[320] Complexity + Big O (Worksheet: Complexity Analysis)

Department of Computer Sciences
University of Wisconsin-Madison

1

```
def search(L, target):
    for x in L:
        if x == target: #line A
            return True
    return False
```

assume this is asked unless otherwise stated

Let **f(N)** be the number of times line A executes, with `N=len(L)`. What is **f(N)** in each case?

Worst Case (target is at end of list): f(N) = ___________

Best Case (target is at beginning of list): f(N) = ___________

Average Case (target in middle of list): f(N) = ___________

A **step** is any unit of work with bounded execution time (it doesn't keep getting slower with growing input size).

We classify algorithm complexity by classifying the **order of growth** of a function f(N), where f gives the number of steps the algorithm must perform for a given input size.

Big O definition: if $\mathbf{f(N) \leq C * g(N)}$ for large **N** values and some fixed constant **C**, then $\mathbf{f(N) \in O(g(N))}$

1

```
def search(L, target):
    for x in L:
        if x == target: #line A
            return True
    return False
```

Let **f(N)** be the number of times line A executes, with `N=len(L)`. What is **f(N)** in each case?

Worst Case (target is at end of list): f(N) = $\mathbf{N \in O(N)}$

assume this is asked unless otherwise stated

Best Case (target is at beginning of list): f(N) = ________

Average Case (target in middle of list): f(N) = ________

A **step** is any unit of work with bounded execution time (it doesn't keep getting slower with growing input size).

We classify algorithm complexity by classifying the **order of growth** of a function f(N), where f gives the number of steps the algorithm must perform for a given input size.

Big O definition: if $\mathbf{f(N) \le C * g(N)}$ for large **N** values and some fixed constant **C**, then $\mathbf{f(N) \in O(g(N))}$

1

```
def search(L, target):
    for x in L:
        if x == target: #line A
            return True
    return False
```

Let **f(N)** be the number of times line A executes, with `N=len(L)`. What is **f(N)** in each case?

Worst Case (target is at end of list): f(N) = $\mathbf{N \in O(N)}$

assume this is asked unless otherwise stated

Best Case (target is at beginning of list): f(N) = $\mathbf{1 \in O(1)}$

Average Case (target in middle of list): f(N) = ________

A **step** is any unit of work with bounded execution time (it doesn't keep getting slower with growing input size).

We classify algorithm complexity by classifying the **order of growth** of a function f(N), where f gives the number of steps the algorithm must perform for a given input size.

Big O definition: if **f(N) ≤ C * g(N)** for large **N** values and some fixed constant **C**, then **f(N)** ∈ **O(g(N))**

1

```
def search(L, target):
    for x in L:
        if x == target: #line A
            return True
    return False
```

assume this is asked unless otherwise stated

Let **f(N)** be the number of times line A executes, with `N=len(L)`. What is **f(N)** in each case?

Worst Case (target is at end of list): $f(N) = \mathbf{N \in O(N)}$

Best Case (target is at beginning of list): $f(N) = \mathbf{1 \in O(1)}$

Average Case (target in middle of list): $f(N) = \mathbf{\frac{N}{2} \in O(N)}$

A **step** is any unit of work with bounded execution time (it doesn't keep getting slower with growing input size).

We classify algorithm complexity by classifying the **order of growth** of a function f(N), where f gives the number of steps the algorithm must perform for a given input size.

Big O definition: if **f(N) ≤ C * g(N)** for large **N** values and some fixed constant **C**, then **f(N)** ∈ **O(g(N))**

2

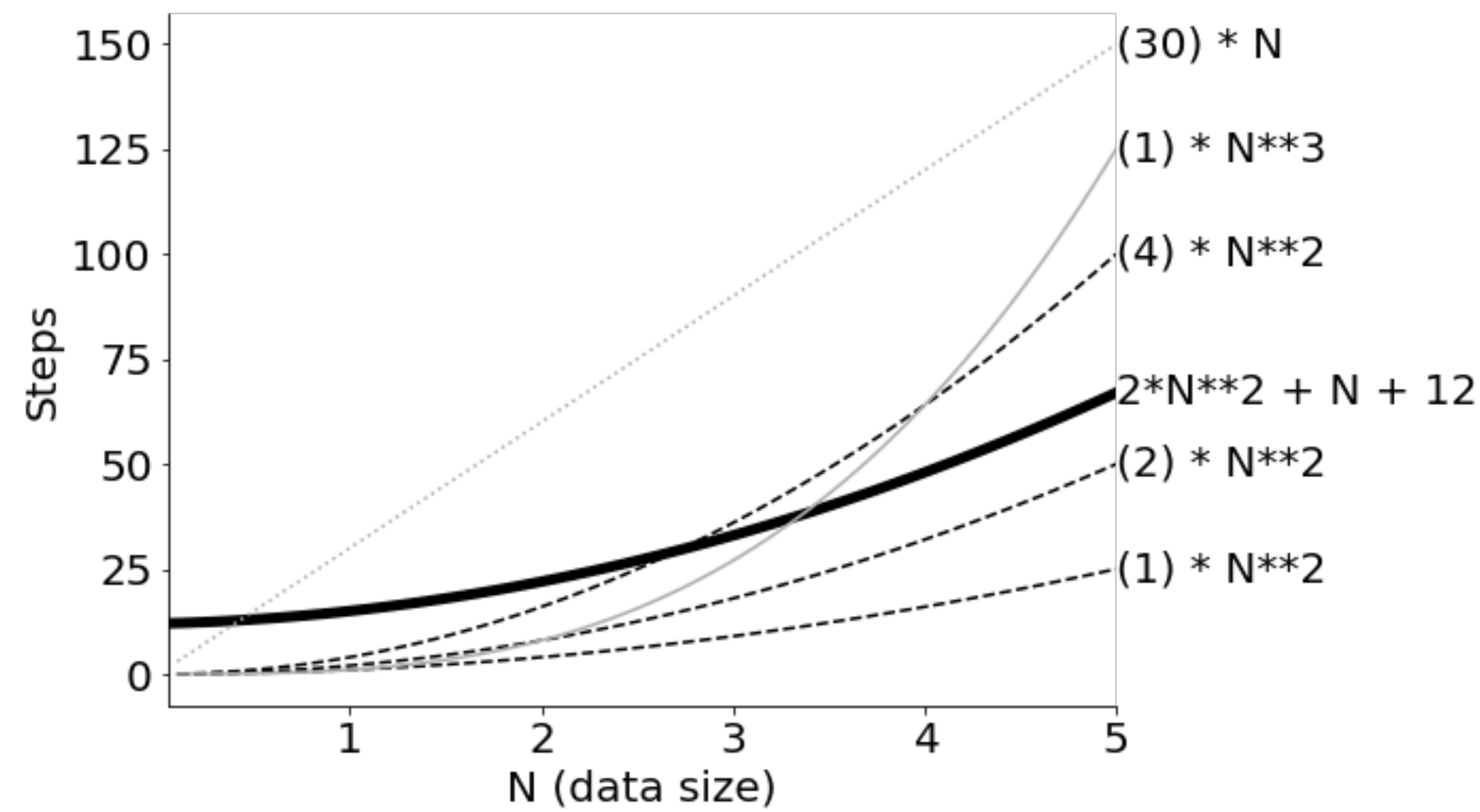
Let $f(N) = 2N^2 + N + 12$

If we want to show $f(N) \in O(N^3)$, what is a good lower bound on N? Let's have C=1.

To show $f(N) \in O(N^2)$, do we pick 1, 2, or 4 for the C? After picking C, what should we choose for N's lower bound?

What is more informative to show? $f(N) \in O(N^3)$ or $f(N) \in O(N^2)$?

Somebody claims $f(N) \in O(N)$, offering C=30 and N>0. Suggest an N value to counter their claim.



2

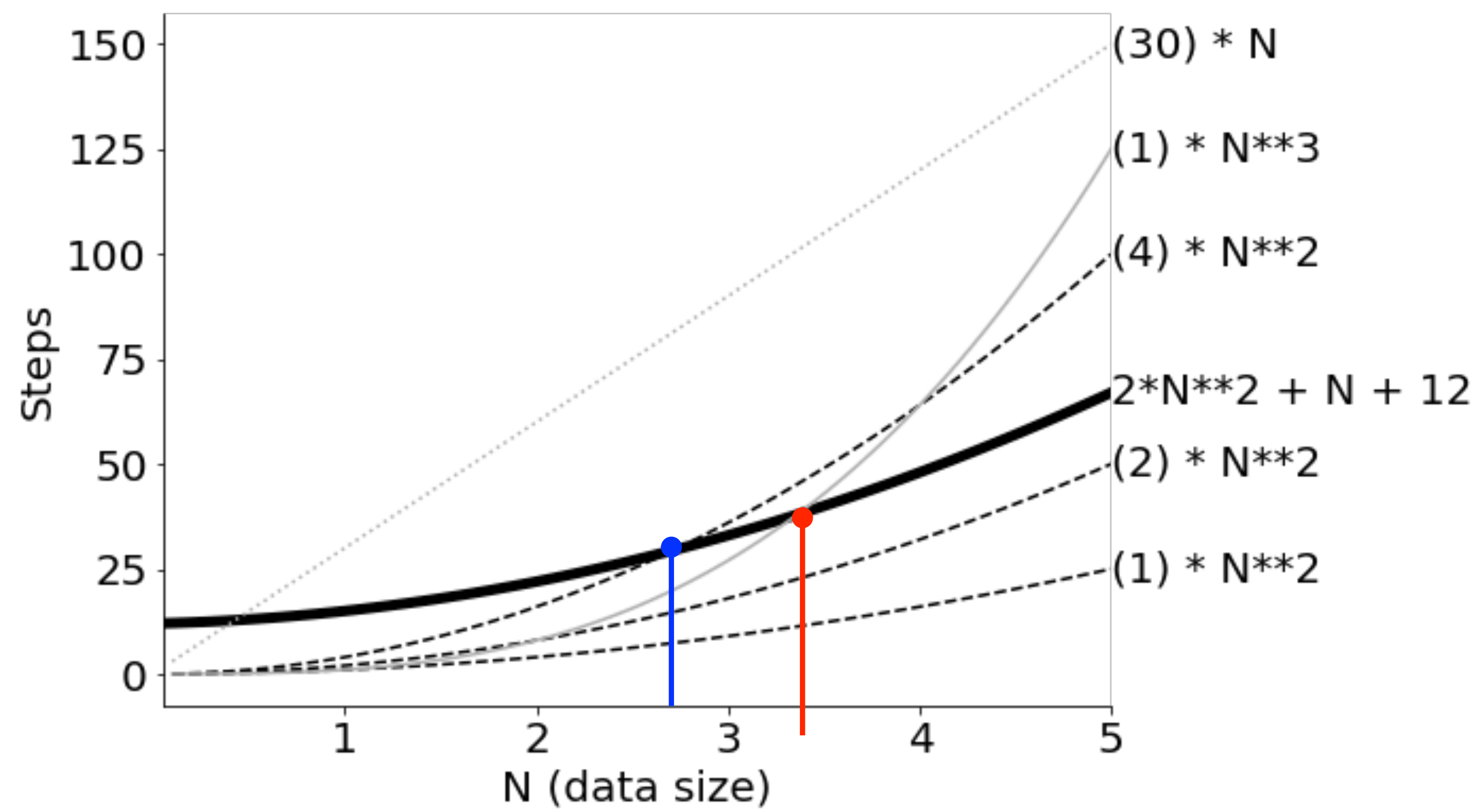
Let $f(N) = 2N^2 + N + 12$

If we want to show $f(N) \in O(N^3)$, what is a good lower bound on N? Let's have C=1. $\mathbf{N \geq 4}$.

To show $f(N) \in O(N^2)$, do we pick 1, 2, or 4 for the C? After picking C, what should we choose for N's lower bound?

What is more informative to show? $f(N) \in O(N^3)$ or $f(N) \in O(N^2)$?

Somebody claims $f(N) \in O(N)$, offering C=30 and N>0. Suggest an N value to counter their claim.



2

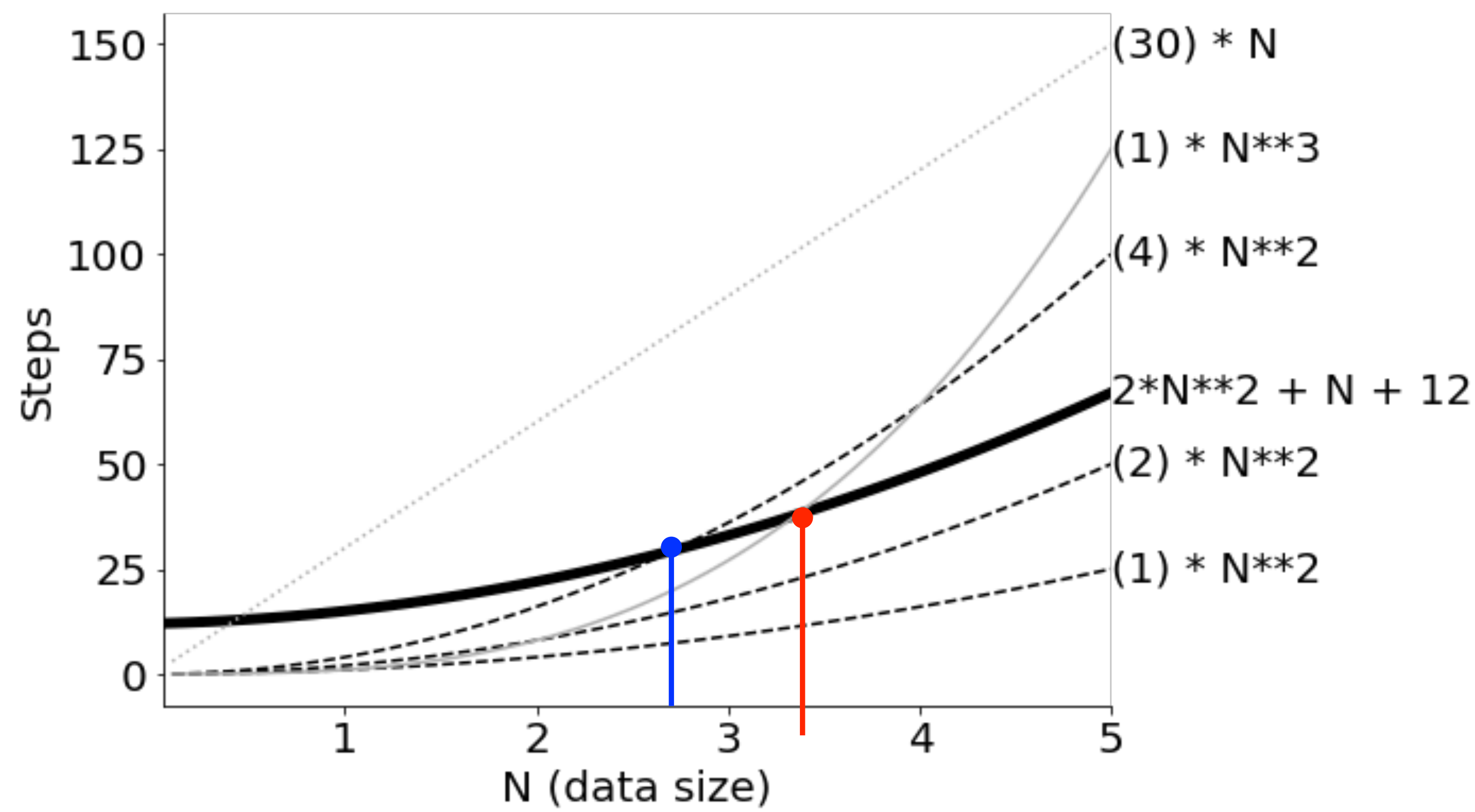
Let $f(N) = 2N^2 + N + 12$

If we want to show $f(N) \in O(N^3)$, what is a good lower bound on N? Let's have C=1. **$N \geq 4$.**

To show $f(N) \in O(N^2)$, do we pick 1, 2, or 4 for the C? After picking C, what should we choose for N's lower bound? **$C = 4$**

What is more informative to show? $f(N) \in O(N^3)$ or $f(N) \in O(N^2)$?

Somebody claims $f(N) \in O(N)$, offering C=30 and N>0. Suggest an N value to counter their claim.



2

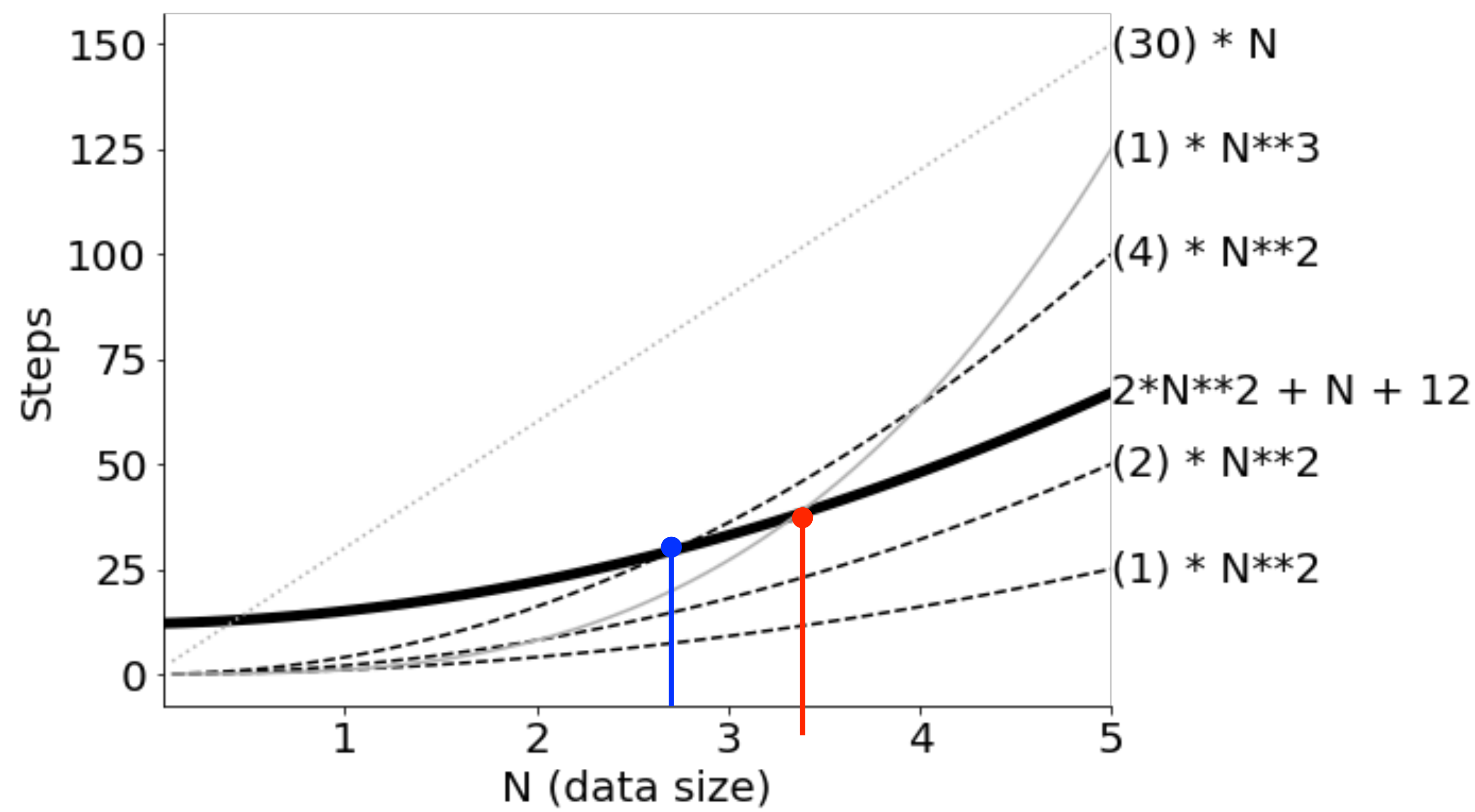
Let $f(N) = 2N^2 + N + 12$

If we want to show $f(N) \in O(N^3)$, what is a good lower bound on N? Let's have C=1. $\mathbf{N \geq 4}$.

To show $f(N) \in O(N^2)$, do we pick 1, 2, or 4 for the C? After picking C, what should we choose for N's lower bound? $\mathbf{C = 4}$ and $\mathbf{N \geq 3}$.

What is more informative to show? $f(N) \in O(N^3)$ or $f(N) \in O(N^2)$?

Somebody claims $f(N) \in O(N)$, offering C=30 and N>0. Suggest an N value to counter their claim.



2

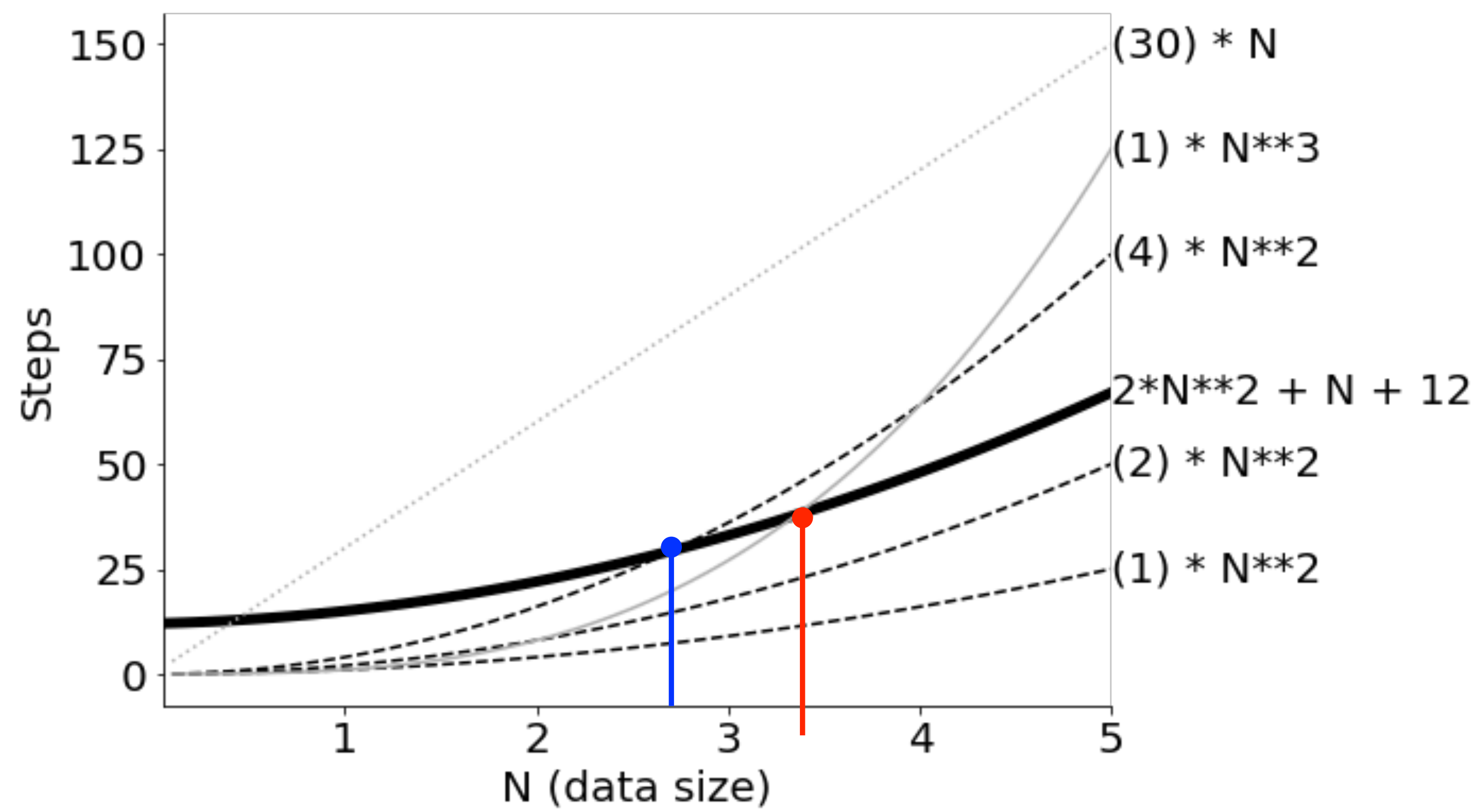
Let f(N) = $2N^2 + N + 12$

If we want to show $f(N) \in O(N^3)$, what is a good lower bound on N? Let's have C=1. $N \geq 4$.

To show $f(N) \in O(N^2)$, do we pick 1, 2, or 4 for the C? After picking C, what should we choose for N's lower bound?
$C = 4$ and $N \geq 3$.

What is more informative to show?
$f(N) \in O(N^3)$ or $f(N) \in O(N^2)$?
$f(N) \in O(N^2)$.

Somebody claims $f(N) \in O(N)$, offering C=30 and N>0. Suggest an N value to counter their claim.



2

Let $f(N) = 2N^2 + N + 12$

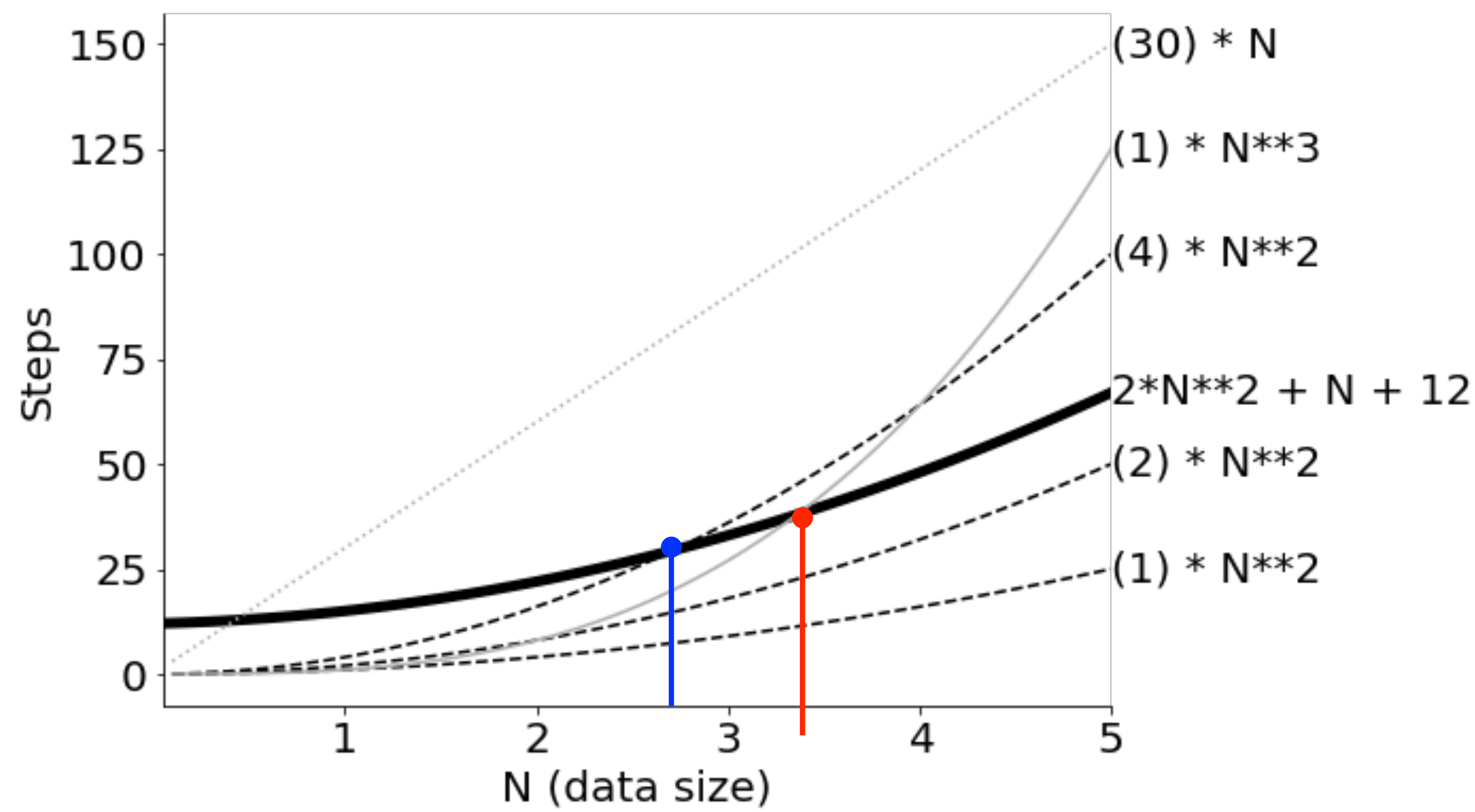
If we want to show $f(N) \in O(N^3)$, what is a good lower bound on N? Let's have C=1. $\mathbf{N \geq 4}$

To show $f(N) \in O(N^2)$, do we pick 1, 2, or 4 for the C? After picking C, what should we choose for N's lower bound?
$\mathbf{C = 4}$ and $\mathbf{N \geq 3}$

What is more informative to show?
$f(N) \in O(N^3)$ or $f(N) \in O(N^2)$?
$\mathbf{f(N) \in O(N^2)}$ (tighter upper bound)

Somebody claims $f(N) \in O(N)$, offering C=30 and N>0. Suggest an N value to counter their claim.

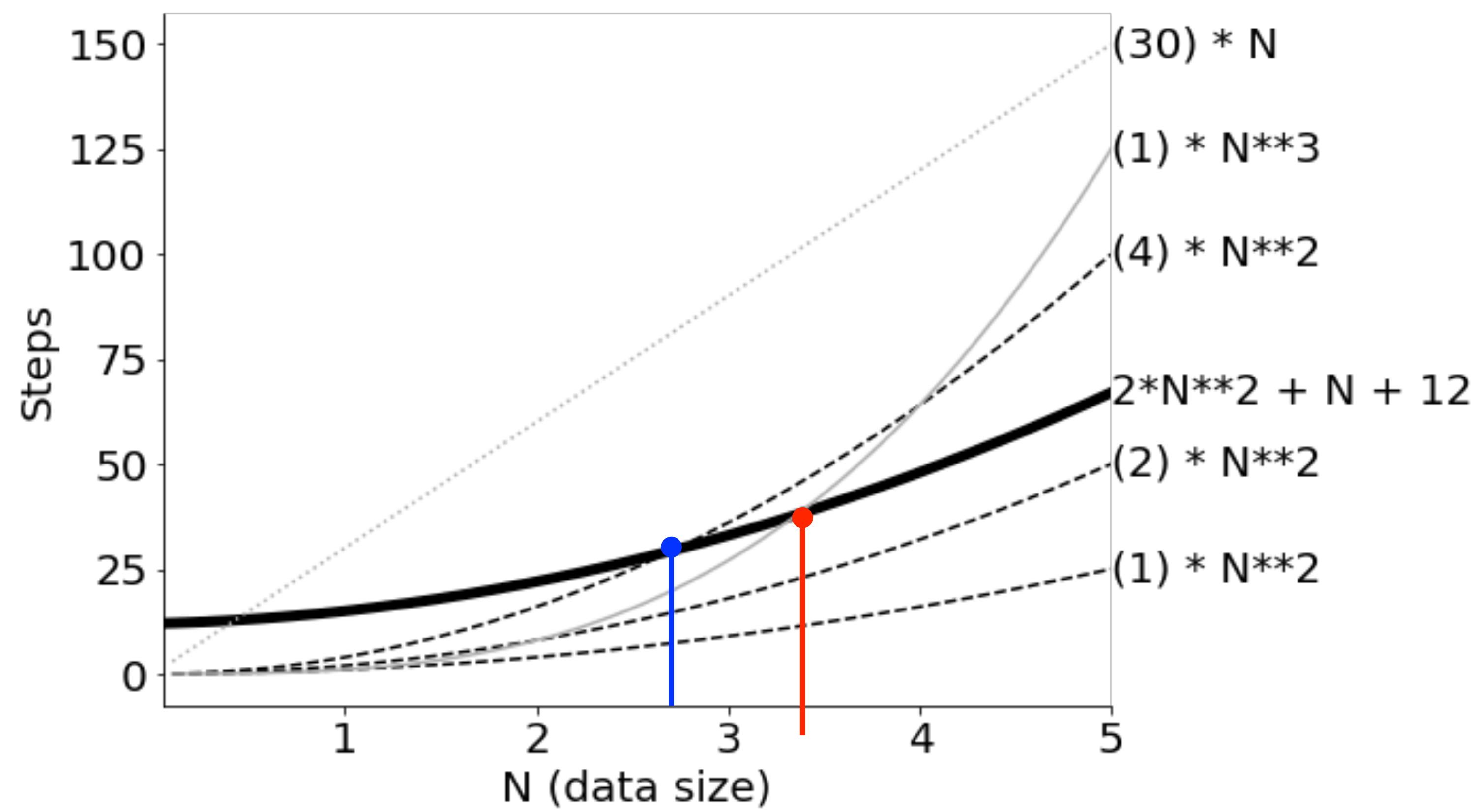
Assume $\mathbf{N = 20}$. and
$\mathbf{2N^2 + N + 12 \leq 30N}$.



2

Let f(N) = $2N^2 + N + 12$

If we want to show $f(N) \in O(N^3)$, what is a good lower bound on N? Let's have C=1. $\mathbf{N \geq 4}$



To show $f(N) \in O(N^2)$, do we pick 1, 2, or 4 for the C? After picking C, what should we choose for N's lower bound?
$\mathbf{C = 4}$ and $\mathbf{N \geq 3}$

What is more informative to show?
$f(N) \in O(N^3)$ or $f(N) \in O(N^2)$?
$\mathbf{f(N) \in O(N^2)}$ (tighter upper bound)

Somebody claims $f(N) \in O(N)$, offering C=30 and N>0. Suggest an N value to counter their claim.

Assume $\mathbf{N = 20.}$ and
$\mathbf{2N^2 + N + 12 \leq 30N}$.
However, $\mathbf{800 + 20 + 12 \nleq 600}$.

2

Let $f(N) = 2N^2 + N + 12$

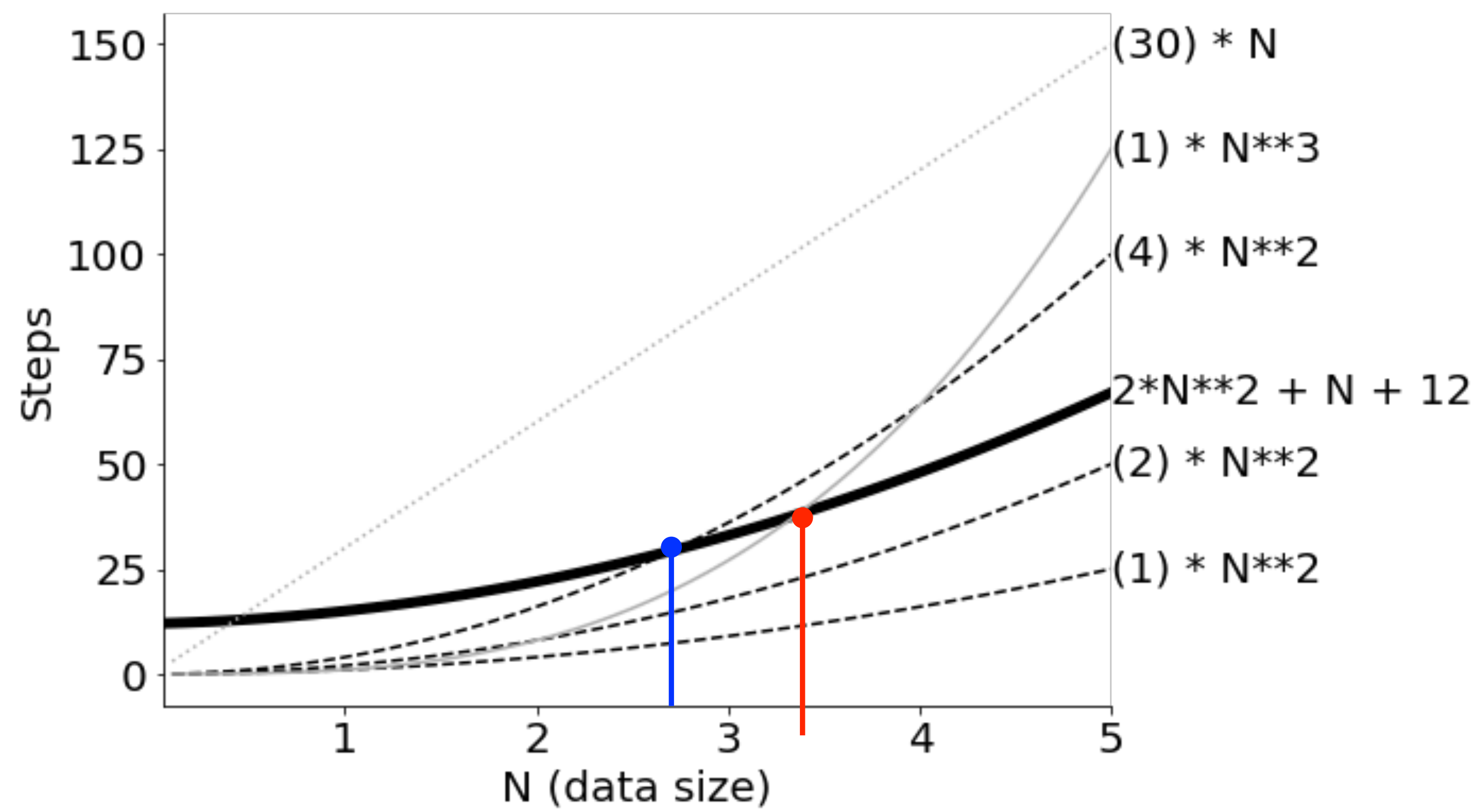
If we want to show $f(N) \in O(N^3)$, what is a good lower bound on N? Let's have C=1. $\mathbf{N \geq 4}$

To show $f(N) \in O(N^2)$, do we pick 1, 2, or 4 for the C? After picking C, what should we choose for N's lower bound?
$\mathbf{C = 4}$ and $\mathbf{N \geq 3}$

What is more informative to show?
$f(N) \in O(N^3)$ or $f(N) \in O(N^2)$?
$\mathbf{f(N) \in O(N^2)}$ (tighter upper bound)

Somebody claims $f(N) \in O(N)$, offering C=30 and N>0. Suggest an N value to counter their claim.

Assume $\mathbf{N = 20}$. and
$\mathbf{2N^2 + N + 12 \leq 30N}$.
However, $\mathbf{800 + 20 + 12 \nleq 600}$.
Therefore, the suggest value of
$\mathbf{N = 20}$.



3

```
nums = [...]

first100sum = 0

for x in nums[:100]:
    first100sum += x
print(first100sum)
```

If we increase the size of nums from 20 items to 100 items, the code will probably take ________ times longer to run.

If we increase the size of nums from 100 to 1000, will the code take longer? Yes / No

The complexity of the code is O(_____), with N=len(nums).

3

```
nums = [...]

first100sum = 0

for x in nums[:100]:
    first100sum += x
print(first100sum)
```

If we increase the size of nums from 20 items to 100 items, the code will probably take **5** times longer to run.

If we increase the size of nums from 100 to 1000, will the code take longer? Yes / No

The complexity of the code is O(______), with N=len(nums).

3

```
nums = [...]

first100sum = 0

for x in nums[:100]:
    first100sum += x
print(first100sum)
```

If we increase the size of nums from 20 items to 100 items, the code will probably take **5** times longer to run.

If we increase the size of nums from 100 to 1000, will the code take longer? Yes / No

No

The complexity of the code is O(______), with N=len(nums).

3

```
nums = [...]

first100sum = 0

for x in nums[:100]:
    first100sum += x
print(first100sum)
```

If we increase the size of nums from 20 items to 100 items, the code will probably take **5** times longer to run.

If we increase the size of nums from 100 to 1000, will the code take longer? Yes / No

No

The complexity of the code is O(**1**), with N=len(nums).

4

Each of the following list operations are either O(1) or O(N), where N is len(L). Circle those you think are O(N).

```
L.insert(0, x)    L.pop(0)     x = L[0]        x = max(L)     x = len(L)

L.append(x)       L.pop(-1)   L2.extend(L)    x = sum(L)     found = X in L
```

4

Each of the following list operations are either O(1) or O(N), where N is len(L). Circle those you think are O(N).

`L.insert(0, x)`	`L.pop(0)`	`x = L[0]`	`x = max(L)`	`x = len(L)`
`L.append(x)`	`L.pop(-1)`	`L2.extend(L)`	`x = sum(L)`	`found = X in L`

4

Each of the following list operations are either **O(1)** or **O(N)**, where N is len(L). Circle those you think are **O(N)**.

```
L.insert(0, x)    L.pop(0)     x = L[0]       x = max(L)    x = len(L)

L.append(x)       L.pop(-1)    L2.extend(L)   x = sum(L)    found = X in L
```

4

Each of the following list operations are either **O(1)** or **O(N)**, where N is len(L). Circle those you think are **O(N)**.

`L.insert(0, x)` `L.pop(0)` `x = L[0]` `x = max(L)` `x = len(L)`

`L.append(x)` `L.pop(-1)` `L2.extend(L)` `x = sum(L)` `found = X in L`

4

Each of the following list operations are either **O(1)** or **O(N)**, where N is len(L). Circle those you think are **O(N)**.

`L.insert(0, x)` `L.pop(0)` `x = L[0]` `x = max(L)` `x = len(L)`

`L.append(x)` `L.pop(-1)` `L2.extend(L)` `x = sum(L)` `found = X in L`

4

Each of the following list operations are either O(1) or O(N), where N is len(L). Circle those you think are O(N).

`L.insert(0, x)` `L.pop(0)` `x = L[0]` `x = max(L)` `x = len(L)`

`L.append(x)` `L.pop(-1)` `L2.extend(L)` `x = sum(L)` `found = X in L`

4

Each of the following list operations are either **O(1)** or **O(N)**, where N is len(L). Circle those you think are **O(N)**.

`L.insert(0, x)` `L.pop(0)` `x = L[0]` `x = max(L)` `x = len(L)`

`L.append(x)` `L.pop(-1)` `L2.extend(L)` `x = sum(L)` `found = X in L`

5

```
L = [...]
for x in L:
    avg = sum(L) / len(L)
    if x > 2*avg:
        print("outlier", x)
```

What is the big O complexity?

Is there a way to optimize the code?

5

```
L = [...]
for x in L: N + 1 steps
    avg = sum(L) / len(L)
    if x > 2*avg:
        print("outlier", x)
```

What is the big O complexity?

Is there a way to optimize the code?

5

```
L = [...]
for x in L: N+1 steps
    avg = sum(L) / len(L) N steps
    if x > 2*avg:
        print("outlier", x)
```

What is the big O complexity?

Is there a way to optimize the code?

5

```
L = [...]
for x in L:
    avg = sum(L) / len(L)
    if x > 2*avg:
        print("outlier", x)
```

$\mathbf{N+1}$ steps (for x in L)

$\mathbf{N}$ steps (avg = sum(L) / len(L))

What is the big O complexity?

$$\mathbf{O((N+1)N) = O(N^2+N)}$$

Is there a way to optimize the code?

5

```
L = [...]
for x in L:            N + 1 steps
    avg = sum(L) / len(L)  N steps
    if x > 2*avg:
        print("outlier", x)
```

What is the big O complexity?

$$\mathbf{O((N+1)N) = O(N^2 + N) = O(N^2)}$$

Is there a way to optimize the code?

5

```
L = [...]
for x in L: N + 1 steps
    avg = sum(L) / len(L) N steps
    if x > 2*avg:
        print("outlier", x)
```

What is the big O complexity?

$$\mathbf{O((N+1)N) = O(N^2 + N) = O(N^2)}$$

Is there a way to optimize the code?

Calculate avg outside the loop.

6

```
A = [...]
B = [...]

for x in A:
    for y in B:
        print(x*y)
```

how would you define the variable(s) to describe the size of the input data?

The complexity of code is

O(________________)

6

```
A = [...]
B = [...]

for x in A:
    for y in B:
        print(x*y)
```

$len(A) = M$

how would you define the variable(s) to describe the size of the input data?

The complexity of code is

O(_______________)

6

```
A = [...]
B = [...]

for x in A:
    for y in B:
        print(x*y)
```

$len(A) = M$

$len(B) = N$

how would you define the variable(s) to describe the size of the input data?

The complexity of code is

O(_______________)

6

```
A = [...]
B = [...]

for x in A:
    for y in B:
        print(x*y)
```

$len(A) = M$
$len(B) = N$

how would you define the variable(s) to describe the size of the input data?

$len(A) = M$ and $len(B) = N$

The complexity of code is

O(________________)

6

```
A = [...]
B = [...]

for x in A:
    for y in B:
        print(x*y)
```

$len(A) = M$

$len(B) = N$

$\mathbf{M+1}$ steps

how would you define the variable(s) to describe the size of the input data?

$len(A) = M$ and $len(B) = N$

The complexity of code is

O(______________)

6

```
A = [...]
B = [...]

for x in A:
    for y in B:
        print(x*y)
```

$len(A) = M$
$len(B) = N$

$\mathbf{M+1}$ steps
$\mathbf{N+1}$ steps

how would you define the variable(s) to describe the size of the input data?

$len(A) = M$ and $len(B) = N$

The complexity of code is

O(______________)

6

```
A = [...]
B = [...]

for x in A:
    for y in B:
        print(x*y)
```

$len(A) = M$

$len(B) = N$

for x in A: $\mathbf{M+1}$ steps

for y in B: $\mathbf{N+1}$ steps

how would you define the variable(s) to describe the size of the input data?

$len(A) = M$ and $len(B) = N$

The complexity of code is

$O((\mathbf{M+1})(\mathbf{N+1})) = O(\mathbf{MN+M+N+1}) = O(\mathbf{MN})$

8

```
s1 = tuple("...") # could be any string
s2 = tuple(“...")
```

```
# version A
import itertools

matches = False
for p in itertools.permutations(s1):
    if p == s2:
        matches = True
```

```
# version B
s1 = sorted(s1)
s2 = sorted(s2)
matches = (s1 == s2)
```

assumed sorted is O(N log N)

what is the complexity of version A? O(_______________)

what is the complexity of version B? O(_______________)

8

```
s1 = tuple("...") # could be any string
s2 = tuple("...")
```

$len(\mathrm{s1}) = N$

```
# version A
import itertools

matches = False
for p in itertools.permutations(s1):
    if p == s2:
        matches = True
```

```
# version B
s1 = sorted(s1)
s2 = sorted(s2)
matches = (s1 == s2)
```

assumed sorted is O(N log N)

what is the complexity of version A? O(______________)

what is the complexity of version B? O(______________)

8

```
s1 = tuple("...") # could be any string
s2 = tuple("...")
```

$len(\mathrm{s1}) = N$
$len(\mathrm{s2}) = N$

```
# version A
import itertools

matches = False
for p in itertools.permutations(s1):
    if p == s2:
        matches = True
```

```
# version B
s1 = sorted(s1)
s2 = sorted(s2)
matches = (s1 == s2)
```

assumed sorted is O(N log N)

what is the complexity of version A? O(_______________)

what is the complexity of version B? O(_______________)

8

```
s1 = tuple("...") # could be any string
s2 = tuple("...")
```

$len(s1) = N$
$len(s2) = N$

```
For Example, s1 = (A, B, C), then permutations of s1 are
```

ABC BCA
ACB CAB
BAC CBA

```
# version A
import itertools

matches = False
for p in itertools.permutations(s1):
    if p == s2:
        matches = True
```

```
# version B
s1 = sorted(s1)
s2 = sorted(s2)
matches = (s1 == s2)
```

assumed sorted is O(N log N)

what is the complexity of version A? O(______________)

what is the complexity of version B? O(______________)

8

```
s1 = tuple("...") # could be any string
s2 = tuple("...")
```

$len(\mathrm{s1}) = N$
$len(\mathrm{s2}) = N$

```
For Example, s1 = (A, B, C), then permutations of s1 are
```

ABC BCA
ACB CAB
BAC CBA

```
# version A
import itertools

matches = False
for p in itertools.permutations(s1):
    if p == s2:
        matches = True
```

N! steps

```
# version B
s1 = sorted(s1)
s2 = sorted(s2)
matches = (s1 == s2)
```

assumed sorted is O(N log N)

what is the complexity of version A? O(____________)

what is the complexity of version B? O(____________)

8

```
s1 = tuple("...") # could be any string
s2 = tuple("...")
```

$len(\mathrm{s1}) = N$
$len(\mathrm{s2}) = N$

```
For Example, s1 = (A, B, C), then permutations of s1 are
```

ABC BCA
ACB CAB
BAC CBA

```
# version A
import itertools

matches = False
for p in itertools.permutations(s1): N! steps
    if p == s2: N steps
        matches = True
```

```
# version B
s1 = sorted(s1)
s2 = sorted(s2)
matches = (s1 == s2)
```

assumed sorted is O(N log N)

what is the complexity of version A? O(______________)

what is the complexity of version B? O(______________)

8

```
s1 = tuple("...") # could be any string
s2 = tuple("...")
```

$len(s1) = N$
$len(s2) = N$

```
For Example, s1 = (A, B, C), then permutations of s1 are
```

ABC BCA
ACB CAB
BAC CBA

```
# version A
import itertools

matches = False
for p in itertools.permutations(s1): N! steps
    if p == s2: N steps
        matches = True
```

```
# version B
s1 = sorted(s1)
s2 = sorted(s2)
matches = (s1 == s2)
```

assumed sorted is O(N log N)

what is the complexity of version A? O(**N * N!**)

what is the complexity of version B? O(______________)

8

```
s1 = tuple("...") # could be any string
s2 = tuple("...")
```

$len(s1) = N$
$len(s2) = N$

```
For Example, s1 = (A, B, C), then permutations of s1 are
```

ABC BCA
ACB CAB
BAC CBA

```
# version A
import itertools

matches = False
for p in itertools.permutations(s1):
    if p == s2:
        matches = True
```

`for p in itertools.permutations(s1):` **N**! steps

`if p == s2:` **N** steps

```
# version B
s1 = sorted(s1)
s2 = sorted(s2)
matches = (s1 == s2)
```

`s1 = sorted(s1)` **N** log**N**

`s2 = sorted(s2)` **N** log**N**

Example, merge sort, quick sort

assumed sorted is O(N log N)

what is the complexity of version A? O(**N** * **N**!)

what is the complexity of version B? O(_______________)

8

```
s1 = tuple("...") # could be any string
s2 = tuple("...")
```

$len(s1) = N$
$len(s2) = N$

```
For Example, s1 = (A, B, C), then permutations of s1 are
```

ABC BCA
ACB CAB
BAC CBA

```
# version A
import itertools

matches = False
for p in itertools.permutations(s1): N! steps
    if p == s2: N steps
        matches = True
```

```
# version B
s1 = sorted(s1)      N logN
s2 = sorted(s2)      N logN
matches = (s1 == s2)

Example, merge sort, quick sort
```

assumed sorted is O(N log N)

what is the complexity of version A? $O(\mathbf{N} * \mathbf{N}!)$

what is the complexity of version B? $O(\mathbf{2N}\ \log\mathbf{N}) = O(\mathbf{N}\ \log\mathbf{N})$