[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 **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
```

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) = ________

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

assume this is asked unless otherwise stated

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) = $\mathbf{N \in O(N)}$

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

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

assume this is asked unless otherwise stated

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): $f(N) = \mathbf{N} \in \mathbf{O(N)}$

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

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

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

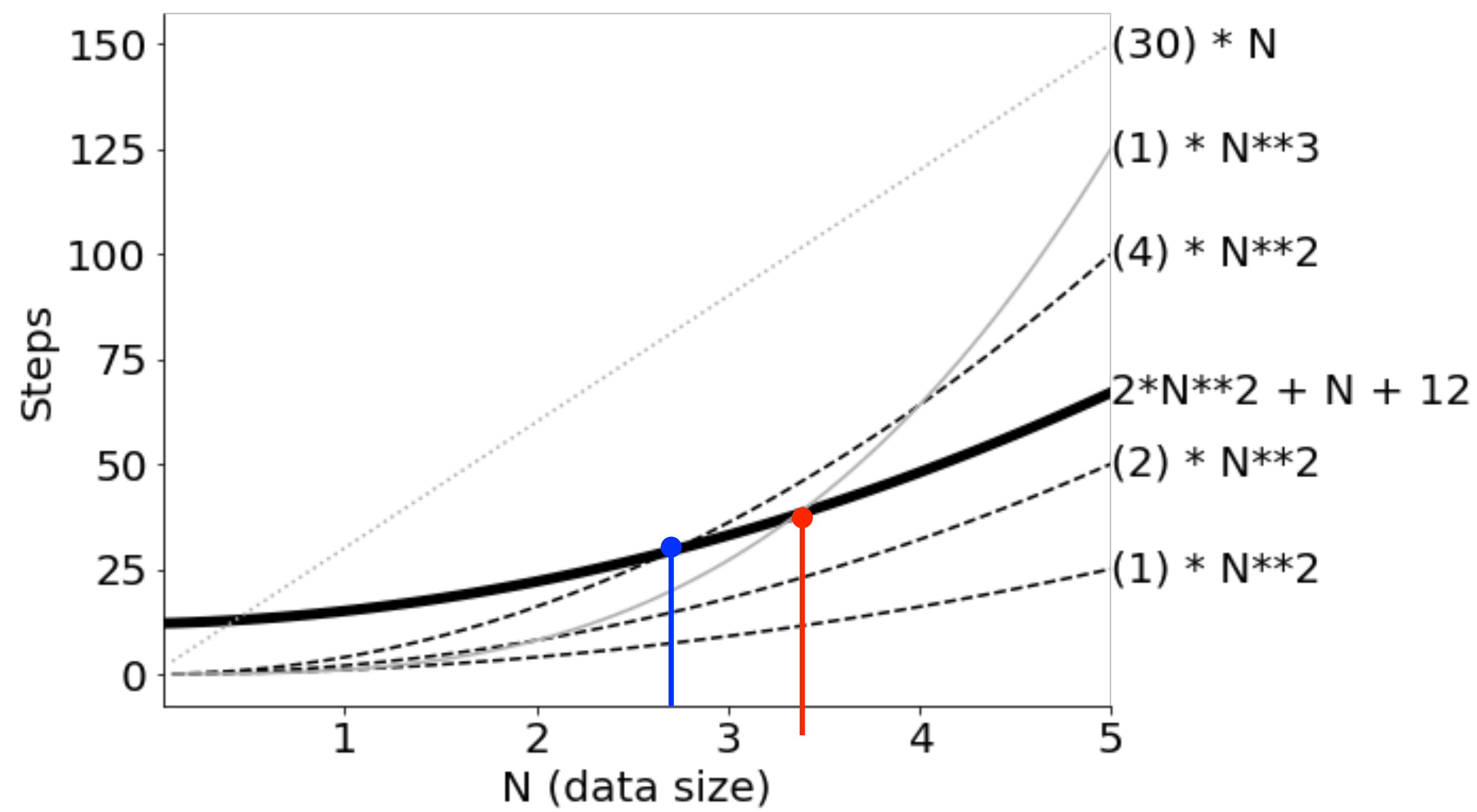
Let $f(N) = 2N^2 + N + 12$

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

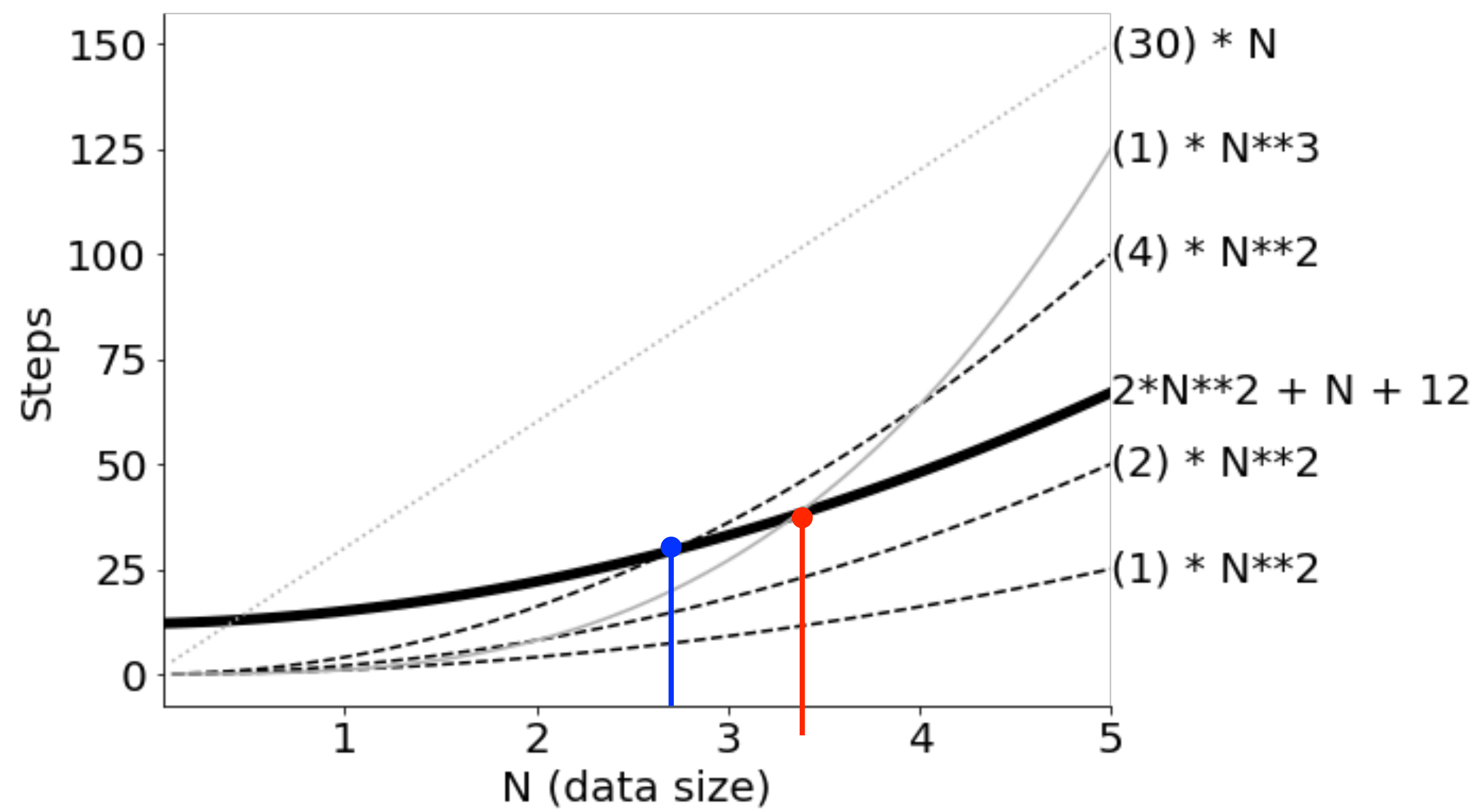
Let $f(N) = 2N^2 + N + 12$

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

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

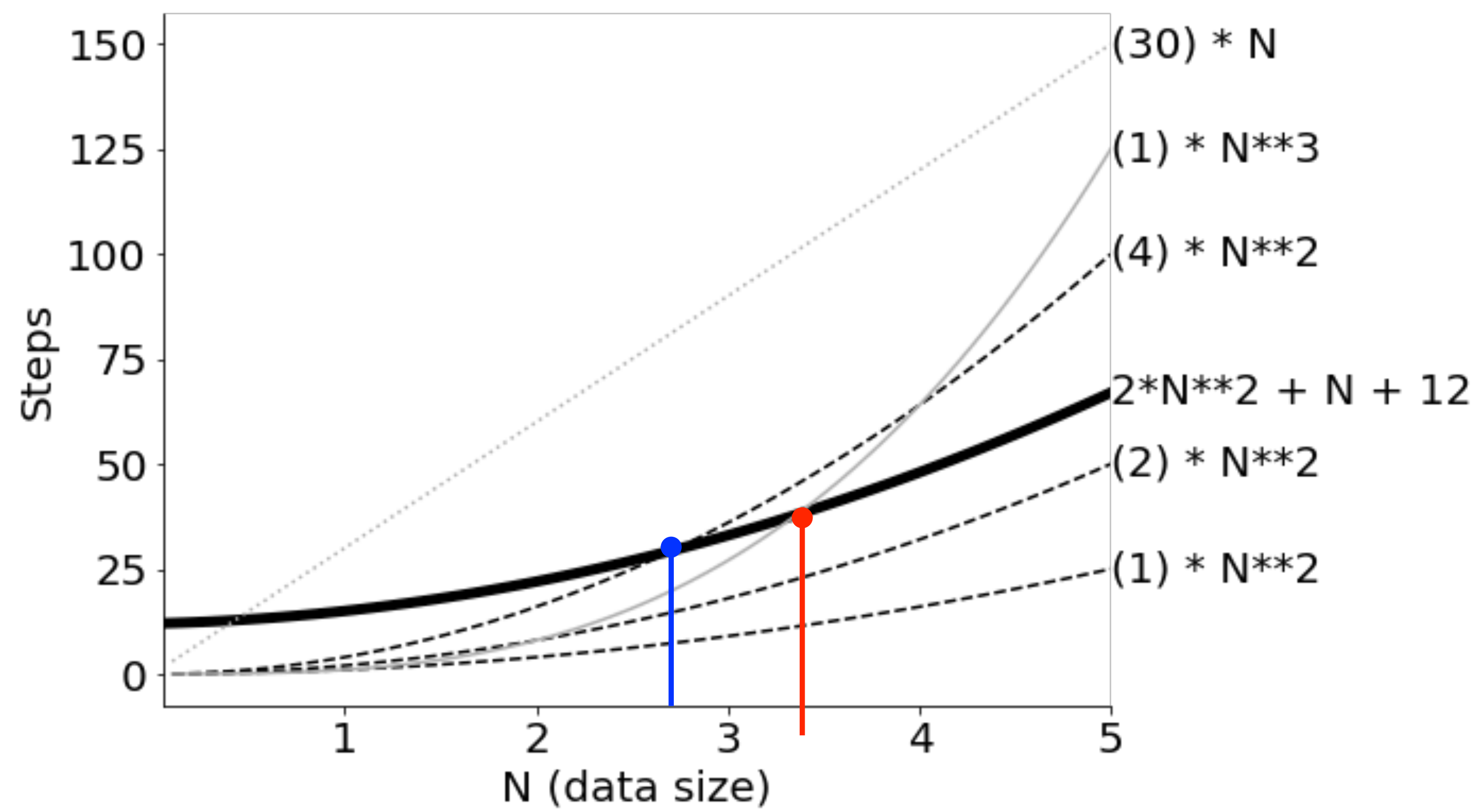
Let $f(N) = 2N^2 + N + 12$

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

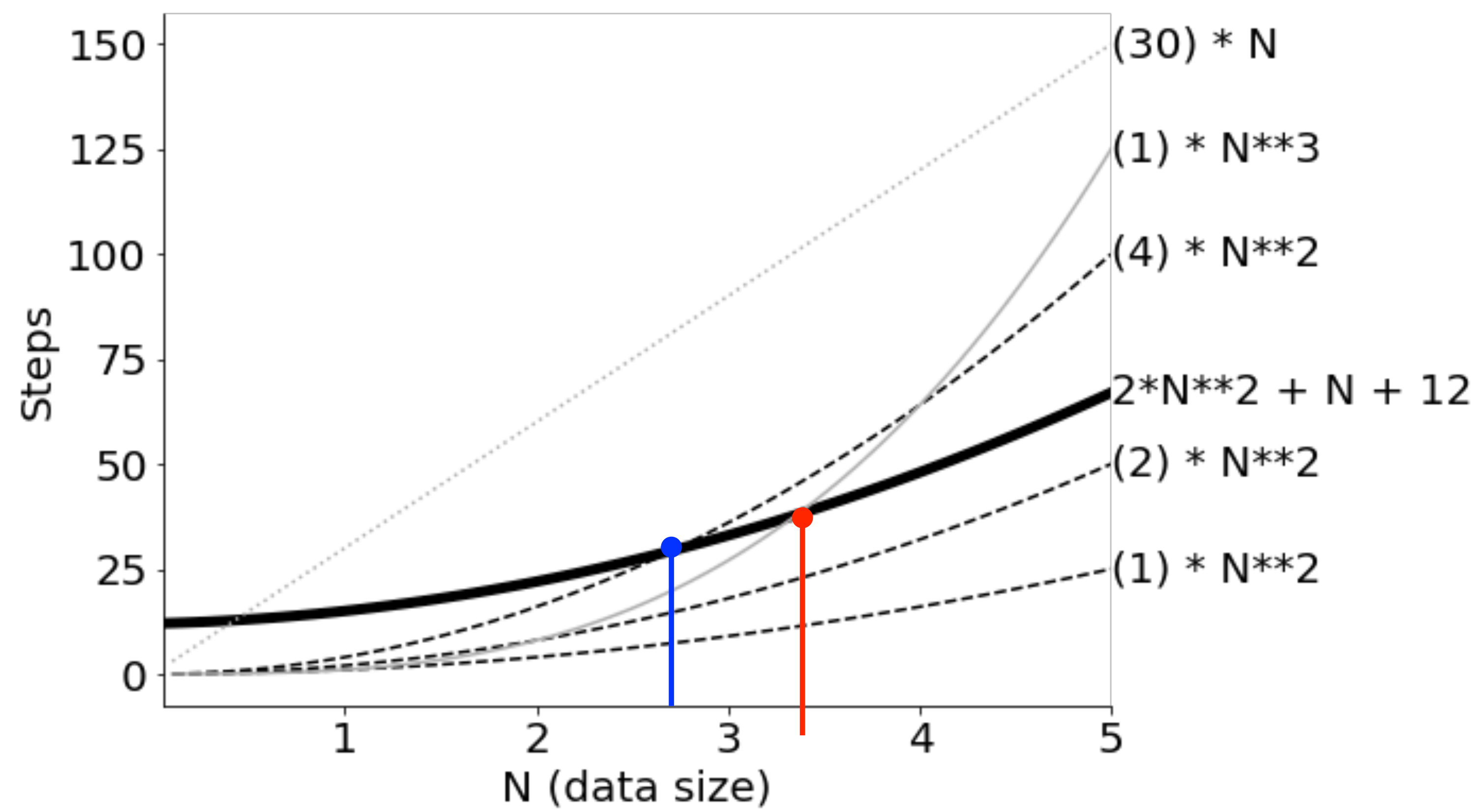
Let f(N) = $2N^2 + N + 12$

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

To show f(N) ∈ 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) ∈ O(N^3) or f(N) ∈ O(N^2)?

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



2

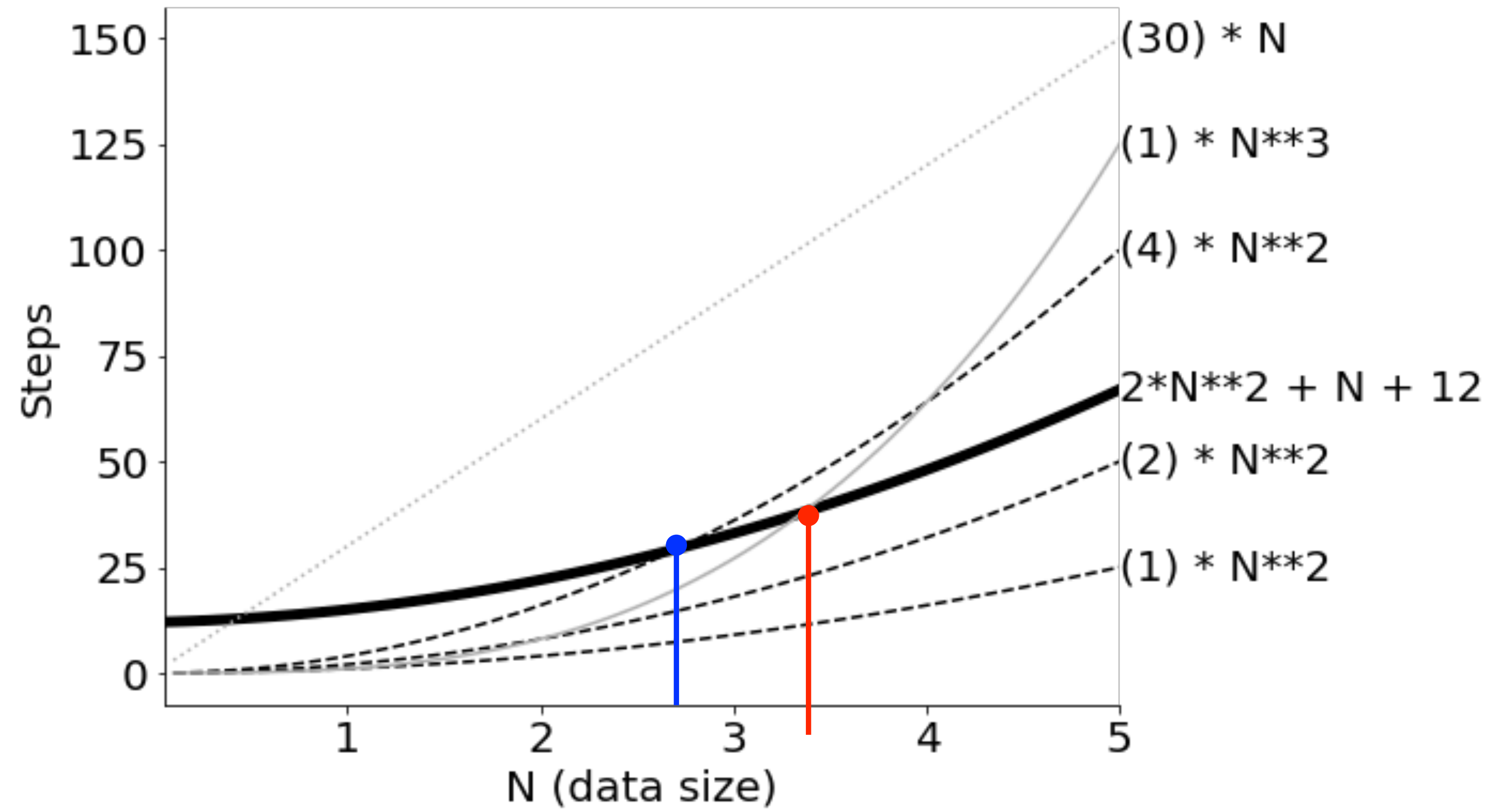
Let $f(N) = 2N^2 + N + 12$

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 $f(N) \in O(N^2)$?
$\mathbf{f(N) \in O(N^2)}$

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



2

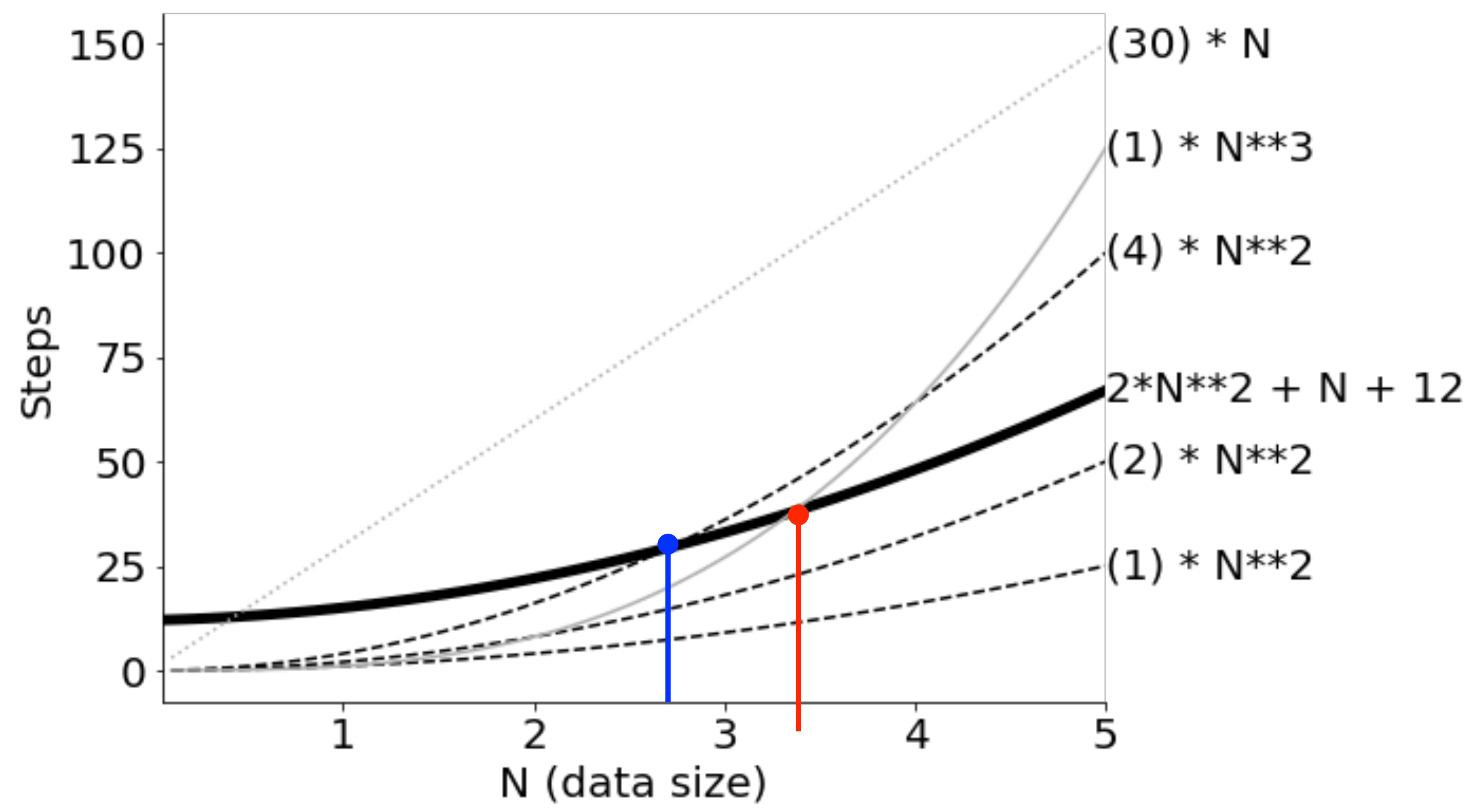
Let $f(N) = 2N^2 + N + 12$

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 $f(N) \in O(N^2)$?
$f(N) \in O(N^2)$ (tighter upper bound)

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



2

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

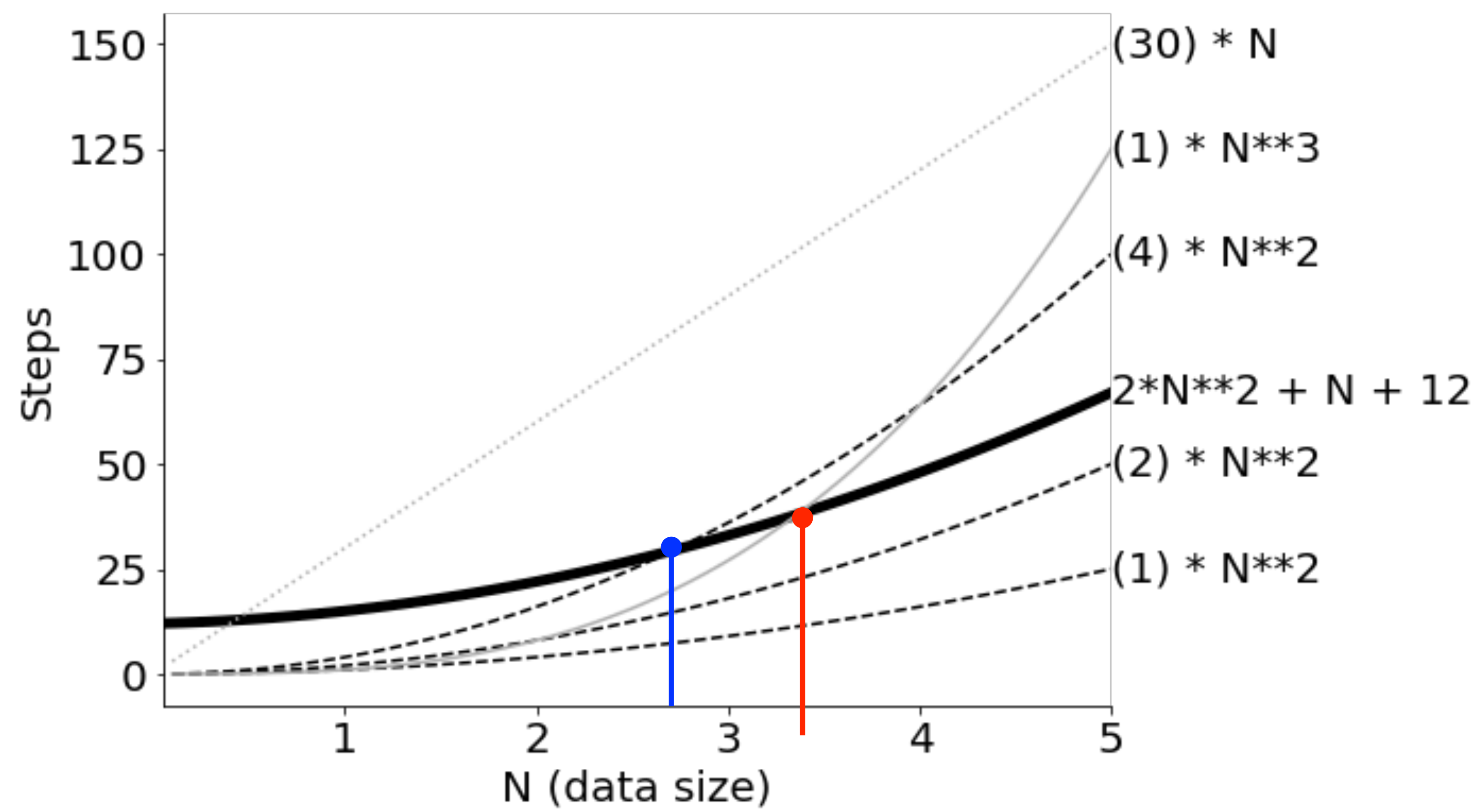
If we want to show $f(N) \in O(N^3)$, what is a good lower bound on N? Let's have C=1. $\mathbf{N \geq 4}$

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

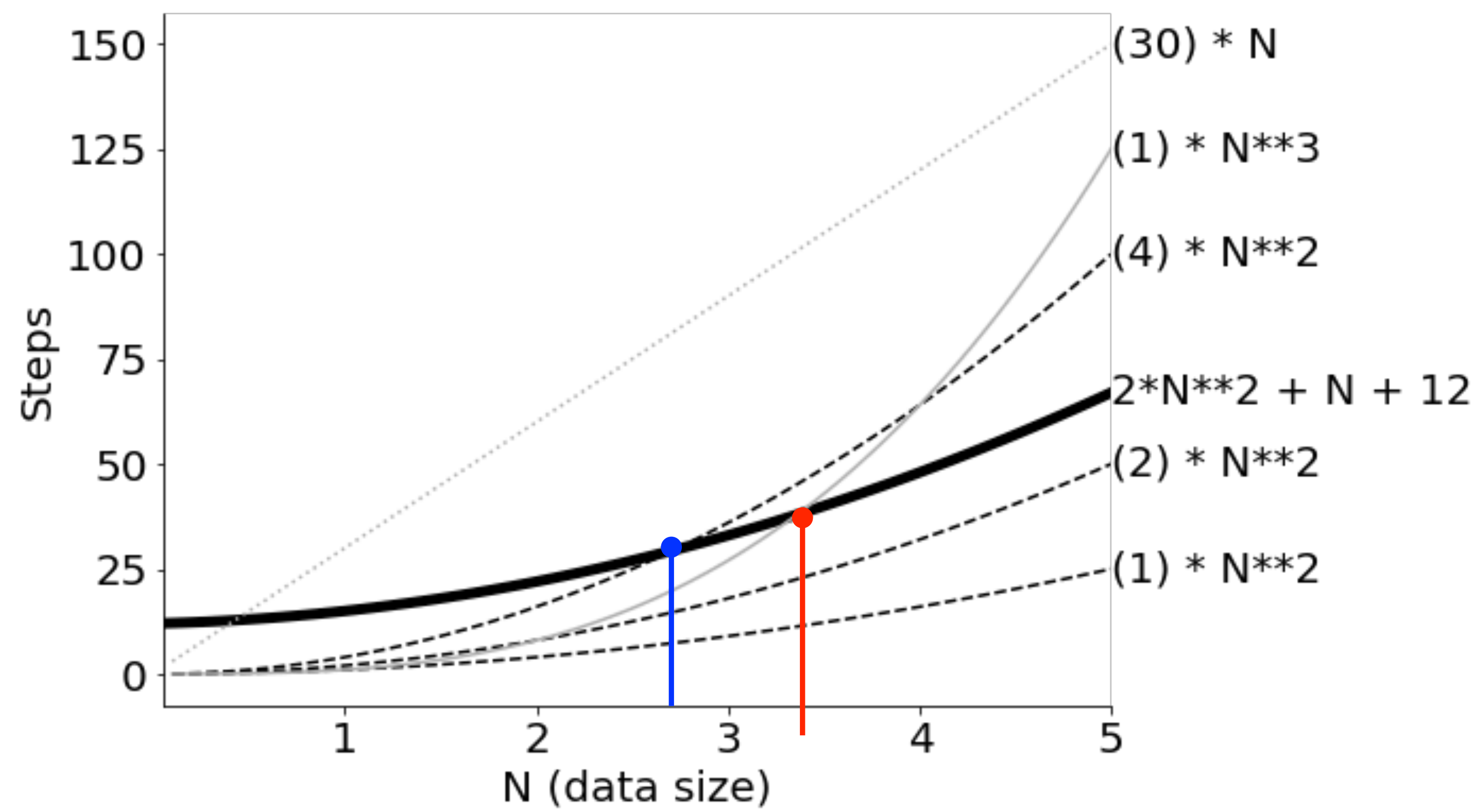
If we want to show $f(N) \in O(N^3)$, what is a good lower bound on N? Let's have C=1. $\mathbf{N \geq 4}$

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, $\mathbf{800 + 20 + 12 \not\leq 600}$.



2

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

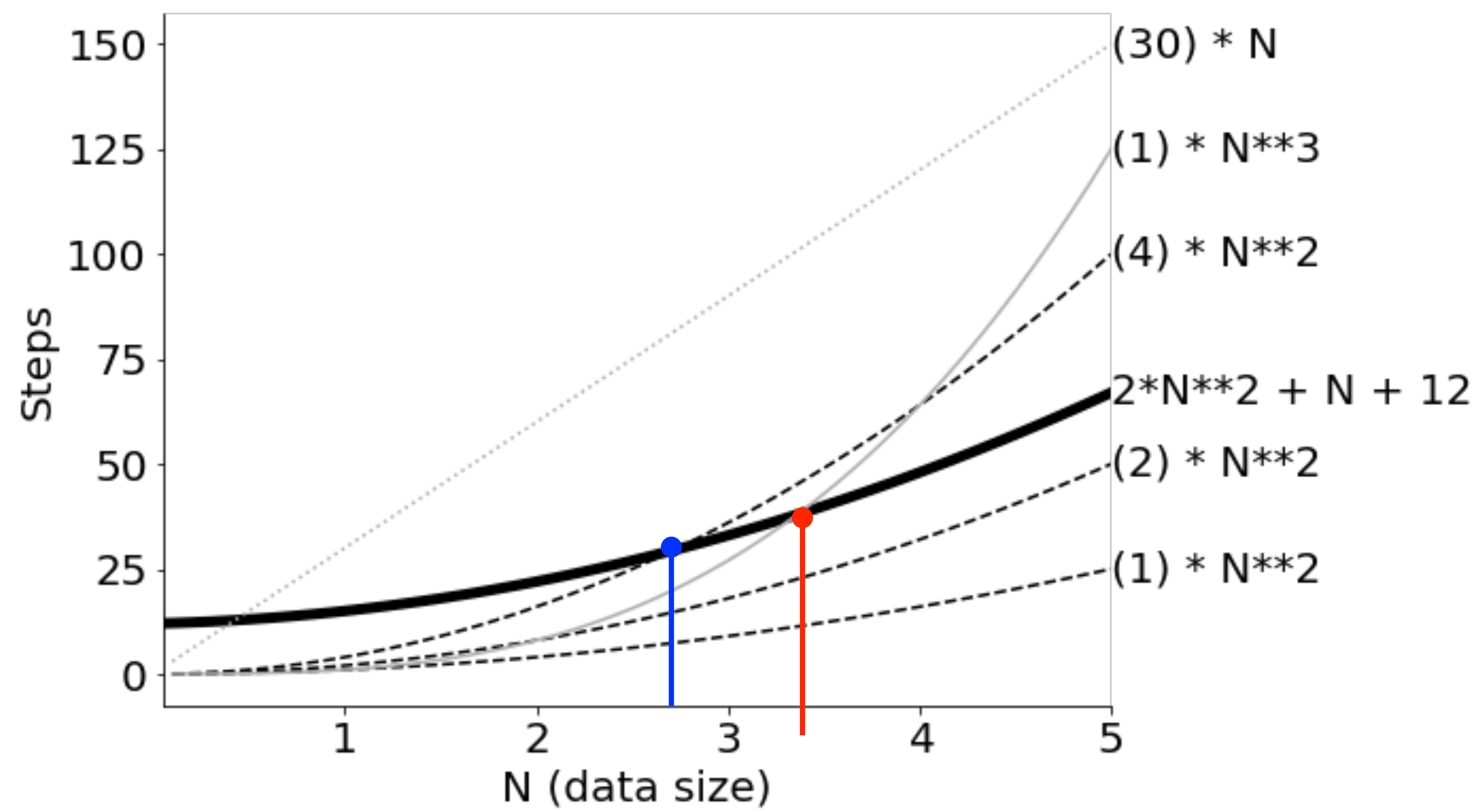
If we want to show $f(N) \in O(N^3)$, what is a good lower bound on N? Let's have C=1. $\mathbf{N \geq 4}$

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) O(N)
    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) O(N)
    if x > 2*avg:
        print("outlier", x)
```

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) O(N)
    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) O(N)
    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)
```

$len(A) = M$

$len(B) = N$

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(______________)

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

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

7

```
# assume L is already sorted, N=len(L)
def binary_search(L, target):
    left_idx = 0 # inclusive
    right_idx = len(L) # exclusive
    while right_idx - left_idx > 1:
        mid_idx = (right_idx + left_idx) // 2
        mid = L[mid_idx]
        if target >= mid:
            left_idx = mid_idx
        else:
            right_idx = mid_idx

    return right_idx > left_idx and L[left_idx] == target
```

how many times does this step run when N = 1? N = 2? N = 4? N = 8?

If **f(N)** is the number of times this step runs, then f(N) =

The complexity of binary search is O(________)

Assume target = 20

7

```
# assume L is already sorted, N=len(L)
def binary_search(L, target):
    left_idx = 0 # inclusive
    right_idx = len(L) # exclusive
    while right_idx - left_idx > 1:
        mid_idx = (right_idx + left_idx) // 2
        mid = L[mid_idx]
        if target >= mid:
            left_idx = mid_idx
        else:
            right_idx = mid_idx

    return right_idx > left_idx and L[left_idx] == target
```

how many times does this step run when N = 1? N = 2? N = 4? N = 8?

If **f(N)** is the number of times this step runs, then f(N) =

The complexity of binary search is O(________)

Assume target = 20

1	3	5	8	10	20	73	80

7

```
# assume L is already sorted, N=len(L)
def binary_search(L, target):
    left_idx = 0 # inclusive
    right_idx = len(L) # exclusive
    while right_idx - left_idx > 1:
        mid_idx = (right_idx + left_idx) // 2
        mid = L[mid_idx]
        if target >= mid:
            left_idx = mid_idx
        else:
            right_idx = mid_idx

    return right_idx > left_idx and L[left_idx] == target
```

how many times does this step run when N = 1? N = 2? N = 4? N = 8?

If **f(N)** is the number of times this step runs, then f(N) =

The complexity of binary search is O(_________)

Assume target = 20

Idx	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
	1	3	5	8	10	20	73	80	

7

```
# assume L is already sorted, N=len(L)
def binary_search(L, target):
    left_idx = 0 # inclusive
    right_idx = len(L) # exclusive
    while right_idx - left_idx > 1:
        mid_idx = (right_idx + left_idx) // 2
        mid = L[mid_idx]
        if target >= mid:
            left_idx = mid_idx
        else:
            right_idx = mid_idx

    return right_idx > left_idx and L[left_idx] == target
```

how many times does this step run when N = 1? N = 2? N = 4? N = 8?

If **f(N)** is the number of times this step runs, then f(N) =

The complexity of binary search is O(_________)

Assume target = 20

Idx	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
	1	3	5	8	10	20	73	80	

7

```
# assume L is already sorted, N=len(L)
def binary_search(L, target):
    left_idx = 0 # inclusive
    right_idx = len(L) # exclusive
    while right_idx - left_idx > 1:
        mid_idx = (right_idx + left_idx) // 2
        mid = L[mid_idx]
        if target >= mid:
            left_idx = mid_idx
        else:
            right_idx = mid_idx

    return right_idx > left_idx and L[left_idx] == target
```

how many times does this step run when N = 1? N = 2? N = 4? N = 8?

If **f(N)** is the number of times this step runs, then f(N) =

The complexity of binary search is O(_________)

Assume target = 20

Idx	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
	1	3	5	8	10	20	73	80	

7

```
# assume L is already sorted, N=len(L)
def binary_search(L, target):
    left_idx = 0 # inclusive
    right_idx = len(L) # exclusive
    while right_idx - left_idx > 1:
        mid_idx = (right_idx + left_idx) // 2
        mid = L[mid_idx]
        if target >= mid:
            left_idx = mid_idx
        else:
            right_idx = mid_idx

    return right_idx > left_idx and L[left_idx] == target
```

how many times does this step run when N = 1? N = 2? N = 4? N = 8?

If **f(N)** is the number of times this step runs, then f(N) =

The complexity of binary search is O(________)

Assume target = 20

Idx	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
	1	3	5	8	10	20	73	80	

7

```
# assume L is already sorted, N=len(L)
def binary_search(L, target):
    left_idx = 0 # inclusive
    right_idx = len(L) # exclusive
    while right_idx - left_idx > 1:
        mid_idx = (right_idx + left_idx) // 2
        mid = L[mid_idx]
        if target >= mid:
            left_idx = mid_idx
        else:
            right_idx = mid_idx

    return right_idx > left_idx and L[left_idx] == target
```

how many times does this step run when N = 1? N = 2? N = 4? N = 8?

If **f(N)** is the number of times this step runs, then f(N) =

The complexity of binary search is O(________)

Assume target = 20

Idx	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
	1	3	5	8	10	20	73	80	

7

```
# assume L is already sorted, N=len(L)
def binary_search(L, target):
    left_idx = 0 # inclusive
    right_idx = len(L) # exclusive
    while right_idx - left_idx > 1:
        mid_idx = (right_idx + left_idx) // 2
        mid = L[mid_idx]
        if target >= mid:
            left_idx = mid_idx
        else:
            right_idx = mid_idx

    return right_idx > left_idx and L[left_idx] == target
```

how many times does this step run when N = 1? N = 2? N = 4? N = 8?

If **f(N)** is the number of times this step runs, then f(N) =

The complexity of binary search is O(________)

Assume target = 20

Idx	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
	1	3	5	8	10	20	73	80	

7

```
# assume L is already sorted, N=len(L)
def binary_search(L, target):
    left_idx = 0 # inclusive
    right_idx = len(L) # exclusive
    while right_idx - left_idx > 1:
        mid_idx = (right_idx + left_idx) // 2
        mid = L[mid_idx]
        if target >= mid:
            left_idx = mid_idx
        else:
            right_idx = mid_idx

    return right_idx > left_idx and L[left_idx] == target
```

how many times does this step run when N = 1? N = 2? N = 4? N = 8?

If **f(N)** is the number of times this step runs, then f(N) =

The complexity of binary search is O(_________)

Assume target = 20

Idx	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
	1	3	5	8	10	20	73	80	

7

```
# assume L is already sorted, N=len(L)
def binary_search(L, target):
    left_idx = 0 # inclusive
    right_idx = len(L) # exclusive
    while right_idx - left_idx > 1:
        mid_idx = (right_idx + left_idx) // 2
        mid = L[mid_idx]
        if target >= mid:
            left_idx = mid_idx
        else:
            right_idx = mid_idx

    return right_idx > left_idx and L[left_idx] == target
```

how many times does this step run when N = 1? N = 2? N = 4? N = 8?

If **f(N)** is the number of times this step runs, then f(N) =

The complexity of binary search is O(_________)

Assume target = 20

Idx	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
	1	3	5	8	10	20	73	80	

7

```
# assume L is already sorted, N=len(L)
def binary_search(L, target):
    left_idx = 0 # inclusive
    right_idx = len(L) # exclusive
    while right_idx - left_idx > 1:
        mid_idx = (right_idx + left_idx) // 2
        mid = L[mid_idx]
        if target >= mid:
            left_idx = mid_idx
        else:
            right_idx = mid_idx

    return right_idx > left_idx and L[left_idx] == target
```

how many times does this step run when N = 1? N = 2? N = 4? N = 8?

If **f(N)** is the number of times this step runs, then f(N) =

The complexity of binary search is O(_________)

Assume target = 20

Idx	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
	1	3	5	8	10	20	73	80	
cut									

7

```
# assume L is already sorted, N=len(L)
def binary_search(L, target):
    left_idx = 0 # inclusive
    right_idx = len(L) # exclusive
    while right_idx - left_idx > 1:
        mid_idx = (right_idx + left_idx) // 2
        mid = L[mid_idx]
        if target >= mid:
            left_idx = mid_idx
        else:
            right_idx = mid_idx

    return right_idx > left_idx and L[left_idx] == target
```

how many times does this step run when N = 1? N = 2? N = 4? N = 8?

If **f(N)** is the number of times this step runs, then f(N) =

The complexity of binary search is O(________)

Assume target = 20

Idx	0	1	2	3	4	5	6	7	8
	1	3	5	8	10	20	73	80	
cut									
cut									

7

```
# assume L is already sorted, N=len(L)
def binary_search(L, target):
    left_idx = 0 # inclusive
    right_idx = len(L) # exclusive
    while right_idx - left_idx > 1:
        mid_idx = (right_idx + left_idx) // 2
        mid = L[mid_idx]
        if target >= mid:
            left_idx = mid_idx
        else:
            right_idx = mid_idx

    return right_idx > left_idx and L[left_idx] == target
```

how many times does this step run when N = 1? N = 2? N = 4? N = 8?

If **f(N)** is the number of times this step runs, then f(N) =

The complexity of binary search is O(_________)

Assume target = 20

Idx	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
	1	3	5	8	10	20	73	80	
cut									
cut									
cut									

7

```
# assume L is already sorted, N=len(L)
def binary_search(L, target):
    left_idx = 0 # inclusive
    right_idx = len(L) # exclusive
    while right_idx - left_idx > 1:
        mid_idx = (right_idx + left_idx) // 2
        mid = L[mid_idx]
        if target >= mid:
            left_idx = mid_idx
        else:
            right_idx = mid_idx

    return right_idx > left_idx and L[left_idx] == target
```

how many times does this step run when N = 1? N = 2? N = 4? N = 8?

3

If **f(N)** is the number of times this step runs, then f(N) =

The complexity of binary search is O(__________)

Assume target = 20

Idx	0	1	2	3	4	5	6	7	8
	1	3	5	8	10	20	73	80	
cut									
cut									
cut									

7

```
# assume L is already sorted, N=len(L)
def binary_search(L, target):
    left_idx = 0 # inclusive
    right_idx = len(L) # exclusive
    while right_idx - left_idx > 1:
        mid_idx = (right_idx + left_idx) // 2
        mid = L[mid_idx]
        if target >= mid:
            left_idx = mid_idx
        else:
            right_idx = mid_idx

    return right_idx > left_idx and L[left_idx] == target
```

how many times does this step run when N = 1? N = 2? N = 4? N = 8?

2 3

If **f(N)** is the number of times this step runs, then f(N) =

The complexity of binary search is O(________)

Assume target = 20

Idx	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
	1	3	5	8	10	20	73	80	
cut									
cut									
cut									

7

```python
# assume L is already sorted, N=len(L)
def binary_search(L, target):
    left_idx = 0 # inclusive
    right_idx = len(L) # exclusive
    while right_idx - left_idx > 1:
        mid_idx = (right_idx + left_idx) // 2
        mid = L[mid_idx]
        if target >= mid:
            left_idx = mid_idx
        else:
            right_idx = mid_idx

    return right_idx > left_idx and L[left_idx] == target
```

how many times does this step run when N = 1? N = 2? N = 4? N = 8?

1 2 3

If **f(N)** is the number of times this step runs, then f(N) =

The complexity of binary search is O(_________)

Assume target = 20

Idx	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
	1	3	5	8	10	20	73	80	
cut									
cut									
cut									

7

```
# assume L is already sorted, N=len(L)
def binary_search(L, target):
    left_idx = 0 # inclusive
    right_idx = len(L) # exclusive
    while right_idx - left_idx > 1:
        mid_idx = (right_idx + left_idx) // 2
        mid = L[mid_idx]
        if target >= mid:
            left_idx = mid_idx
        else:
            right_idx = mid_idx

    return right_idx > left_idx and L[left_idx] == target
```

how many times does this step run when N = 1? N = 2? N = 4? N = 8?

0 1 2 3

If **f(N)** is the number of times this step runs, then f(N) =

The complexity of binary search is O(_________)

Assume target = 20

Idx	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
	1	3	5	8	10	20	73	80	
cut									
cut									
cut									

7

```
# assume L is already sorted, N=len(L)
def binary_search(L, target):
    left_idx = 0 # inclusive
    right_idx = len(L) # exclusive
    while right_idx - left_idx > 1:
        mid_idx = (right_idx + left_idx) // 2
        mid = L[mid_idx]
        if target >= mid:
            left_idx = mid_idx
        else:
            right_idx = mid_idx

    return right_idx > left_idx and L[left_idx] == target
```

how many times does this step run when N = 1? N = 2? N = 4? N = 8?

0 1 2 3

If **f(N)** is the number of times this step runs, then f(N) = $\mathbf{\log_2 N}$

The complexity of binary search is O(_________)

Assume target = 20

Idx	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
	1	3	5	8	10	20	73	80	
cut									
cut									
cut									

7

```python
# assume L is already sorted, N=len(L)
def binary_search(L, target):
    left_idx = 0 # inclusive
    right_idx = len(L) # exclusive
    while right_idx - left_idx > 1:
        mid_idx = (right_idx + left_idx) // 2
        mid = L[mid_idx]
        if target >= mid:
            left_idx = mid_idx
        else:
            right_idx = mid_idx

    return right_idx > left_idx and L[left_idx] == target
```

how many times does this step run when N = 1? N = 2? N = 4? N = 8?

0 1 2 3

If **f(N)** is the number of times this step runs, then f(N) = **$\log_2 N$**

The complexity of binary search is O(**logN**)

Assume target = 20

Idx	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
	1	3	5	8	10	20	73	80	
cut									
cut									
cut									

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)

Examples, merge sort, quick sort
```

assumed sorted is O(N log N)

what is the complexity of version A?

what is the complexity of version B?

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)

Examples, merge sort, quick sort
```

assumed sorted is O(N log N)

what is the complexity of version A?

what is the complexity of version B?

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
```

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

Examples, merge sort, quick sort
```

assumed sorted is O(N log N)

what is the complexity of version A?

what is the complexity of version B?

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

N choices	N-1 choices	N-2 choices	...	2 choices	1 choice

```
# 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)

Examples, merge sort, quick sort
```

assumed sorted is O(N log N)

what is the complexity of version A?

what is the complexity of version B?

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
```

N choices	N-1 choices	N-2 choices	. . .	2 choices	1 choice

Total choices = N*(N-1)* (N-2)*. . .*2*1

```
# 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)

Examples, merge sort, quick sort
```

assumed sorted is O(N log N)

what is the complexity of version A?

what is the complexity of version B?

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

N choices	N-1 choices	N-2 choices	. . .	2 choices	1 choice

Total choices = N*(N-1)* (N-2)*. . .*2*1 = 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)

Examples, merge sort, quick sort
```

assumed sorted is O(N log N)

what is the complexity of version A?

what is the complexity of version B?

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

N choices	N-1 choices	N-2 choices	. . .	2 choices	1 choice

Total choices = N*(N-1)* (N-2)*. . .*2*1 = N!
Therefore, total permutations for (A,B,C) = 3*2*1

```
# 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)

Examples, merge sort, quick sort
```

assumed sorted is O(N log N)

what is the complexity of version A?

what is the complexity of version B?

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

N choices	N-1 choices	N-2 choices	. . .	2 choices	1 choice

Total choices = N*(N-1)* (N-2)*. . .*2*1 = N!
Therefore, total permutations for (A,B,C) = 3*2*1 = 3!

```
# 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)

Examples, merge sort, quick sort
```

assumed sorted is O(N log N)

what is the complexity of version A?

what is the complexity of version B?

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

N choices	N-1 choices	N-2 choices	. . .	2 choices	1 choice

Total choices = N*(N-1)* (N-2)*. . .*2*1 = N!
Therefore, total permutations for (A,B,C) = 3*2*1 = 3!

```
# version A
import itertools

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

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

Examples, merge sort, quick sort
```

assumed sorted is O(N log N)

what is the complexity of version A?

what is the complexity of version B?

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

N choices	N-1 choices	N-2 choices	. . .	2 choices	1 choice

Total choices = N*(N-1)* (N-2)*. . .*2*1 = N!
Therefore, total permutations for (A,B,C) = 3*2*1 = 3!

```
# 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)

Examples, merge sort, quick sort
```

assumed sorted is O(N log N)

what is the complexity of version A?

what is the complexity of version B?

8

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

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

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

ABC BCA
ACB CAB
BAC CBA

N choices	N-1 choices	N-2 choices	...	2 choices	1 choice

Total choices = N*(N-1)* (N-2)*. . .*2*1 = N!
Therefore, total permutations for (A,B,C) = 3*2*1 = 3!

```
# 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)

Examples, 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?

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

N choices	N-1 choices	N-2 choices	. . .	2 choices	1 choice

Total choices = N*(N-1)* (N-2)*. . .*2*1 = N!
Therefore, total permutations for (A,B,C) = 3*2*1 = 3!

```
# 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)        O(N logN)
s2 = sorted(s2)        O(N logN)
matches = (s1 == s2)

Examples, 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?

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

N choices	N-1 choices	N-2 choices	. . .	2 choices	1 choice

Total choices = N*(N-1)* (N-2)*. . .*2*1 = N!
Therefore, total permutations for (A,B,C) = 3*2*1 = 3!

```
# 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)        O(N logN)
s2 = sorted(s2)        O(N logN)
matches = (s1 == s2) O(N)

Examples, 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?

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

N choices	N-1 choices	N-2 choices	. . .	2 choices	1 choice

Total choices = N*(N-1)* (N-2)*. . .*2*1 = N!
Therefore, total permutations for (A,B,C) = 3*2*1 = 3!

```
# 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)       O(N logN)
s2 = sorted(s2)       O(N logN)
matches = (s1 == s2)  O(N)

Examples, 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? $O(N \log N + N \log N + N)$

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

N choices	N-1 choices	N-2 choices	. . .	2 choices	1 choice

Total choices = N*(N-1)* (N-2)*. . .*2*1 = N!
Therefore, total permutations for (A,B,C) = 3*2*1 = 3!

```
# 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)        O(N logN)
s2 = sorted(s2)        O(N logN)
matches = (s1 == s2)   O(N)

Examples, 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? $O(\mathbf{N}\log\mathbf{N} + \mathbf{N}\log\mathbf{N} + \mathbf{N}) = O(\mathbf{2N}\log\mathbf{N} + \mathbf{N})$

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

N choices	N-1 choices	N-2 choices	. . .	2 choices	1 choice

Total choices = N*(N-1)* (N-2)*. . .*2*1 = N!
Therefore, total permutations for (A,B,C) = 3*2*1 = 3!

```
# 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)        O(N logN)
s2 = sorted(s2)        O(N logN)
matches = (s1 == s2)   O(N)

Examples, 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? $O(N \log N + N \log N + N) = O(2N \log N + N) = O(N \log N)$

8

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

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

ABC BCA
ACB CAB
BAC CBA

N choices	N-1 choices	N-2 choices	...	2 choices	1 choice

Total choices = N*(N-1)* (N-2)*. . .*2*1 = N!
Therefore, total permutations for (A,B,C) = 3*2*1 = 3!

```
# 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)       O(N logN)
s2 = sorted(s2)       O(N logN)
matches = (s1 == s2)  O(N)

Examples, 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(N logN)

9

```
def selection_sort(L):
    for i in range(len(L)):
        idx_min = i
        for j in range(i, len(L)):
            if L[j] < L[idx_min]:
                idx_min = j
        # swap values at i and idx_min
        L[idx_min], L[i] = L[i], L[idx_min]

nums = [2, 4, 3, 1]
selection_sort(nums)
print(nums)
```

if this runs **f(N)** times, where **N=len(L)**,

then **f(N)** =

The complexity of selection sort is

O(_____________)

9

```
def selection_sort(L):
    for i in range(len(L)):
        idx_min = i
        for j in range(i, len(L)):
            if L[j] < L[idx_min]:
                idx_min = j
        # swap values at i and idx_min
        L[idx_min], L[i] = L[i], L[idx_min]

nums = [2, 4, 3, 1]
selection_sort(nums)
print(nums)
```

if this runs **f(N)** times, where **N=len(L)**,

then **f(N)** =

The complexity of selection sort is

O(____________)

i	**# of items for the inner for loop**
0	N
1	N-1
2	N-2
...	...
N-1	1
N	0

9

```
def selection_sort(L):
    for i in range(len(L)):
        idx_min = i
        for j in range(i, len(L)):
            if L[j] < L[idx_min]:
                idx_min = j
        # swap values at i and idx_min
        L[idx_min], L[i] = L[i], L[idx_min]

nums = [2, 4, 3, 1]
selection_sort(nums)
print(nums)
```

if this runs **f(N)** times, where **N=len(L)**,

then **f(N)** = $\mathbf{N + (N-1) + (N-2) + \ldots + 2 + 1 + 0}$

The complexity of selection sort is

O(______________)

i	**# of items for the inner for loop**
0	N
1	N-1
2	N-2
...	...
N-1	1
N	0

9

```
def selection_sort(L):
    for i in range(len(L)):
        idx_min = i
        for j in range(i, len(L)):
            if L[j] < L[idx_min]:
                idx_min = j
        # swap values at i and idx_min
        L[idx_min], L[i] = L[i], L[idx_min]

nums = [2, 4, 3, 1]
selection_sort(nums)
print(nums)
```

if this runs **f(N)** times, where **N=len(L)**,

then **f(N)** $= \mathbf{N + (N-1) + (N-2) + ... + 2 + 1 + 0}$

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

The complexity of selection sort is

O(______________)

i	**# of items for the inner for loop**
0	N
1	N-1
2	N-2
...	...
N-1	1
N	0

9

```
def selection_sort(L):
    for i in range(len(L)):
        idx_min = i
        for j in range(i, len(L)):
            if L[j] < L[idx_min]:
                idx_min = j
        # swap values at i and idx_min
        L[idx_min], L[i] = L[i], L[idx_min]

nums = [2, 4, 3, 1]
selection_sort(nums)
print(nums)
```

if this runs **f(N)** times, where **N=len(L)**,

then **f(N)** = $\mathbf{N + (N-1) + (N-2) + \ldots + 2 + 1 + 0}$

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

The complexity of selection sort is

O(____________)

i	**# of items for the inner for loop**
0	N
1	N-1
2	N-2
...	...
N-1	1
N	0

9

```
def selection_sort(L):
    for i in range(len(L)):
        idx_min = i
        for j in range(i, len(L)):
            if L[j] < L[idx_min]:
                idx_min = j
        # swap values at i and idx_min
        L[idx_min], L[i] = L[i], L[idx_min]

nums = [2, 4, 3, 1]
selection_sort(nums)
print(nums)
```

if this runs **f(N)** times, where **N=len(L)**,

then **f(N)** $= \mathbf{N + (N-1) + (N-2) + \ldots + 2 + 1 + 0}$

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

The complexity of selection sort is

O(____________)

i	**# of items for the inner for loop**
0	N
1	N-1
2	N-2
...	...
N-1	1
N	0

9

```
def selection_sort(L):
    for i in range(len(L)):
        idx_min = i
        for j in range(i, len(L)):
            if L[j] < L[idx_min]:
                idx_min = j
        # swap values at i and idx_min
        L[idx_min], L[i] = L[i], L[idx_min]

nums = [2, 4, 3, 1]
selection_sort(nums)
print(nums)
```

if this runs f(N) times, where N=len(L),

then f(N) $= \mathbf{N + (N-1) + (N-2) + \ldots + 2 + 1 + 0}$

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

The complexity of selection sort is

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

i	# of items for the inner for loop
0	N
1	N-1
2	N-2
...	...
N-1	1
N	0

9

```
def selection_sort(L):
    for i in range(len(L)):
        idx_min = i
        for j in range(i, len(L)):
            if L[j] < L[idx_min]:
                idx_min = j
        # swap values at i and idx_min
        L[idx_min], L[i] = L[i], L[idx_min]

nums = [2, 4, 3, 1]
selection_sort(nums)
print(nums)
```

if this runs **f(N)** times, where **N=len(L)**,

then **f(N)** $= \mathbf{N + (N-1) + (N-2) + ... + 2 + 1 + 0}$

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

The complexity of selection sort is

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

i	**# of items for the inner for loop**
0	N
1	N-1
2	N-2
...	...
N-1	1
N	0

9

```python
def selection_sort(L):
    for i in range(len(L)):
        idx_min = i
        for j in range(i, len(L)):
            if L[j] < L[idx_min]:
                idx_min = j
        # swap values at i and idx_min
        L[idx_min], L[i] = L[i], L[idx_min]

nums = [2, 4, 3, 1]
selection_sort(nums)
print(nums)
```

if this runs **f(N)** times, where **N=len(L)**,

then **f(N)** $= \mathbf{N + (N-1) + (N-2) + ... + 2 + 1 + 0}$

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

The complexity of selection sort is

$$O\left(\frac{\mathbf{N^2}}{\mathbf{2}} + \frac{\mathbf{N}}{\mathbf{2}}\right) = O\left(\frac{\mathbf{N^2}}{\mathbf{2}}\right) = O(\mathbf{N^2})$$

i	# of items for the inner for loop
0	N
1	N-1
2	N-2
...	...
N-1	1
N	0