Efficiency

Class outline:

- Exponentiation
- Orders of Growth
- Memoization

Exponentiation

Exponentiation approach #1

Based on this recursive definition:

$$b^n = \begin{cases} 1 & \text{if } n = 0 \\ b \cdot b^{(n-1)} & \text{otherwise} \end{cases}$$

```
def exp(b, n):
  if n == 0:
    return 1
  else:
    return b * exp(b, n-1)
```

How many calls are required to calculate `exp(2, 16)`?

Exponentiation approach #1

Based on this recursive definition:

$$b^n = \begin{cases} 1 & \text{if } n = 0 \\ b \cdot b^{(n-1)} & \text{otherwise} \end{cases}$$

```python
def exp(b, n):
  if n == 0:
    return 1
  else:
    return b * exp(b, n-1)
```

How many calls are required to calculate `exp(2, 16)`?

Can we do better?

Exponentiation approach #2

Based on this alternate definition:

$$b^n = \begin{cases} 1 & \text{if } n = 0 \\ (b^{\frac{1}{2}n})^2 & \text{if } n \text{ is even} \\ b \cdot b^{(n-1)} & \text{if } n \text{ is odd} \end{cases}$$

```
def exp_fast(b, n):
    if n == 0:
        return 1
    elif n % 2 == 0:
        return square(exp_fast(b, n//2))
    else:
        return b * exp_fast(b, n-1)

square = lambda x: x * x
```

How many calls are required to calculate `exp(2, 16)`?

Exponentiation approach #2

Based on this alternate definition:

$$b^n = \begin{cases} 1 & \text{if } n = 0 \\ (b^{\frac{1}{2}n})^2 & \text{if } n \text{ is even} \\ b \cdot b^{(n-1)} & \text{if } n \text{ is odd} \end{cases}$$

```
def exp_fast(b, n):
    if n == 0:
        return 1
    elif n % 2 == 0:
        return square(exp_fast(b, n//2))
    else:
        return b * exp_fast(b, n-1)

square = lambda x: x * x
```

How many calls are required to calculate `exp(2, 16)`?

Some algorithms are more efficient than others!

Orders of Growth

Common orders of growth

One way to describe the efficiency of an algorithm according to its **order of growth**, the effect of increasing the size of input on the number of steps required.

Order of growth	**Description**
Constant growth	Always takes same number of steps, regardless of input size.
Logarithmic growth	Number of steps increases proportionally to the logarithm of the input size.
Linear growth	Number of steps increases in direct proportion to the input size.
Quadratic growth	Number of steps increases in proportion to the square of the input size.
Exponential growth	Number of steps increases faster than a polynomial function of the input size.

Adding to the front of linked list

```
def insert_front(linked_list, new_val):
    """Inserts NEW_VAL in front of LINKED_LIST,returning new linked list.
    >>> ll = Link(1, Link(3, Link(5)))
    >>> insert_front(ll, 0)
    Link(0, Link(1, Link(3, Link(5))))
    """
    return Link(new_val, linked_list)
```

How many operations will this require for increasing lengths of the list?

List size	Operations
1	
10	
100	
1000	

Adding to the front of linked list

```
def insert_front(linked_list, new_val):
    """Inserts NEW_VAL in front of LINKED_LIST,returning new linked list.
    >>> ll = Link(1, Link(3, Link(5)))
    >>> insert_front(ll, 0)
    Link(0, Link(1, Link(3, Link(5))))
    """
    return Link(new_val, linked_list)
```

How many operations will this require for increasing lengths of the list?

List size	Operations
1	1
10	
100	
1000	

Adding to the front of linked list

```
def insert_front(linked_list, new_val):
    """Inserts NEW_VAL in front of LINKED_LIST,returning new linked list.
    >>> ll = Link(1, Link(3, Link(5)))
    >>> insert_front(ll, 0)
    Link(0, Link(1, Link(3, Link(5))))
    """
    return Link(new_val, linked_list)
```

How many operations will this require for increasing lengths of the list?

List size	Operations
1	1
10	1
100	
1000	

Adding to the front of linked list

```
def insert_front(linked_list, new_val):
    """Inserts NEW_VAL in front of LINKED_LIST,returning new linked list.
    >>> ll = Link(1, Link(3, Link(5)))
    >>> insert_front(ll, 0)
    Link(0, Link(1, Link(3, Link(5))))
    """
    return Link(new_val, linked_list)
```

How many operations will this require for increasing lengths of the list?

List size	Operations
1	1
10	1
100	1
1000	

Adding to the front of linked list

```
def insert_front(linked_list, new_val):
    """Inserts NEW_VAL in front of LINKED_LIST,returning new linked list.
    >>> ll = Link(1, Link(3, Link(5)))
    >>> insert_front(ll, 0)
    Link(0, Link(1, Link(3, Link(5))))
    """
    return Link(new_val, linked_list)
```

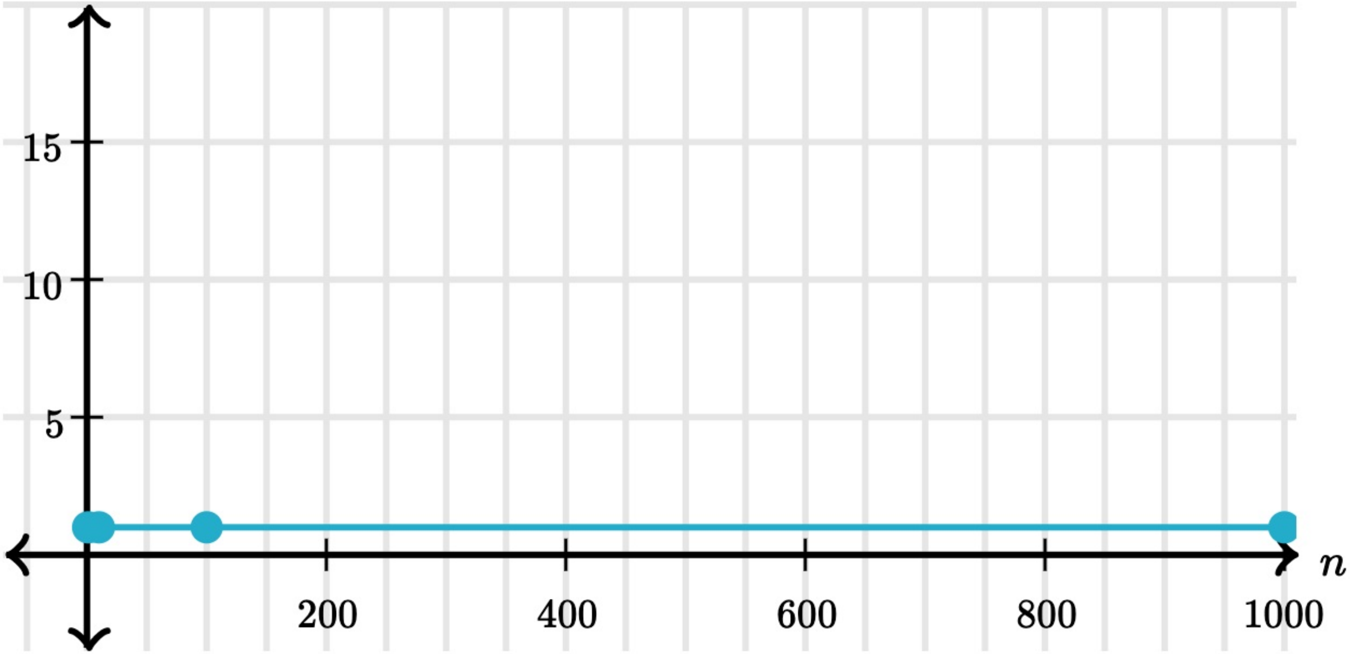
How many operations will this require for increasing lengths of the list?

List size	Operations
1	1
10	1
100	1
1000	1

Constant time

An algorithm that takes **constant time**, always makes a fixed number of operations regardless of the input size.

List size	Operations
1	1
10	1
100	1
1000	1


15
10
5
200
400
600
800
1000
n

Fast exponentiation

```
def exp_fast(b, n):
    if n == 0:
        return 1
    elif n % 2 == 0:
        return square(exp_fast(b, n//2))
    else:
        return b * exp_fast(b, n-1)

square = lambda x: x * x
```

How many operations will this require for increasing values of n?

N	Operations
0	
8	
16	
1024	

Fast exponentiation

```
def exp_fast(b, n):
    if n == 0:
        return 1
    elif n % 2 == 0:
        return square(exp_fast(b, n//2))
    else:
        return b * exp_fast(b, n-1)

square = lambda x: x * x
```

How many operations will this require for increasing values of n?

N	Operations
0	1
8	
16	
1024	

Fast exponentiation

```
def exp_fast(b, n):
    if n == 0:
        return 1
    elif n % 2 == 0:
        return square(exp_fast(b, n//2))
    else:
        return b * exp_fast(b, n-1)

square = lambda x: x * x
```

How many operations will this require for increasing values of n?

N	Operations
0	1
8	5
16	
1024	

Fast exponentiation

```
def exp_fast(b, n):
    if n == 0:
        return 1
    elif n % 2 == 0:
        return square(exp_fast(b, n//2))
    else:
        return b * exp_fast(b, n-1)

square = lambda x: x * x
```

How many operations will this require for increasing values of n?

N	Operations
0	1
8	5
16	6
1024	

Fast exponentiation

```
def exp_fast(b, n):
    if n == 0:
        return 1
    elif n % 2 == 0:
        return square(exp_fast(b, n//2))
    else:
        return b * exp_fast(b, n-1)

square = lambda x: x * x
```

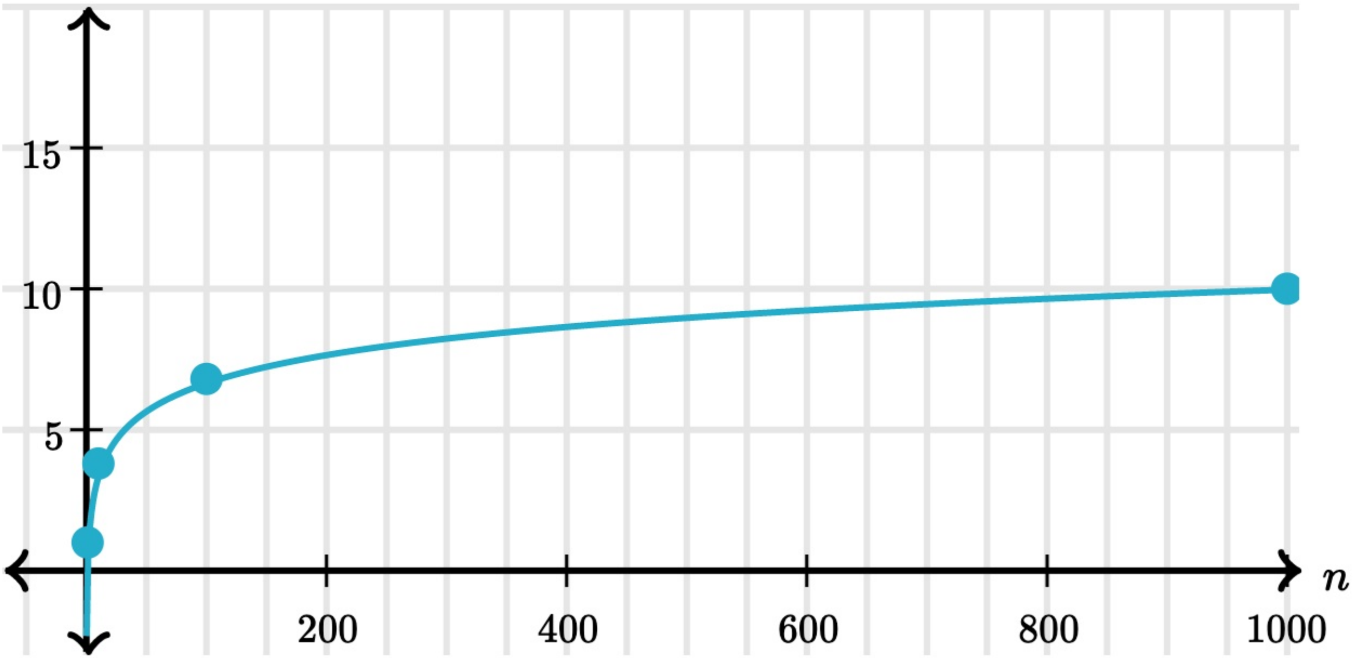
How many operations will this require for increasing values of n?

N	Operations
0	1
8	5
16	6
1024	12

Logarithmic time

When an algorithm takes **logarithmic time**, the time that it takes increases proportionally to the logarithm of the input size.

N	Operations
0	1
8	5
16	6
1024	12



15
10
5
200
400
600
800
1000
n

Finding value in a linked list

```
def find_in_link(ll, value):
    """Return true if linked list LL contains VALUE.
    >>> find_in_link(Link(3, Link(4, Link(5))), 4)
    True
    >>> find_in_link(Link(3, Link(4, Link(5))), 7)
    False
    """
    if ll is Link.empty:
        return False
    elif ll.first == value:
        return True
    return find_in_link(ll.rest, value)
```

How many operations will this require for increasing lengths of the list? Consider both the **best case** and **worst case**.

List size	Best case: Operations	Worst case: Operations
1		
10		
100		
1000		

Finding value in a linked list

```
def find_in_link(ll, value):
    """Return true if linked list LL contains VALUE.
    >>> find_in_link(Link(3, Link(4, Link(5))), 4)
    True
    >>> find_in_link(Link(3, Link(4, Link(5))), 7)
    False
    """
    if ll is Link.empty:
        return False
    elif ll.first == value:
        return True
    return find_in_link(ll.rest, value)
```

How many operations will this require for increasing lengths of the list? Consider both the **best case** and **worst case**.

List size	Best case: Operations	Worst case: Operations
1	1	
10		
100		
1000		

Finding value in a linked list

```
def find_in_link(ll, value):
    """Return true if linked list LL contains VALUE.
    >>> find_in_link(Link(3, Link(4, Link(5))), 4)
    True
    >>> find_in_link(Link(3, Link(4, Link(5))), 7)
    False
    """
    if ll is Link.empty:
        return False
    elif ll.first == value:
        return True
    return find_in_link(ll.rest, value)
```

How many operations will this require for increasing lengths of the list? Consider both the **best case** and **worst case**.

List size	Best case: Operations	Worst case: Operations
1	1	1
10		
100		
1000		

Finding value in a linked list

```
def find_in_link(ll, value):
    """Return true if linked list LL contains VALUE.
    >>> find_in_link(Link(3, Link(4, Link(5))), 4)
    True
    >>> find_in_link(Link(3, Link(4, Link(5))), 7)
    False
    """
    if ll is Link.empty:
        return False
    elif ll.first == value:
        return True
    return find_in_link(ll.rest, value)
```

How many operations will this require for increasing lengths of the list? Consider both the **best case** and **worst case**.

List size	Best case: Operations	Worst case: Operations
1	1	1
10	1	
100		
1000		

Finding value in a linked list

```
def find_in_link(ll, value):
    """Return true if linked list LL contains VALUE.
    >>> find_in_link(Link(3, Link(4, Link(5))), 4)
    True
    >>> find_in_link(Link(3, Link(4, Link(5))), 7)
    False
    """
    if ll is Link.empty:
        return False
    elif ll.first == value:
        return True
    return find_in_link(ll.rest, value)
```

How many operations will this require for increasing lengths of the list? Consider both the **best case** and **worst case**.

List size	Best case: Operations	Worst case: Operations
1	1	1
10	1	10
100		
1000		

Finding value in a linked list

```
def find_in_link(ll, value):
    """Return true if linked list LL contains VALUE.
    >>> find_in_link(Link(3, Link(4, Link(5))), 4)
    True
    >>> find_in_link(Link(3, Link(4, Link(5))), 7)
    False
    """
    if ll is Link.empty:
        return False
    elif ll.first == value:
        return True
    return find_in_link(ll.rest, value)
```

How many operations will this require for increasing lengths of the list? Consider both the **best case** and **worst case**.

List size	Best case: Operations	Worst case: Operations
1	1	1
10	1	10
100	1	
1000		

Finding value in a linked list

```
def find_in_link(ll, value):
    """Return true if linked list LL contains VALUE.
    >>> find_in_link(Link(3, Link(4, Link(5))), 4)
    True
    >>> find_in_link(Link(3, Link(4, Link(5))), 7)
    False
    """
    if ll is Link.empty:
        return False
    elif ll.first == value:
        return True
    return find_in_link(ll.rest, value)
```

How many operations will this require for increasing lengths of the list? Consider both the **best case** and **worst case**.

List size	Best case: Operations	Worst case: Operations
1	1	1
10	1	10
100	1	100
1000		

Finding value in a linked list

```
def find_in_link(ll, value):
    """Return true if linked list LL contains VALUE.
    >>> find_in_link(Link(3, Link(4, Link(5))), 4)
    True
    >>> find_in_link(Link(3, Link(4, Link(5))), 7)
    False
    """
    if ll is Link.empty:
        return False
    elif ll.first == value:
        return True
    return find_in_link(ll.rest, value)
```

How many operations will this require for increasing lengths of the list? Consider both the **best case** and **worst case**.

List size	Best case: Operations	Worst case: Operations
1	1	1
10	1	10
100	1	100
1000	1	

Finding value in a linked list

```
def find_in_link(ll, value):
    """Return true if linked list LL contains VALUE.
    >>> find_in_link(Link(3, Link(4, Link(5))), 4)
    True
    >>> find_in_link(Link(3, Link(4, Link(5))), 7)
    False
    """
    if ll is Link.empty:
        return False
    elif ll.first == value:
        return True
    return find_in_link(ll.rest, value)
```

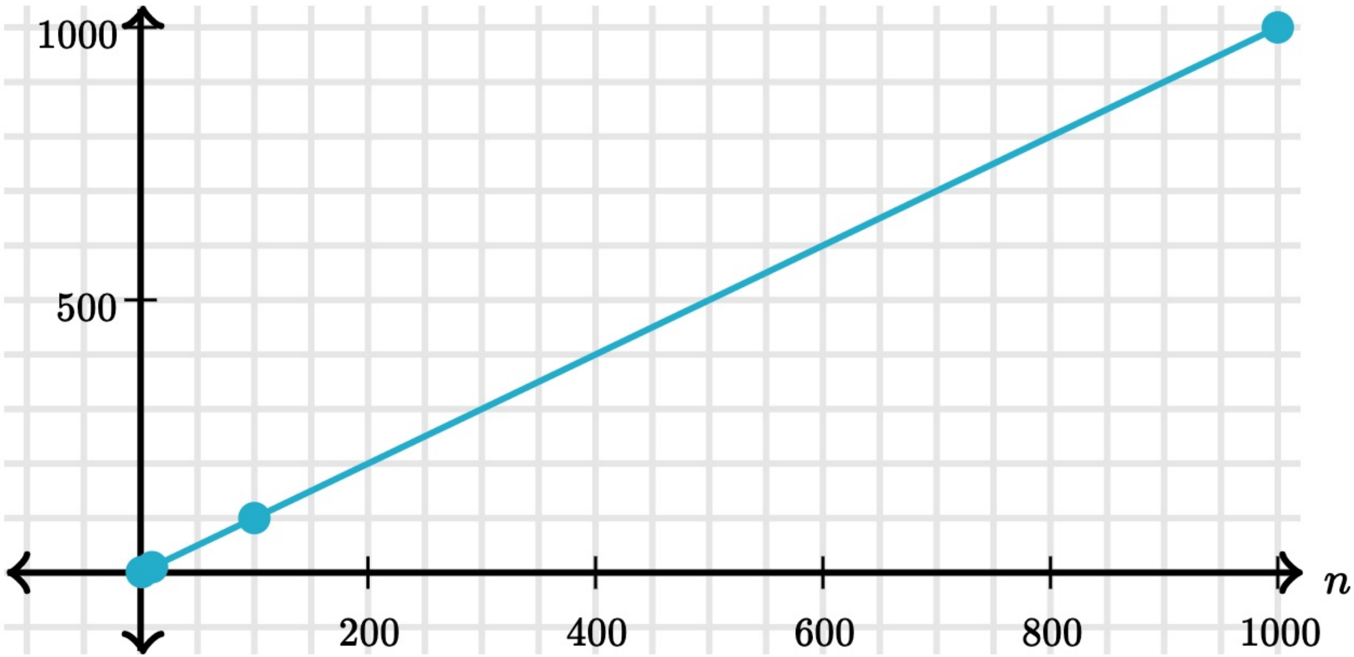
How many operations will this require for increasing lengths of the list? Consider both the **best case** and **worst case**.

List size	Best case: Operations	Worst case: Operations
1	1	1
10	1	10
100	1	100
1000	1	1000

Linear time

When an algorithm takes **linear time**, its number of operations increases in direct proportion to the input size.

List size	Worst case: Operations
1	1
10	10
100	100
100	1000


1000
500
200
400
600
800
1000
n

Counting overlapping items in lists

```
def overlap(a, b):
    """
    >>> overlap([3, 5, 7, 6], [4, 5, 6, 5])
    3
    """
    count = 0
    for item in a:
        for other in b:
            if item == other:
                count += 1
    return count
```

	3	5	6	7
4				
5		+		
6				+
5		+		

How many operations are required for increasing lengths of the lists?

List size	Operations

1
10
100
1000

Counting overlapping items in lists

```python
def overlap(a, b):
    """
    >>> overlap([3, 5, 7, 6], [4, 5, 6, 5])
    3
    """
    count = 0
    for item in a:
        for other in b:
            if item == other:
                count += 1
    return count
```

	3	5	6	7
4				
5		+		
6				+
5		+		

How many operations are required for increasing lengths of the lists?

List size	Operations

1	1
10	
100	
1000	

Counting overlapping items in lists

```
def overlap(a, b):
    """
    >>> overlap([3, 5, 7, 6], [4, 5, 6, 5])
    3
    """
    count = 0
    for item in a:
        for other in b:
            if item == other:
                count += 1
    return count
```

	3	5	6	7
4				
5		+		
6				+
5		+		

How many operations are required for increasing lengths of the lists?

List size	Operations

1	1
10	100
100	
1000	

Counting overlapping items in lists

```python
def overlap(a, b):
    """
    >>> overlap([3, 5, 7, 6], [4, 5, 6, 5])
    3
    """
    count = 0
    for item in a:
        for other in b:
            if item == other:
                count += 1
    return count
```

	3	5	6	7
4				
5		+		
6				+
5		+		

How many operations are required for increasing lengths of the lists?

List size	Operations

1	1
10	100
100	10000
1000	

Counting overlapping items in lists

```
def overlap(a, b):
    """
    >>> overlap([3, 5, 7, 6], [4, 5, 6, 5])
    3
    """
    count = 0
    for item in a:
        for other in b:
            if item == other:
                count += 1
    return count
```

	3	5	6	7
4				
5		+		
6				+
5		+		

How many operations are required for increasing lengths of the lists?

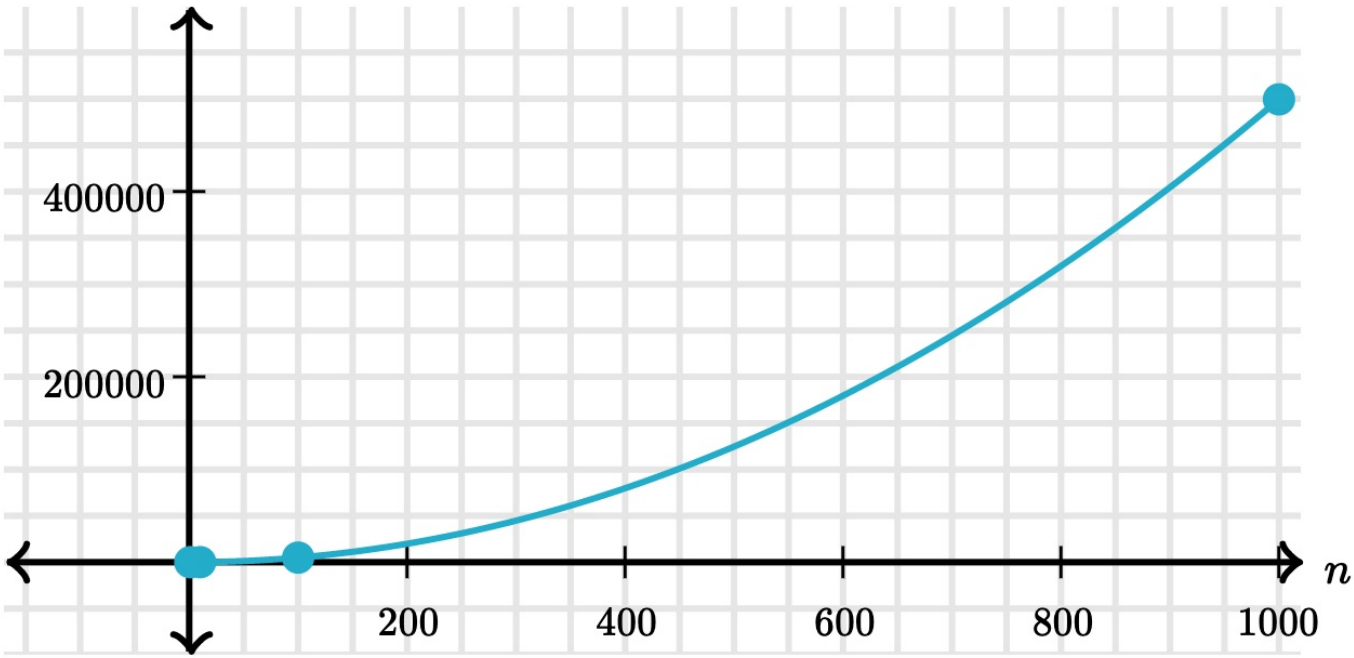
List size	**Operations**

1	1
10	100
100	10000
1000	1000000

Quadratic time

When an algorithm grows in **quadratic time**, its steps increase in proportion to square of the input size.

List size	**Operations**
1	1
10	100
100	10000
1000	1000000


400000
200000
200
400
600
800
1000
n

Recursive Virahanka-Fibonacci

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

How many operations are required for increasing values of n?

N	Operations
1	
2	
3	
4	
7	
8	
20	

Recursive Virahanka-Fibonacci

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

How many operations are required for increasing values of n?

N	Operations
1	1
2	
3	
4	
7	
8	
20	

Recursive Virahanka-Fibonacci

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

How many operations are required for increasing values of n?

N	Operations
1	1
2	3
3	
4	
7	
8	
20	

Recursive Virahanka-Fibonacci

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

How many operations are required for increasing values of n?

N	Operations
1	1
2	3
3	5
4	
7	
8	
20	

Recursive Virahanka-Fibonacci

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

How many operations are required for increasing values of n?

N	Operations
1	1
2	3
3	5
4	9
7	
8	
20	

Recursive Virahanka-Fibonacci

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

How many operations are required for increasing values of n?

N	Operations
1	1
2	3
3	5
4	9
7	41
8	
20	

Recursive Virahanka-Fibonacci

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

How many operations are required for increasing values of n?

N	Operations
1	1
2	3
3	5
4	9
7	41
8	67
20	

Recursive Virahanka-Fibonacci

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

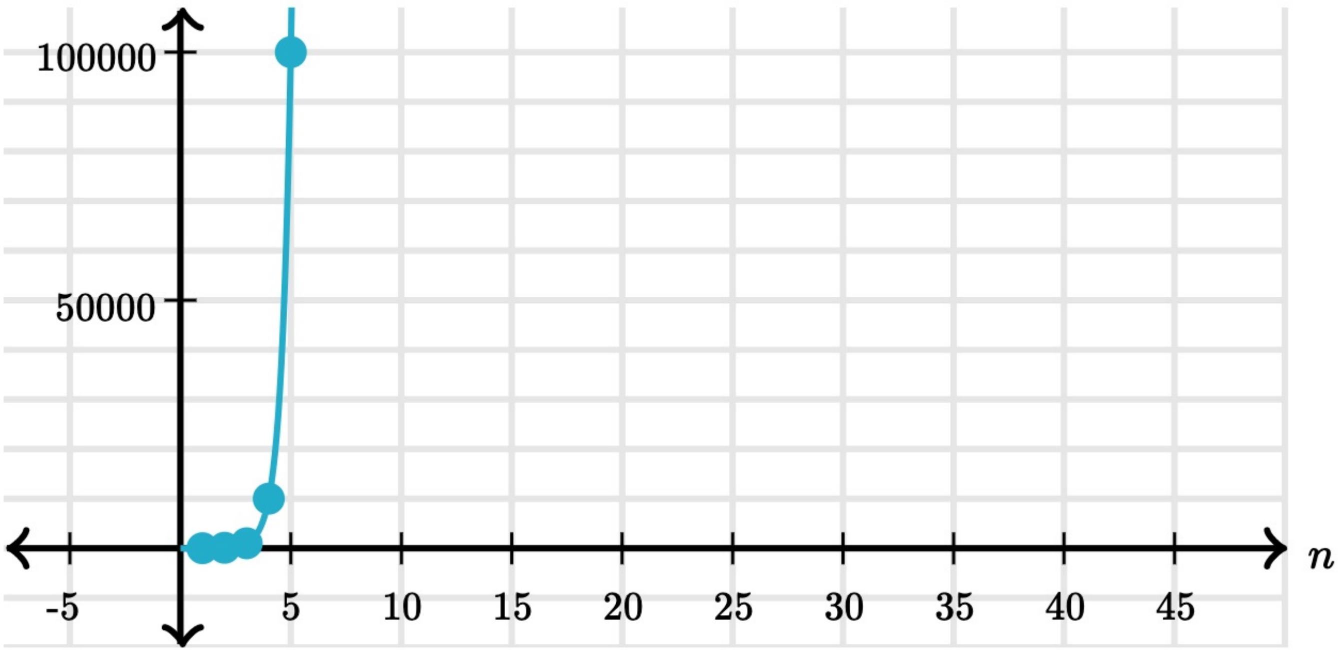
How many operations are required for increasing values of n?

N	Operations
1	1
2	3
3	5
4	9
7	41
8	67
20	21891

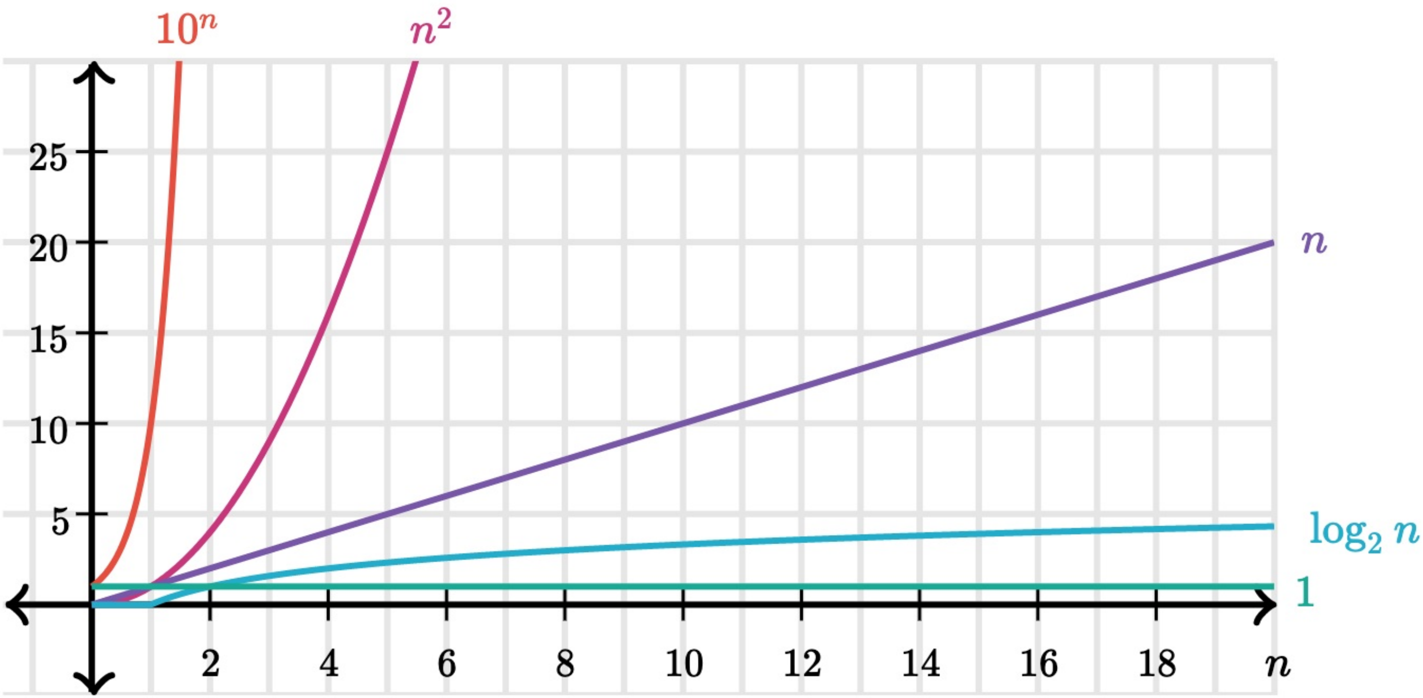
Exponential time

When an algorithm grows in **exponential time**, its number of steps increases faster than a polynomial function of the input size.

N	Operations
1	1
2	3
3	5
4	9
7	41
8	67
20	21891


100000
50000
-5
5
10
15
20
25
30
35
40
45
n

Comparing orders of growth



Big O/Big Theta Notation

A formal notation for describing the efficiency of an algorithm, using asymptotic analysis.

Order of growth	Example	Big Theta	Big O
Exponential growth	recursive `virfib`		
Quadratic growth	`overlap`		
Linear growth	`find_in_link`		
Logarithmic growth	`exp_fast`		
Constant growth	`add_to_front`		

Big O/Big Theta Notation

A formal notation for describing the efficiency of an algorithm, using asymptotic analysis.

Order of growth	Example	Big Theta	Big O
Exponential growth	recursive `virfib`	$\Theta(b^n)$	
Quadratic growth	`overlap`	$\Theta(n^2)$	
Linear growth	`find_in_link`	$\Theta(n)$	
Logarithmic growth	`exp_fast`	$\Theta(log_n)$	
Constant growth	`add_to_front`	$\Theta(1)$	

Big O/Big Theta Notation

A formal notation for describing the efficiency of an algorithm, using asymptotic analysis.

Order of growth	**Example**	**Big Theta**	**Big O**
Exponential growth	recursive `virfib`	$\Theta(b^n)$	$O(b^n)$
Quadratic growth	`overlap`	$\Theta(n^2)$	$O(n^2)$
Linear growth	`find_in_link`	$\Theta(n)$	$O(n)$
Logarithmic growth	`exp_fast`	$\Theta(log_n)$	$O(log_n)$
Constant growth	`add_to_front`	$\Theta(1)$	$O(1)$

Space

Space and environments

The space needed for a program depends on the environments in use.

At any moment there is a set of **active environments**.

Values and frames in active environments consume memory.

Memory that is used for other values and frames can be recycled.

Active environments:

- Environments for any function calls currently being evaluated.
- Parent environments of functions named in active environments.

Active environments in PythonTutor

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

View in PythonTutor

Make sure to select "don't display exited functions".

Visualization of space consumption

Memoization

Memoization

Memoization is a strategy to reduce redundant computation by remembering the results of previous function calls in a "memo".

A memoization HOF

```
def memo(f):
    cache = {}
    def memoized(n):
        if n not in cache:
            cache[n] = f(n)
        return cache[n]
    return memoized
```

Memoizing Virahanka-Fibonacci

n	**Original**	**Memoized**
5	15	9
6	25	11
7	41	13
8	67	15
9	109	17
10	177	19

Video visualization