one with largest growth rate, it can be kept, and all others omitted.

$n^2 + n + 1$ is $O(n^2)$

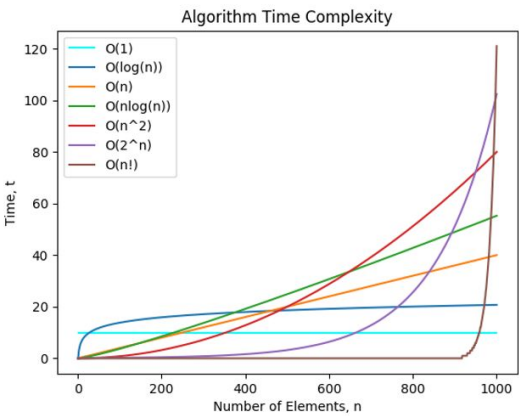
- If $f(x)$ is a product of several factors, any constants (terms in the product that do not depend on x) can be omitted.

$3n$ is $O(n)$

$n/2 = 0.5 \cdot n$ is $O(n)$

Common Time Complexities (best to worst)

Name	Running Time	Example Algorithms
constant time	$O(1)$	Hashtable lookup, insert, delete; Linked list insert, delete
logarithmic time	$O(\log n)$	Binary search; Balanced binary tree lookup, insert, delete
linear time	$O(n)$	Linear search; Linked list search
n-log-n time	$O(n \log n)$	Merge sort; Best possible comparison sort
quadratic time	$O(n^2)$	Bubble sort
exponential time	$O(2^n)$	Solving the traveling salesman problem using dynamic programming
factorial time	$O(n!)$	Solving the traveling salesman problem via brute-force search



Practice

What is the big-O order for the following functions?
Respond to the poll/chatstorm

1. $5n + 5$
2. $1000n + 1000000$
3. $3n^2 + 2n + 5$
4. $n + \log n + 1$
5. $n \log n + n^2 + n + 2^n$

Example: For vs. While

```
def sum1(lst):
    sum = 0
    i = 0
    while i < len(lst):
        sum += lst[i]
        i += 1
    return sum
```

Length n loop dominates the cost: $O(n)$

```
def sum2(lst):
    sum = 0
    for item in lst:
        sum += item
    return sum
```

Length n loop still dominates: $O(n)$

Example: Nested vs. consecutive loops

```
def f1(lst):
    i = 0
    while i < len(lst):
        j = 0
        while j < len(lst):
            print(lst[i], lst[j])
            j += 1
        i += 1
```

$1 + n * (2n + 2) = 2n^2 + 2n + 1$ operations

f1 is $O(n^2)$

```
def f2(lst):
    i = 0
    while i < len(lst):
        print(lst[i])
        i += 1
    j = 0
    while j < len(lst):
        print(lst[j])
        j += 1
```

$1 + 2n + 1 + 2n = 4n + 2$ operations

f2 is $O(n)$

Example: Don't just count nested loops

```
def f1(lst):
    i = 0
    while i < len(lst):
        j = 0
        while j < len(lst):
            print(lst[i], lst[j])
            j += 1
        i += 1
```

$1 + n * (2n + 2) = 2n^2 + 2n + 1$ operations

f1 is $O(n^2)$

```
def f1(lst):
    i = 0
    while i < len(lst):
        j = 0
        while j < 10:
            print(lst[i], lst[j])
            j += 1
        i += 1
```

$1 + n * (2 * 10 + 2) = 22n + 1$ operations

f2 is $O(n)$

Example: More complex loops

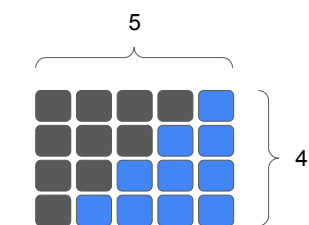
```
def f(lst):
    i = 0
    while i < len(lst):
        j = i
        while j < len(lst):
            print(lst[i], lst[j])
            j += 1
        i += 1
```

```
len(lst) == 4
i = 0:
    j = 0, 1, 2, 3
i = 1:
    j = 1, 2, 3
i = 2:
    j = 2, 3
i = 3:
    j = 3
```

Outer loop: n times

Inner loop: $1 + 2 + \dots + n$ times = $n(n+1)/2$

$2n(n+1)/2 + 2n + 1$ operations
f is $O(n^2)$



$$1 + 2 + \dots + n = \frac{n(n+1)}{2} = \frac{(n^2 + n)}{2} = \frac{1}{2}n^2 + \frac{1}{2}n = O(n^2)$$

$$1 + 2 + 3 + 4 = \frac{4(4+1)}{2} = \frac{4(5)}{2} = \frac{20}{2} = 10$$

Functions and Built-in operations

You also need to account for the complexity of built-in operations and other functions you define






For example:

`b = max(a)`

likely has complexity $O(n)$

Time Complexity of List Operations

Time Complexity of List operations

$O(1)$ $O(N)$ $O(N^2)$ $O(N^3)$ $O(N^4)$

Function	Complexity, where n is len(lst)
lst[i]	?
len(lst)	?
x in lst	?
max(lst), min(lst)	?
sum(lst)	?
for x in lst:	?
doubles = [2*x for x in lst]	?
lst[:]	?

Quick (and Incomplete) Aside

How is Python's list implemented?

Breakout: Complexity of List operations

<https://docs.python.org/3/tutorial/datastructures.html#more-on-lists>

In breakout rooms, find these list functions in the Python documentation. Try to guess the time complexity of each function **in terms of the length of the list, n**.

- **Group 1:** append, extend**, insert
- **Group 2:** remove, pop, pop(i)
- **Group 3:** clear*, index, count
- **Group 4:** sort*, reverse, copy

*These ones are a little tricky - you may need to look these up

**You might need to express this one in terms of something else

Complexity of List operations

Function	Worst Case Complexity, where n is len(lst)
append(x)	?
extend(iterable)	?
insert(i, x)	?
remove(x)	?
pop()	?
pop(i)	?
clear()	?

Complexity of List operations (continued)

Function	Worst Case Complexity, where n is len(lst)
index(x)	?
count(x)	?
lst.sort(), sorted(lst)	?
reverse()	?
copy()	?

Which one is better?

Type Left or Right in the chat

```
def reverse(lst):
    new_lst = []
    for item in lst:
        new_lst.insert(0, item)
    return new_lst
```

```
def reverse(lst):
    new_lst = []
    for i in range(len(lst)-1, -1, -1):
        new_lst.append(lst[i])
    return new_lst
```

Which one is better?

`new_lst.insert(0, item)` has to scoot all the items in the list to the right by one to put the new item at the front.
The first iteration there is 1 scoot, then 2 scoots, then 3 scoots, ..., then n scoots.
This is the triangle sum (!!!) which is $O(n^2)$.

```
def reverse(lst):
    new_lst = []
    for item in lst:
        new_lst.insert(0, item)
    return new_lst
```

$O(n^2)$

```
def reverse(lst):
    new_lst = []
    for i in range(len(lst)-1, -1, -1):
        new_lst.append(lst[i])
    return new_lst
```

$O(n)$ is better!

Example

Let's look at the last iteration of the loop for each solution using the list:
['daikon', 'carrot', 'beets', 'apple']

The last (worst case) insert

```
fridge = ['beets', 'carrot', 'daikon']
fridge.insert(0, 'apple')
```

'beets'	'beets'	'carrot'	'daikon'
---------	---------	----------	----------

4 writes

insert is $O(n)$

The last append

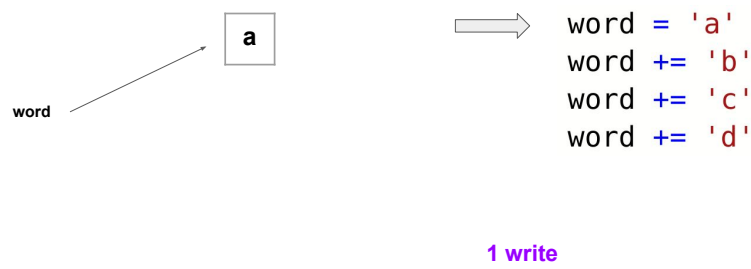
'apple'	'beets'	'carrot'	'daikon'
---------	---------	----------	----------

1 write

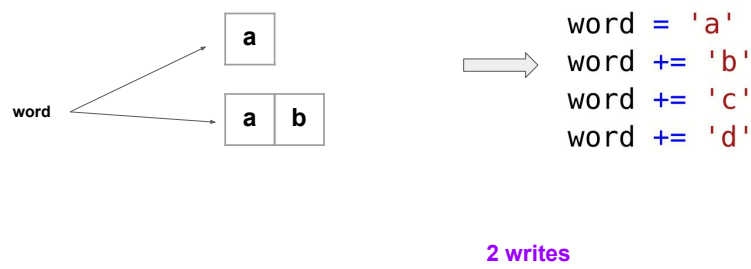
append is $O(1)$

Building a String

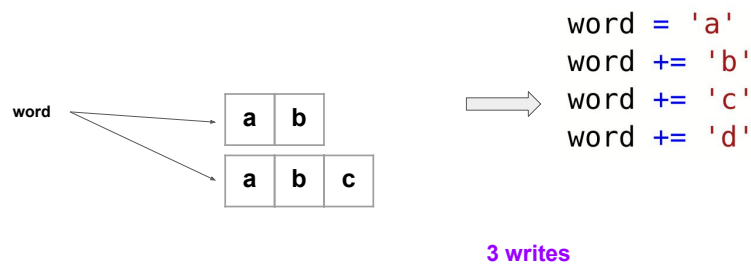
Building a String - using concatenation



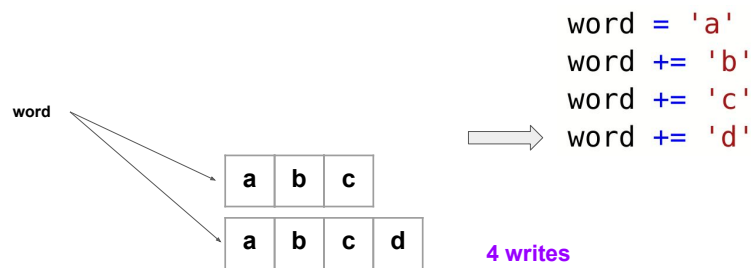
Building a String - revisited



Building a String - revisited



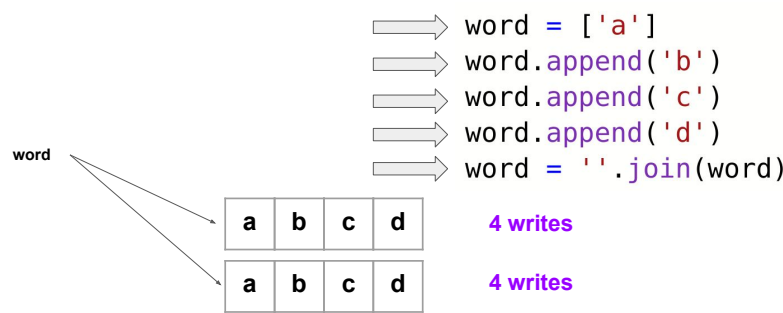
Building a String - revisited



Time Complexity

1 + 2 + 3 + 4 + ... + n is the triangle sum again, which is $O(n^2)$

Building a String - using a list



Time Complexity

4 + 4 writes for this specific example

n + n writes in general

2n is **O(n)**

Space Complexity

Space Complexity

- It is also possible to express the amount of "working memory" as a function of the input size and treat it using the same approach
- Note that both solutions **create a new list** of same length as input list, so we say that the space complexity is **O(n)**
- We'll only count **additional** memory allocated by the function (not size of the input)

```
def reverse(lst):
    new_lst = []
    for item in lst:
        new_lst.insert(0, item)
    return new_lst
```

```
def reverse(lst):
    new_lst = []
    for i in range(len(lst)-1, -1, -1):
        new_lst.append(lst[i])
    return new_lst
```

Time & Space Tradeoffs

Time & Space Tradeoffs

```
def remove_spinach(fridge):
    tasty_foods = []
    for food in fridge:
        if food != 'spinach':
            tasty_foods.append(food)
    fridge.clear()
    fridge.extend(tasty_foods)
```

What is the space complexity? O(n)

What is the time complexity? O(n)

```
def remove_spinach(fridge):
    for i in range(len(fridge)-1, -1, -1):
        if fridge[i] == 'spinach':
            fridge.pop(i)
```

What is the space complexity? O(1)

What is the time complexity? O(n²)

(Hint: think about what the worst-case scenario would be)

Best-Case is O(n)

```
def remove_spinach(fridge):
    for i in range(len(fridge)-1, -1, -1):
        if fridge[i] == 'spinach':
            fridge.pop(i)
```

No spinach:

['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']

Scan through the list once, never popping anything

All spinach:

['spinach', 'spinach', 'spinach', 'spinach', 'spinach',
'spinach', 'spinach', 'spinach']

Each pop is O(1) since it removes the last element

Worst-Case: half-spinach

```
def remove_spinach(fridge):
    for i in range(len(fridge)-1, -1, -1):
        if fridge[i] == 'spinach':
            fridge.pop(i)
```

['spinach', 'spinach', 'spinach', 'spinach', 'w', 'x', 'y', 'z']

Worst-Case: half-spinach

```
def remove_spinach(fridge):
    for i in range(len(fridge)-1, -1, -1):
        if fridge[i] == 'spinach':
            fridge.pop(i)
```

['spinach', 'w', 'spinach', 'x', 'spinach', 'y', 'spinach', 'z']



O(1)

Walkthrough

```
def remove_spinach(fridge):
    for i in range(len(fridge)-1, -1, -1):
        if fridge[i] == 'spinach':
            fridge.pop(i)
```

['spinach', 'w', 'spinach', 'x', 'spinach', 'y', 'spinach', 'z']



O(1)

['spinach', 'w', 'spinach', 'x', 'spinach', 'y', 'z']



1 scoot

Walkthrough

```
def remove_spinach(fridge):
    for i in range(len(fridge)-1, -1, -1):
        if fridge[i] == 'spinach':
            fridge.pop(i)
```

['spinach', 'w', 'spinach', 'x', 'spinach', 'y', 'z']



O(1)

Walkthrough

```
def remove_spinach(fridge):
    for i in range(len(fridge)-1, -1, -1):
        if fridge[i] == 'spinach':
            fridge.pop(i)
```

['spinach', 'w', 'spinach', 'x', 'spinach', 'y', 'z']



O(1)

['spinach', 'w', 'spinach', 'x', 'y', 'z']



2 scoots

Walkthrough

```
def remove_spinach(fridge):
    for i in range(len(fridge)-1, -1, -1):
        if fridge[i] == 'spinach':
            fridge.pop(i)
```

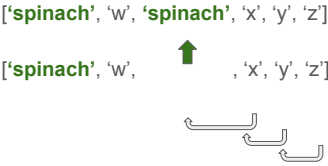
['spinach', 'w', 'spinach', 'x', 'y', 'z']



O(1)

Walkthrough

```
def remove_spinach(fridge):
    for i in range(len(fridge)-1, -1, -1):
        if fridge[i] == 'spinach':
            fridge.pop(i)
```

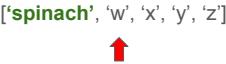


O(1)

3 scoots

Walkthrough

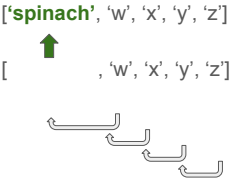
```
def remove_spinach(fridge):
    for i in range(len(fridge)-1, -1, -1):
        if fridge[i] == 'spinach':
            fridge.pop(i)
```



O(1)

Walkthrough

```
def remove_spinach(fridge):
    for i in range(len(fridge)-1, -1, -1):
        if fridge[i] == 'spinach':
            fridge.pop(i)
```



O(1)

4 scoots

Summary

```
def remove_spinach(fridge):
    for i in range(len(fridge)-1, -1, -1):
        if fridge[i] == 'spinach':
            fridge.pop(i)
```

of scoots = $1 + 2 + 3 + \dots + n/2$.
This is half the triangle sum, so $n(n+1)/2 \div 2$
of checks if something is spinach = n
Total runtime is $n(n+1)/4 + n$
...which is $\frac{1}{4}n^2 + \frac{1}{4}n$
...which is **$O(n^2)$**

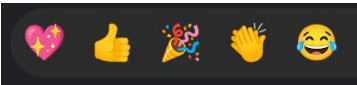
Practice Problem: Time Complexity

You will be working in teams of 2 or 3. The goal is to collaboratively find a solution and be able to explain it to the class. Use the table below to figure out what your role is.

Role	Responsibilities	Assignment Criteria
Driver	Copy and share the repl.it, write the code, make sure you're listening to ideas from your teammates	Person who's gone bowling most recently
Tester	Play devil's advocate, thinks of edge cases, write unit tests for the driver's code	Person who's gone bowling least recently (or never!)
Presenter	Document the code, be prepared to present the team's design decisions, and share one thing the team learned from the problem	Person who has gone bowling medium-recently

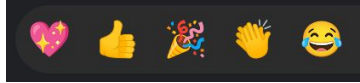
If there are only 2 members in your team, the tester will also take on the presenter role.

```
def fn1(lst):
    for i in range(len(lst)):
        for j in range(10):
            for k in range(20):
                for m in range(10):
                    print('hi')
```



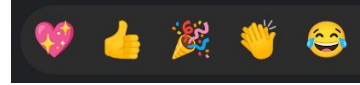
O(1) O(N) O(N²) O(N³) O(N⁴)

```
def fn2(lst):
    for i in range(len(lst)):
        for j in range(len(lst)):
            print('hi')
    return lst[j]
```



$O(1)$ $O(N)$ $O(N^2)$ $O(N^3)$ $O(N^4)$

```
def fn3(lst):
    x = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
    for i in range(len(x)):
        for j in range(len(x)):
            for k in range(len(x)):
                print lst[i + j + k]
```



$O(1)$ $O(N)$ $O(N^2)$ $O(N^3)$ $O(N^4)$