Algorithms Design & Recursion

CS177 – Recitation 14

Agenda

- What’s an Algorithm.
- Search algorithms
  - Linear search
  - Binary search
- Recursion.
- Optional arguments in functions

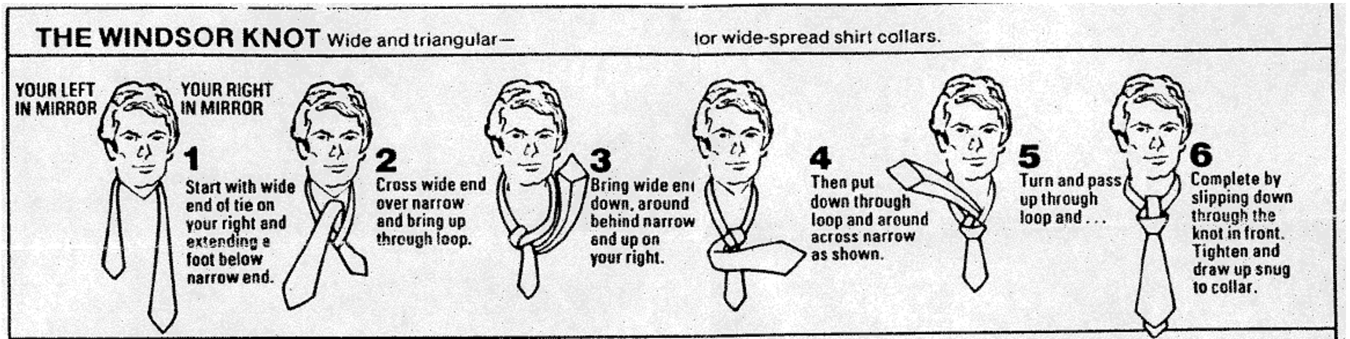
What’s an Algorithm

- An algorithm is a step-by-step list of instructions to solve a problem.
- An algorithm is like a recipe.

Best Brownies

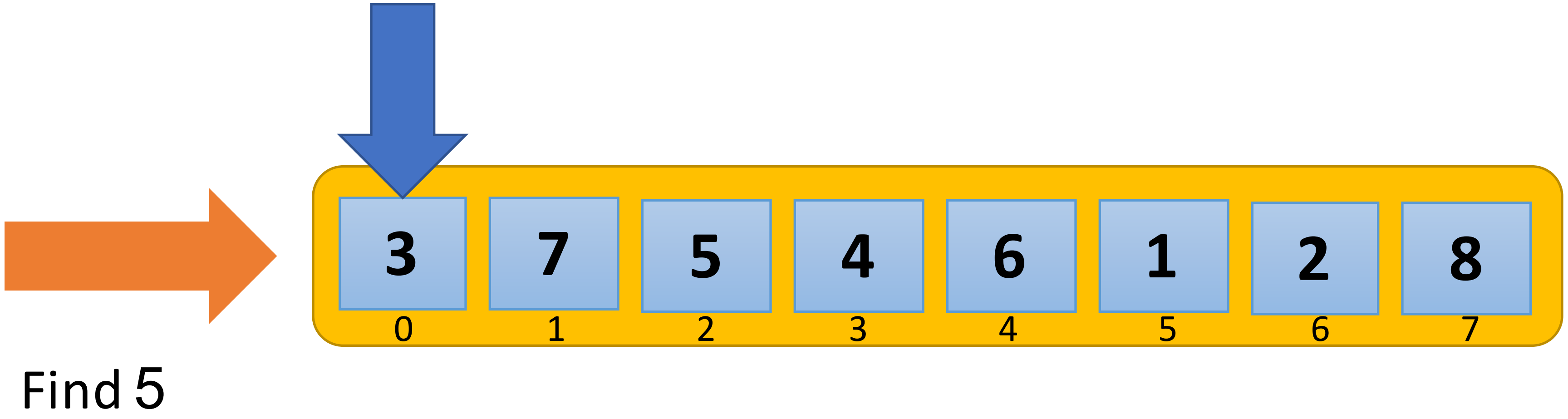
Directions

1. Preheat oven to 350 degrees F (175 degrees C). Grease and flour an 8-inch square pan.
2. In a large saucepan, melt 1/2 cup butter. Remove from heat, and stir in sugar, eggs, and 1 teaspoon vanilla. Beat in 1/3 cup cocoa, 1/2 cup flour, salt, and baking powder. Spread batter into prepared pan.
3. Bake in preheated oven for 25 to 30 minutes. Do not overcook.
4. To Make Frosting: Combine 3 tablespoons softened butter, 3 tablespoons cocoa, honey, 1 teaspoon vanilla extract, and 1 cup confectioners' sugar. Stir until smooth. Frost brownies while they are still warm.



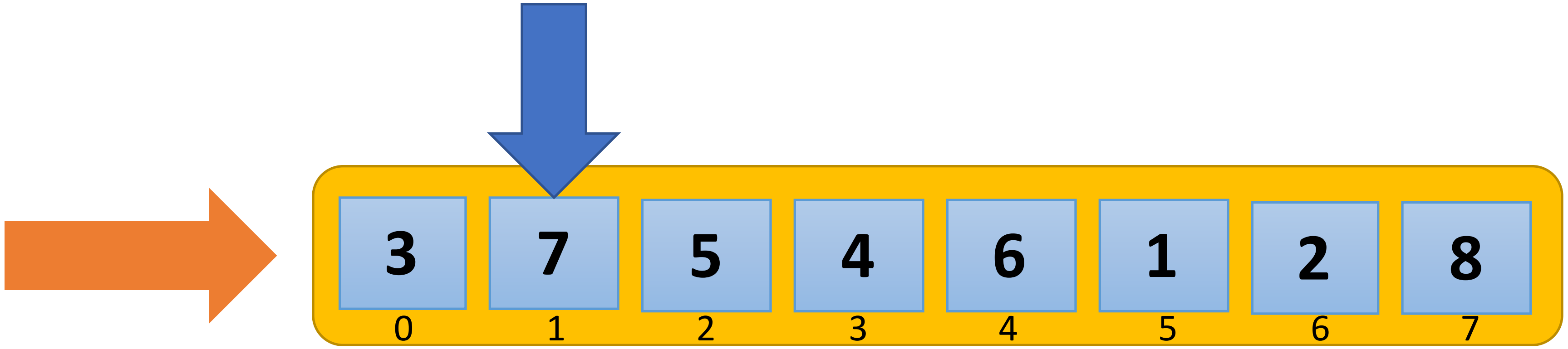
Search

- How would you find a number in a list of numbers?



Search

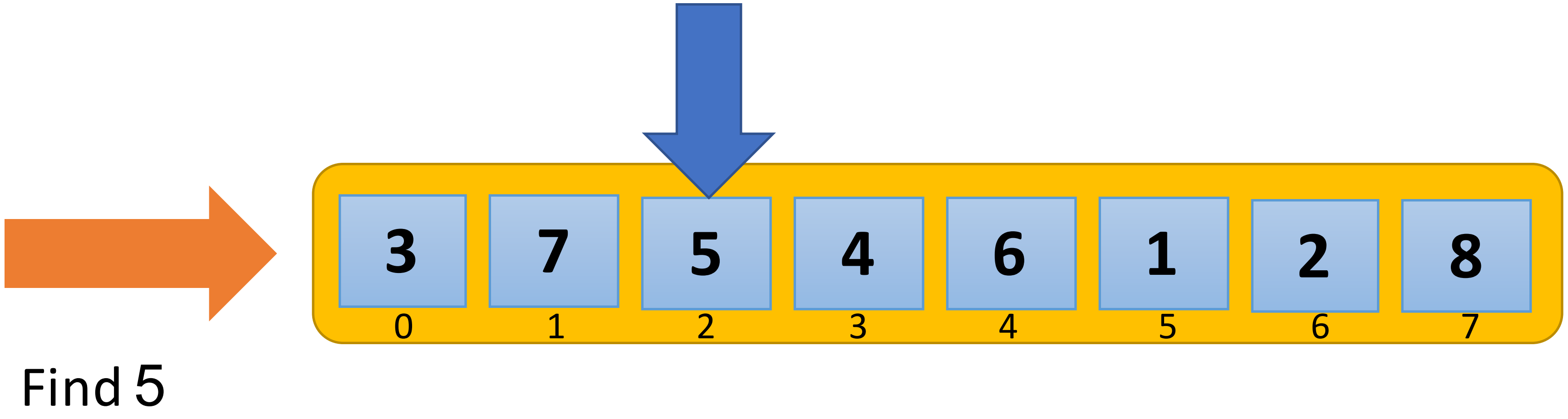
- How would you find a number in a list of numbers?



Find 5

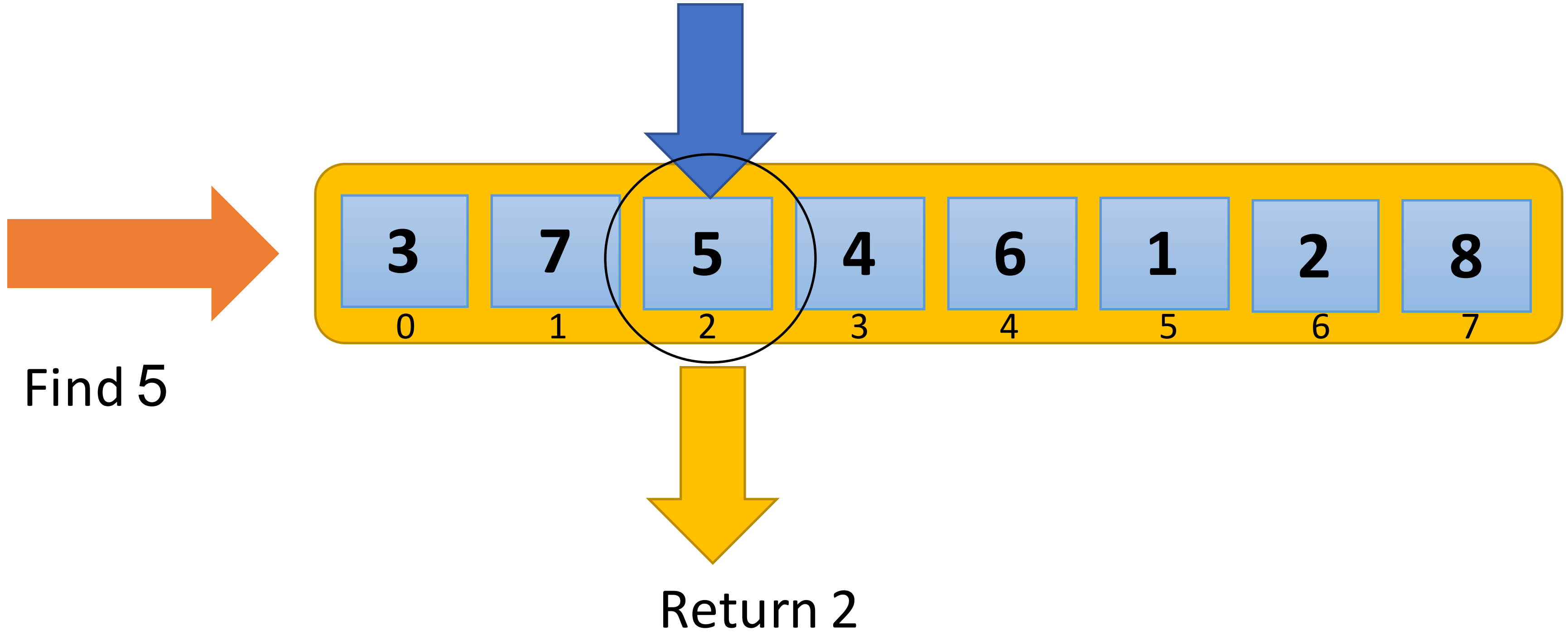
Search

- How would you find a number in a list of numbers?



Search

- How would you find a number in a list of numbers?



Search

- What we did is called “Sequential search” or “Linear search”.
- Keep going through the elements one by one till you find your match.
- How can we write this in Python?

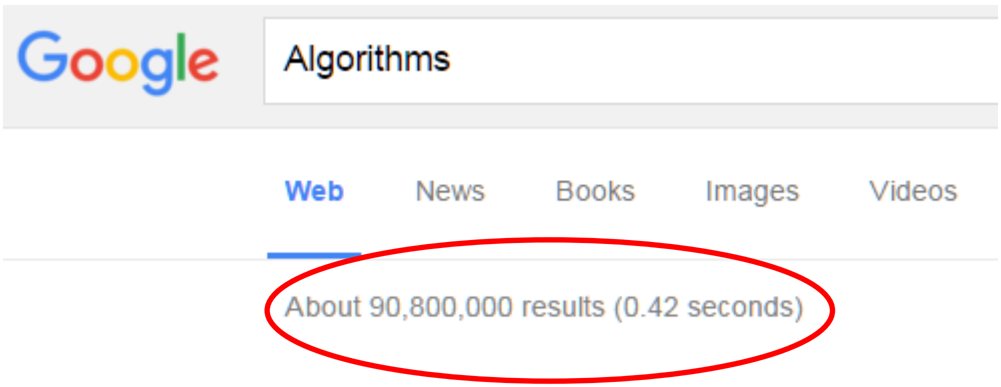
Sequential Search

```
def seqSsearch(nums, n):
  for i in range(len(nums)):
    if nums[i] == n:
      return i
  return -1
```

Is this the best way to do it !?

Search

- What happens if you are searching among very big number of elements ?



- There are also many algorithms solving the same problem.
- We want a good algorithm. But what defines “goodness”?

Evaluation of an Algorithm

- We evaluate an algorithm using two criteria's:
  - Space complexity: How much memory the algorithm needs? In other words, how many variables the algorithm needs?
  - Time complexity: The number of steps executed by the algorithms?
  - Why not just measure the time the algorithm takes !?
    - Different machines, architectures → different execution times !

- We need to express the space/time complexity in terms of the data size. For example: the size of the list we search in.

Space Complexity for Sequential Search

```python
def seqSsearch(nums, n):
  for i in range(len(nums)):
    if nums[i] == n:
      return i
  return -1
```

Uses only one variable: i

- If len(nums) equals 5, this algorithm will use only one variable (i).
- If len(nums) equals 5000, this algorithm will STILL use only one variable (i).
- This means the number of variables this algorithm uses is constant with respect the number of elements we process.
- The space complexity of this algorithm is *constant*.

Time Complexity for Sequential Search

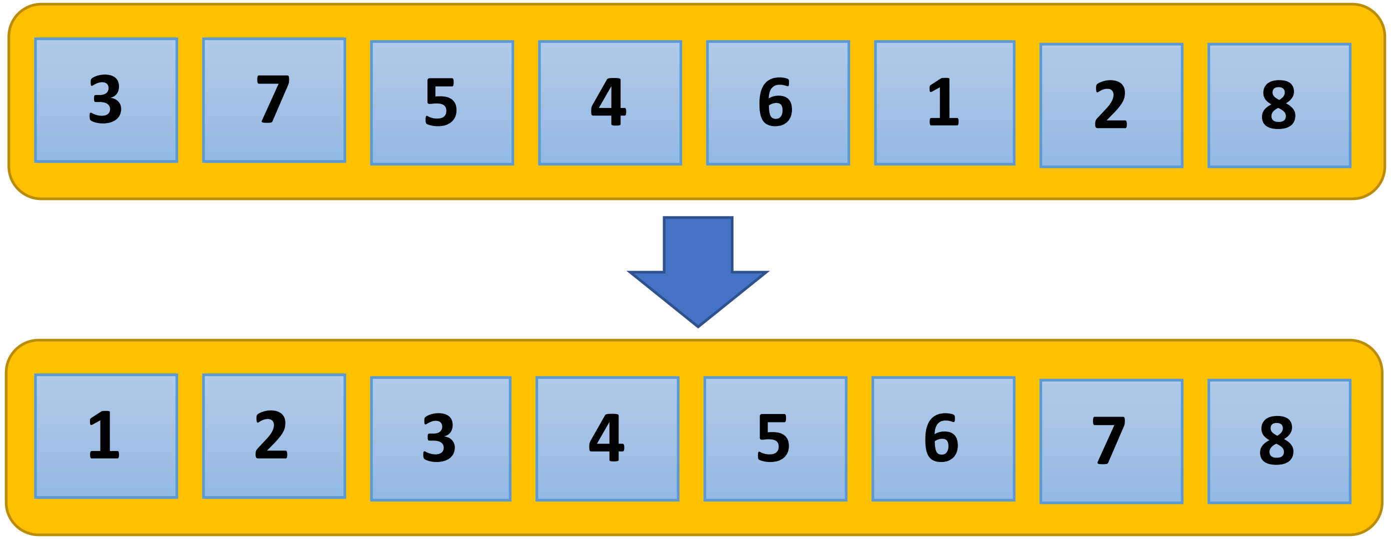
```
def seqSsearch(nums, n):
  for i in range(len(nums)):
    if nums[i] == n:
      return i
  return -1
```

Checking if two numbers are equal or not is the core operation of this algorithm.

- If len(nums) equals 5, this algorithm will check the if condition 5 times.
- If len(nums) equals 5000, this algorithm will the if condition 5000 times.
- This means the number of times the if condition is evaluated depends on the number of elements we process.
- The space complexity of this algorithm is *linear* with the size of the data.

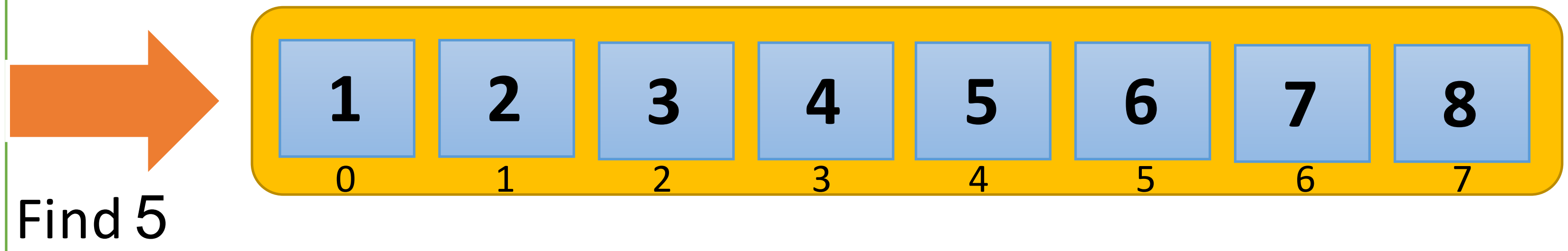
Binary Search

What if the list of numbers is sorted, how can we use that to enhance the algorithm?



Binary search

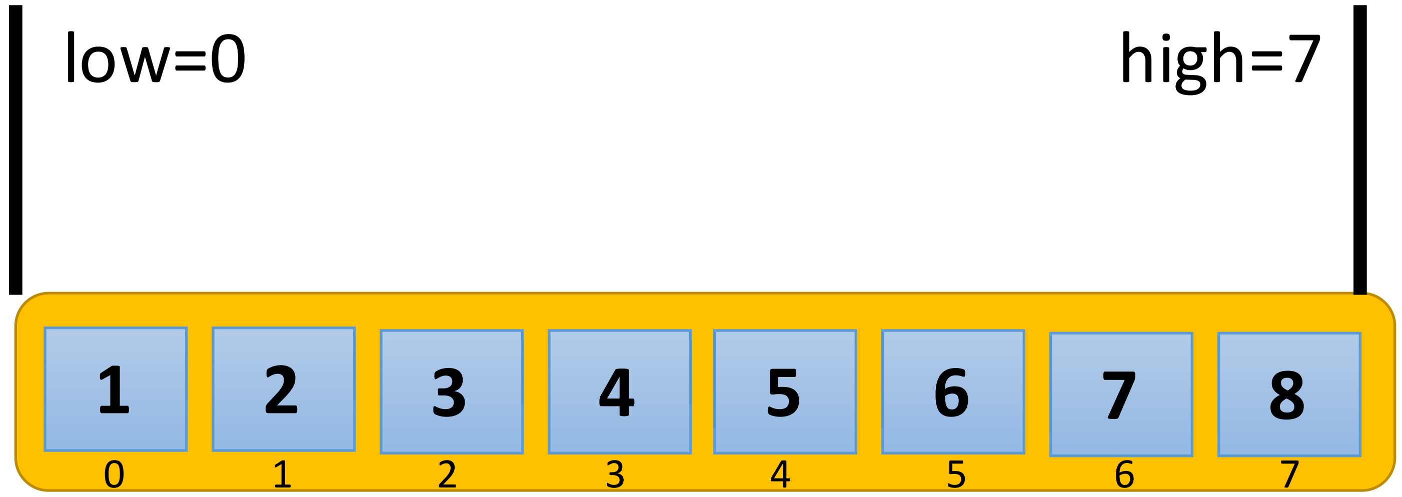
```
def bsearch(x, nums):
  low = 0
  high = len(nums) - 1
  while low <= high:
    mid = (low+high)//2
    item = nums[mid]
    if x = item:
      return mid
    elif x < item:
      high = mid - 1
    else:
      low = mid + 1
  return -1
```



Binary search

```python
def bsearch(x, nums):
  low = 0
  high = len(nums) - 1
  while low <= high:
    mid = (low+high)//2
    item = nums[mid]
    if x = item:
      return mid
    elif x < item:
      high = mid - 1
    else:
      low = mid + 1
  return -1
```

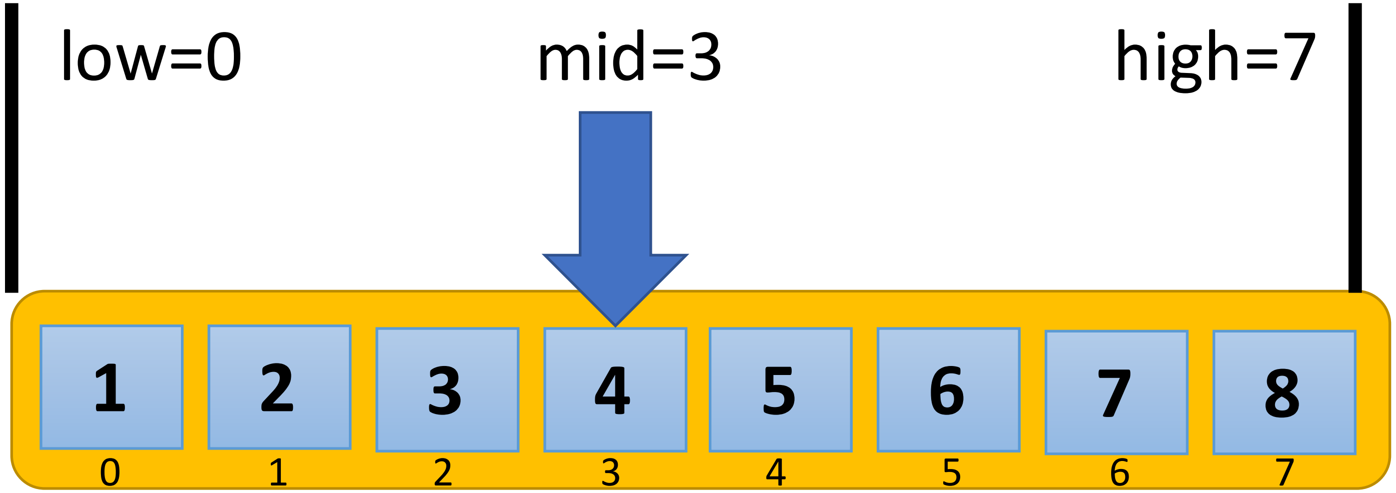
Find 5



Binary search

```python
def bsearch(x, nums):
  low = 0
  high = len(nums) - 1
  while low <= high:
    mid = (low+high)//2
    item = nums[mid]
    if x = item:
      return mid
    elif x < item:
      high = mid - 1
    else:
      low = mid + 1
  return -1
```

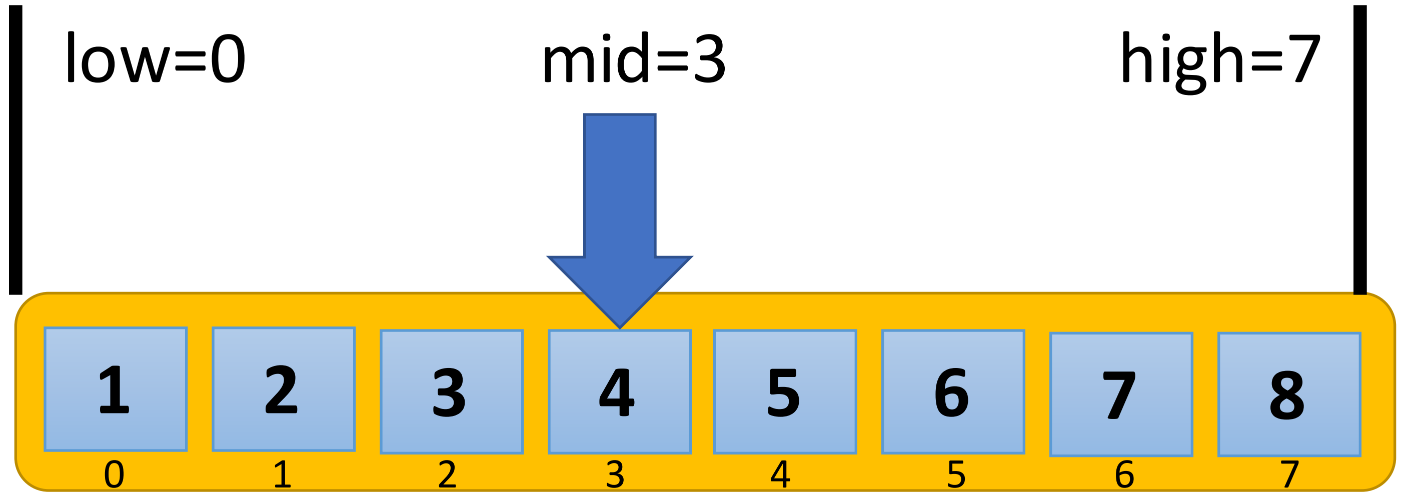
Find 5



Binary search

```
def bsearch(x, nums):
  low = 0
  high = len(nums) - 1
  while low <= high:
    mid = (low+high)//2
    item = nums[mid]
    if x = item:
      return mid
    elif x < item:
      high = mid - 1
    else:
      low = mid + 1
  return -1
```

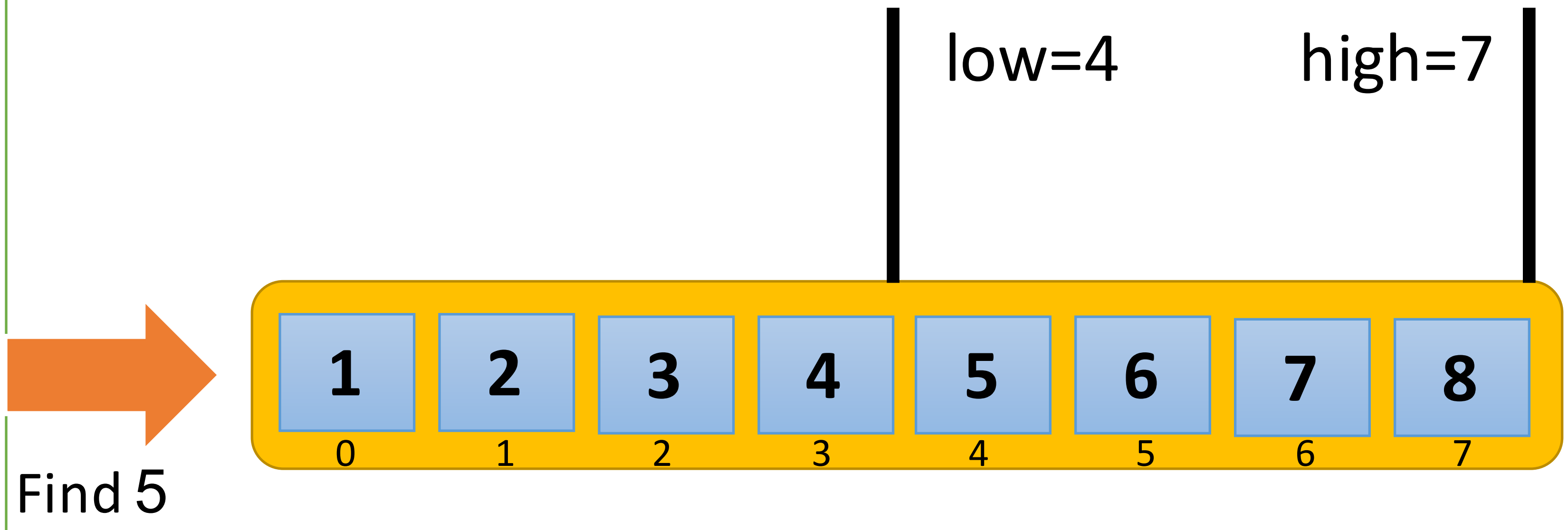
Find 5



item = nums[mid] = 4

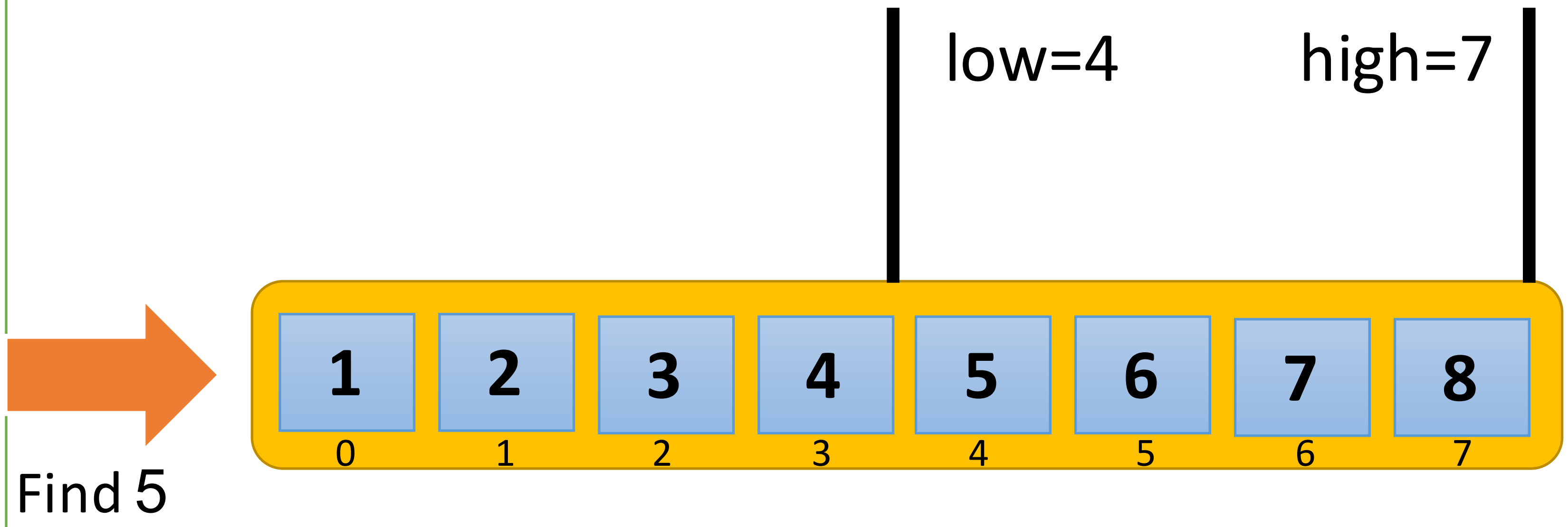
Binary search

```
def bsearch(x, nums):
  low = 0
  high = len(nums) - 1
  while low <= high:
    mid = (low+high)//2
    item = nums[mid]
    if x = item:
      return mid
    elif x < item:
      high = mid - 1
    else:
      low = mid + 1
  return -1
```



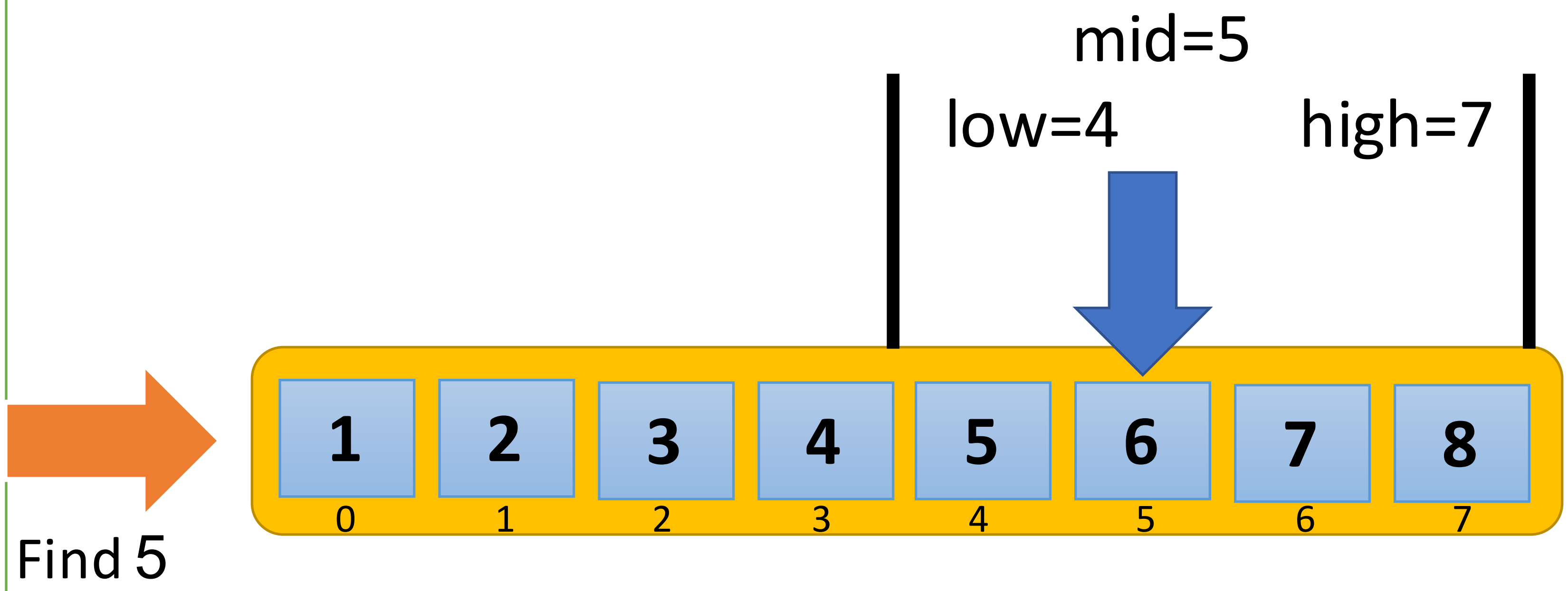
Binary search

```
def bsearch(x, nums):
  low = 0
  high = len(nums) - 1
  while low <= high:
    mid = (low+high)//2
    item = nums[mid]
    if x = item:
      return mid
    elif x < item:
      high = mid - 1
    else:
      low = mid + 1
  return -1
```



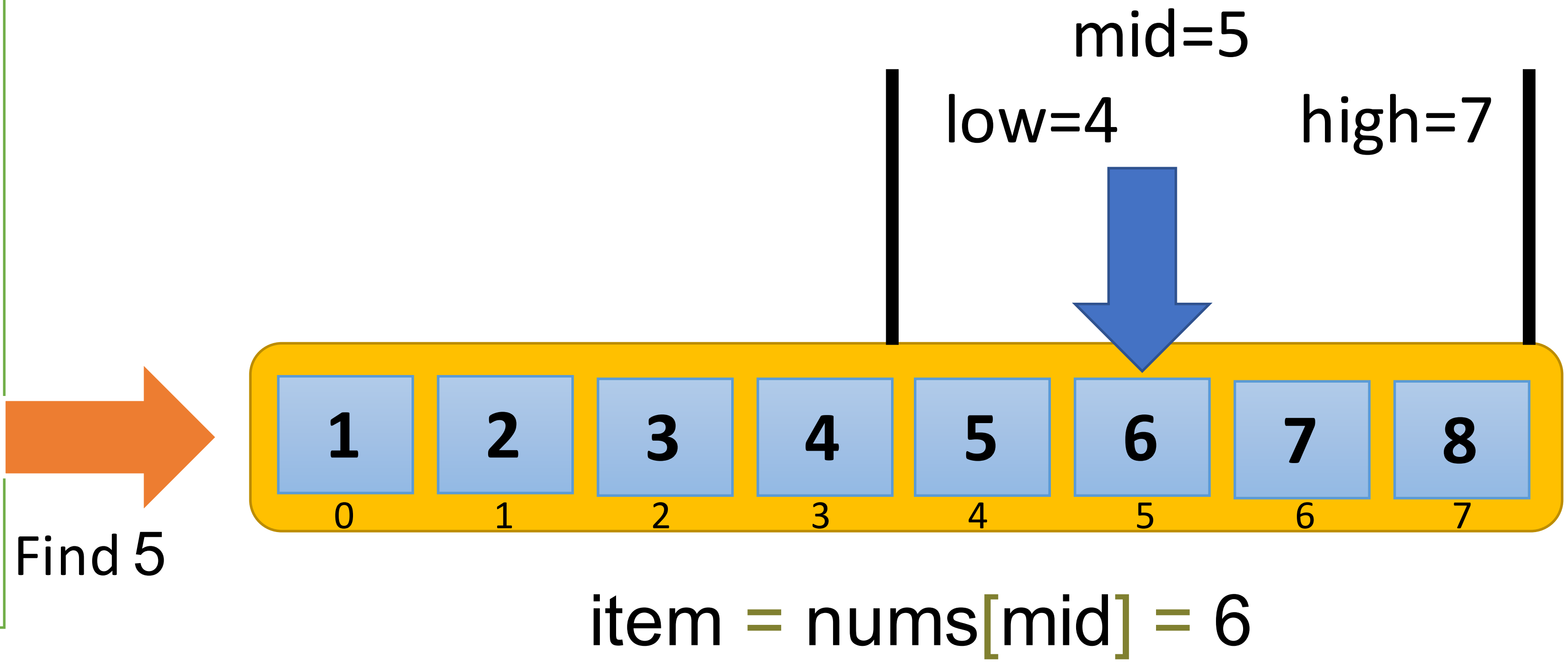
Binary search

```
def bsearch(x, nums):
  low = 0
  high = len(nums) - 1
  while low <= high:
    mid = (low+high)//2
    item = nums[mid]
    if x = item:
      return mid
    elif x < item:
      high = mid - 1
    else:
      low = mid + 1
  return -1
```



Binary search

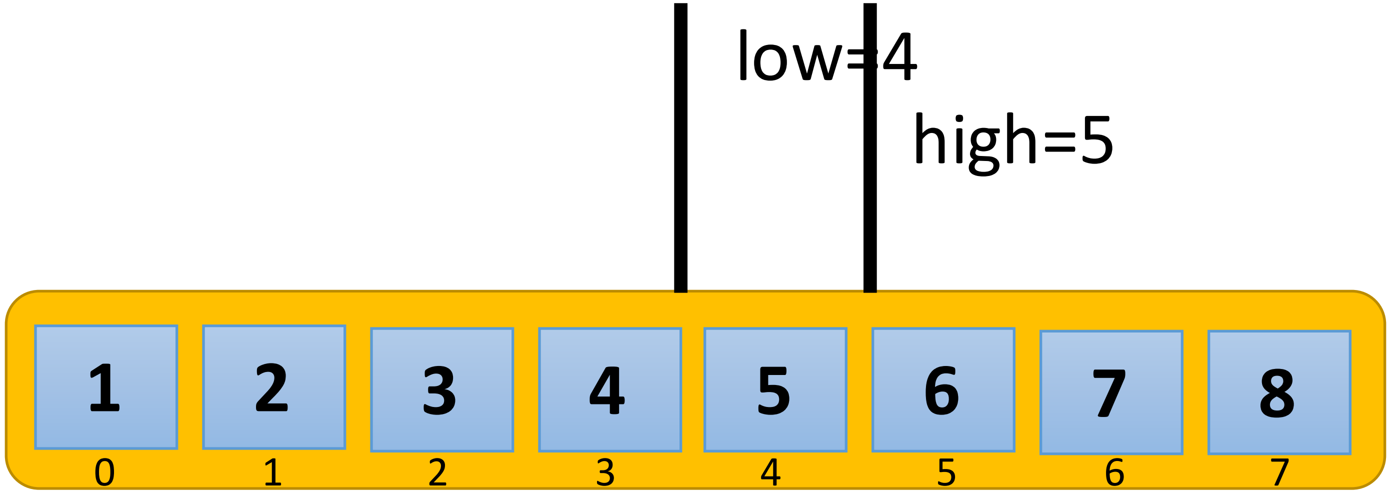
```
def bsearch(x, nums):
  low = 0
  high = len(nums) - 1
  while low <= high:
    mid = (low+high)//2
    item = nums[mid]
    if x = item:
      return mid
    elif x < item:
      high = mid - 1
    else:
      low = mid + 1
  return -1
```



Binary search

```
def bsearch(x, nums):
  low = 0
  high = len(nums) - 1
  while low <= high:
    mid = (low+high)//2
    item = nums[mid]
    if x = item:
      return mid
    elif x < item:
      high = mid - 1
    else:
      low = mid + 1
  return -1
```

Find 5

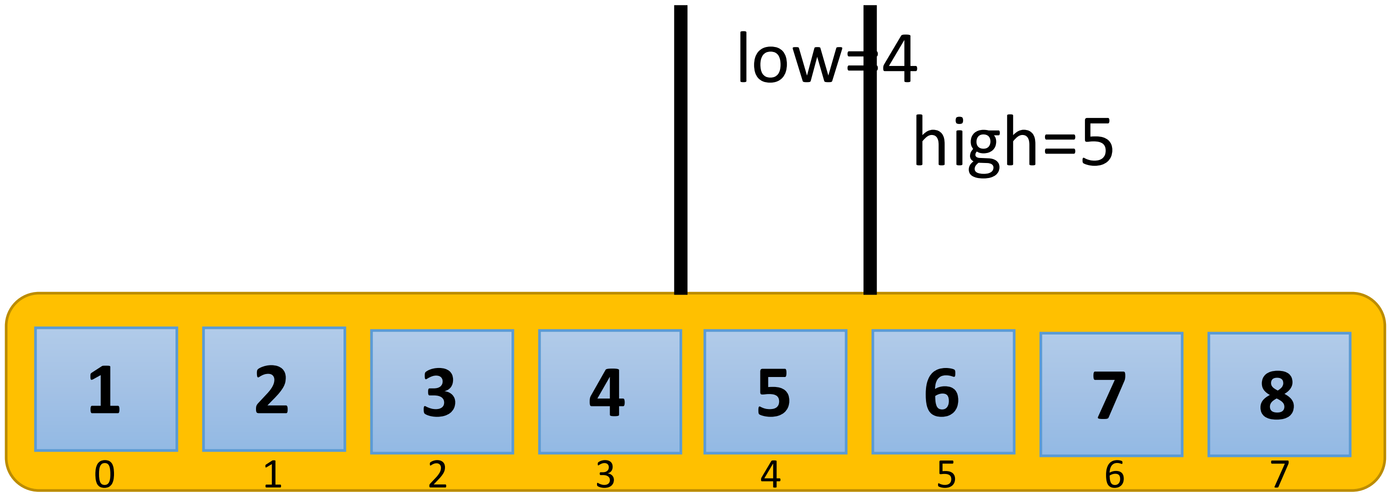


item = nums[mid] = 6

Binary search

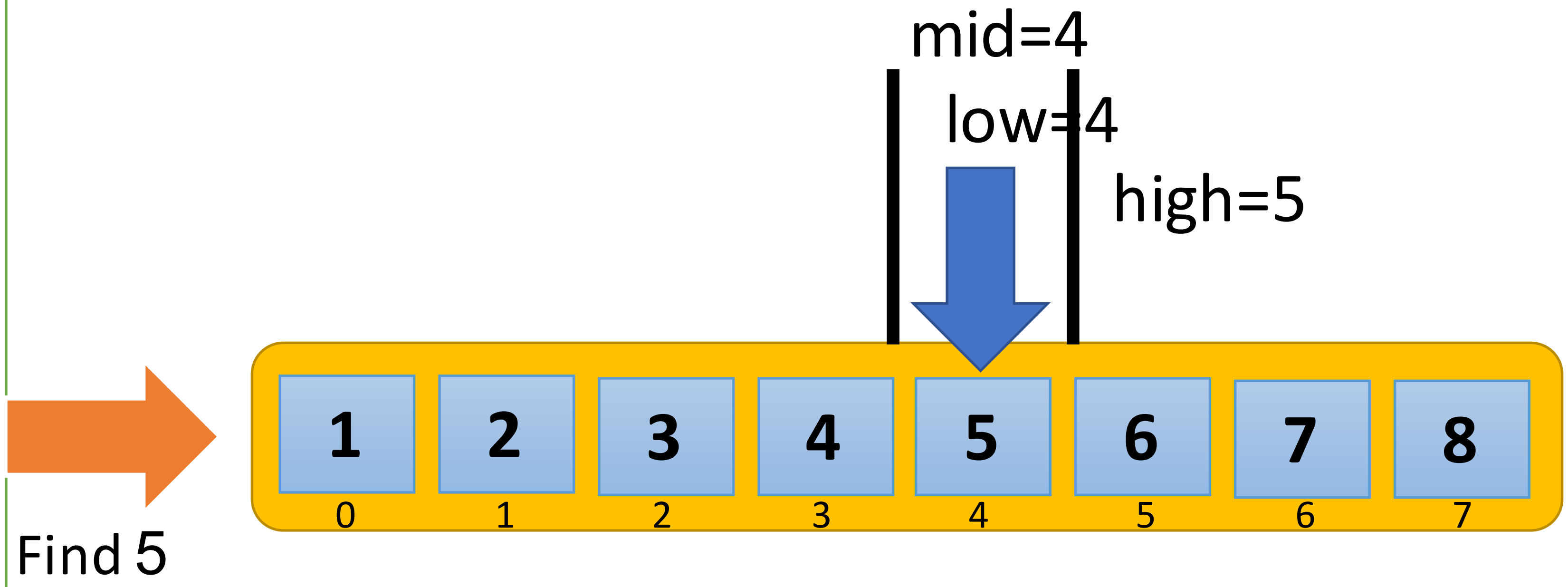
```python
def bsearch(x, nums):
  low = 0
  high = len(nums) - 1
  while low <= high:
    mid = (low+high)//2
    item = nums[mid]
    if x = item:
      return mid
    elif x < item:
      high = mid - 1
    else:
      low = mid + 1
  return -1
```

Find 5



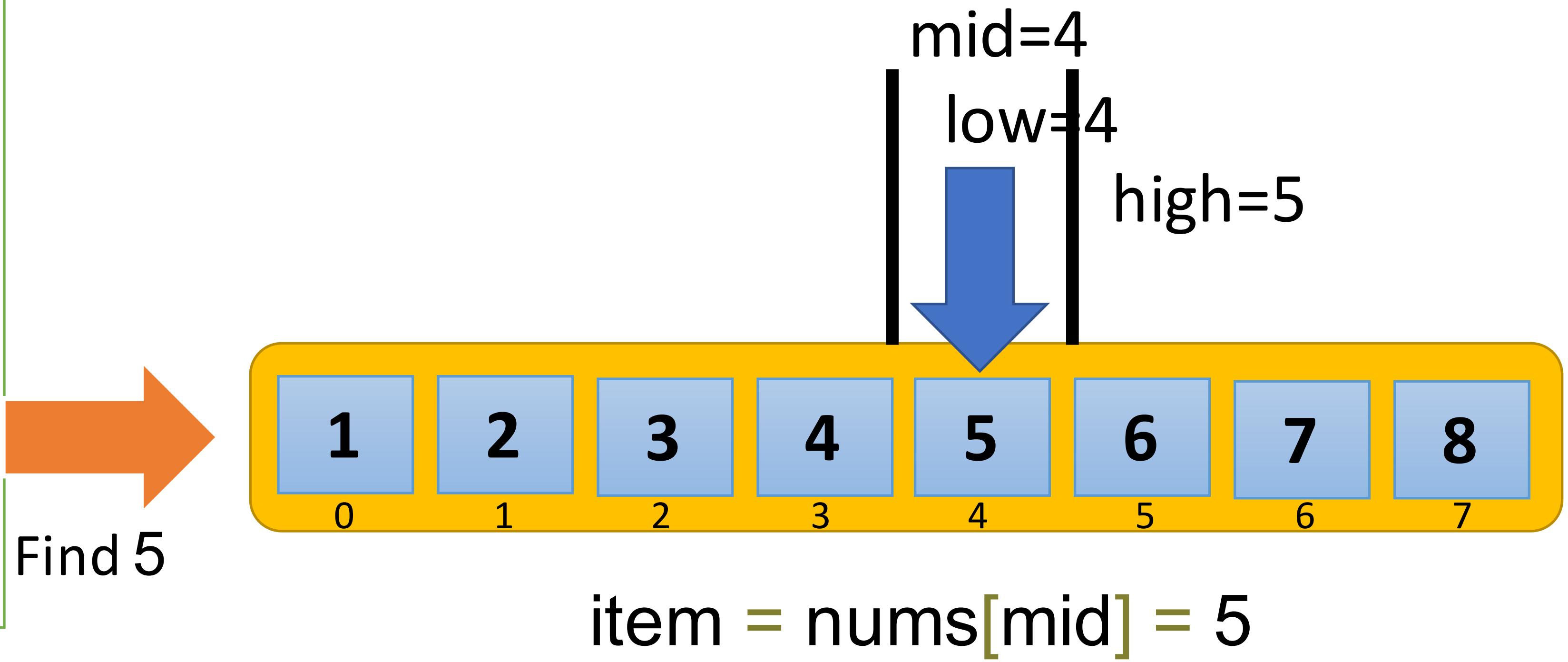
Binary search

```
def bsearch(x, nums):
  low = 0
  high = len(nums) - 1
  while low <= high:
    mid = (low+high)//2
    item = nums[mid]
    if x = item:
      return mid
    elif x < item:
      high = mid - 1
    else:
      low = mid + 1
  return -1
```



Binary search

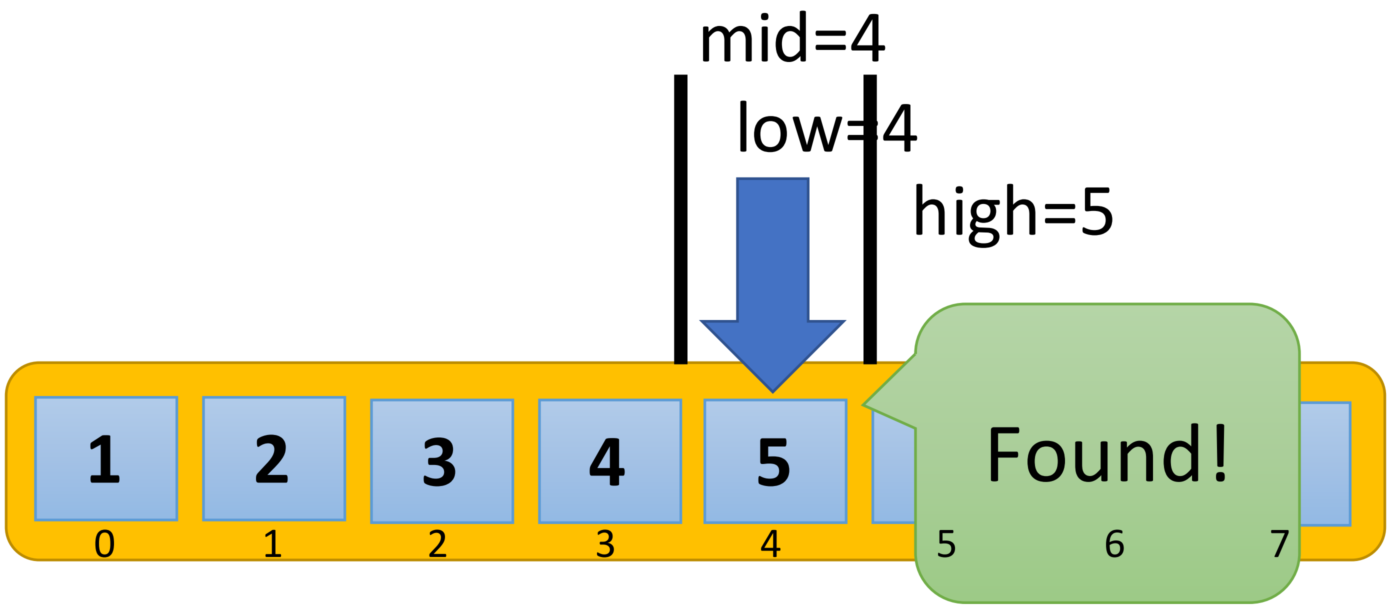
```
def bsearch(x, nums):
  low = 0
  high = len(nums) - 1
  while low <= high:
    mid = (low+high)//2
    item = nums[mid]
    if x = item:
      return mid
    elif x < item:
      high = mid - 1
    else:
      low = mid + 1
  return -1
```



Binary search

```
def bsearch(x, nums):
  low = 0
  high = len(nums) - 1
  while low <= high:
    mid = (low+high)//2
    item = nums[mid]
    if x = item:
      return mid
    elif x < item:
      high = mid - 1
    else:
      low = mid + 1
  return -1
```

Find 5



item = nums[mid] = 5

Binary search: Analysis

```
def bsearch(x, nums):
  low = 0
  high = len(nums) - 1
  while low <= high:
    mid = (low+high)//2
    item = nums[mid]
    if x = item:
      return mid
    elif x < item:
      high = mid - 1
    else:
      low = mid + 1
  return -1
```

- In each iteration, search space is reduced by half.
  - Initially, search in 8 numbers (1~8)
  - Then, search in 4 numbers (5~8)
  - Finally, search in one number (5)
  - The number of iterations is $\log_2$(len(nums))=3
  - **Logarithmic time complexity**
- Use four variables: low, high, mid, item
  - Independent of len(nums)
  - **Constant space complexity**

Ok…. So what?

- Have you heard about the buzzword "**BigData**"?
- What if you are asked to search in a list of a billion numbers?

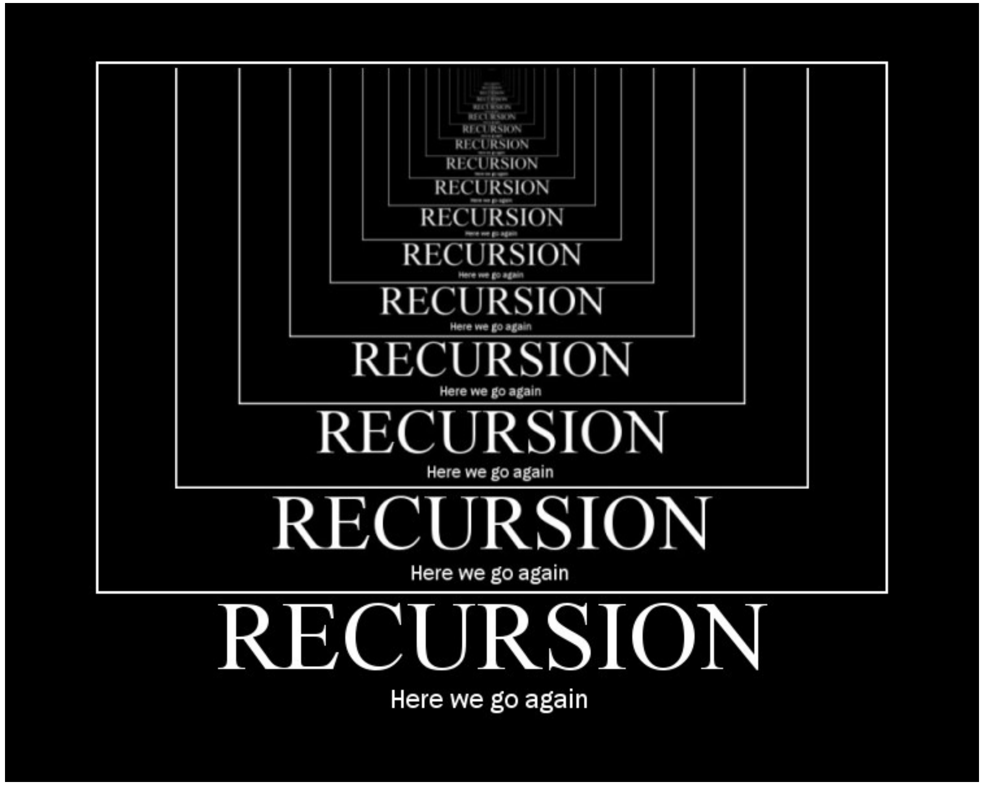
Algorithm	Time complexity
Linear search	Run billions of steps
Binary search	several dozen steps

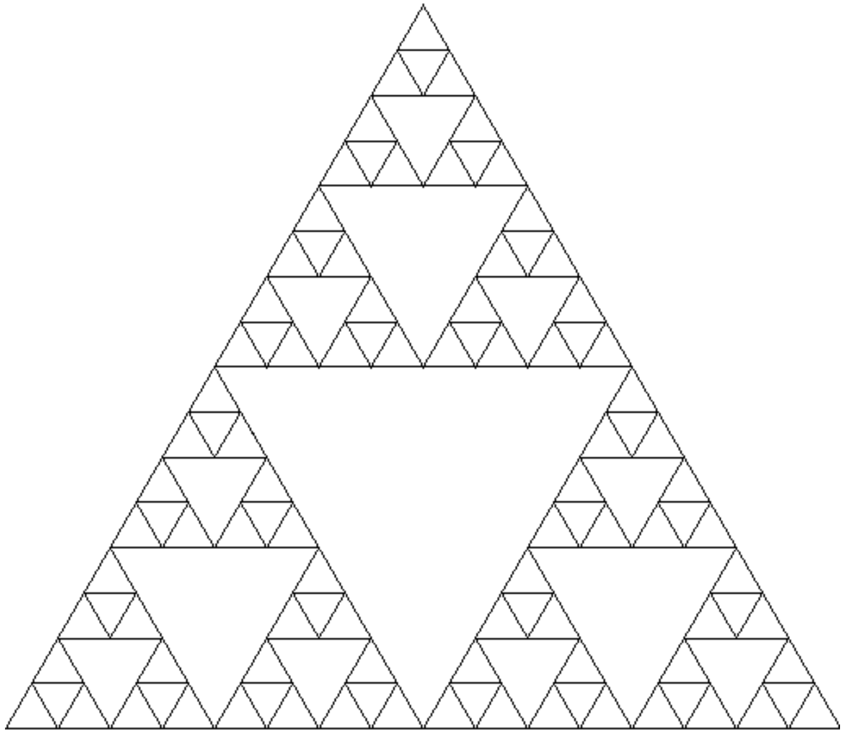
Win!

Recursion

Recursion

- Recursion is the process of repeating items in a self-similar way.





Recursion

- You have the function “Dream” ☺
- Each time the function dream calls it self (recursive call), you get into a deeper dream level.
- To wake up from the first dream, you need to wake up from all dreams !
- To wake up you need a kick ! The kick in recursion is the return statement.

Calculating Factorial

- Given that Factorial (1)=Factorial (0)=1
- Factorial (5) = 5 * 4 * 3 * 2 * 1 = 120
- We can write factorial (5) in term of the factorial of smaller numbers:
- Factorial (5) = 5 * Factorial (4)

  = 5 * 4 * Factorial (3)

  = 5 * 4 * 3 * Factorial (2)

  = 5 * 4 * 3 * 2 * Factorial (1)

  = 5 * 4 * 3 * 2 * 1 = 120

- Generally: Factorial (x) = x * Factorial (x-1)

Calculating Factorial

```python
def factorial(x):
    if(x<2):
        return 1
    return x * factorial(x-1)

def main():
    print(factorial(5))
```

Result = 5 * factorial (4)

24

4 * factorial (3)

6

3 * factorial (2)

120

2

2 * factorial (1)

1

* 1

Optional arguments in functions

If b is given, use given b

If b is not given, use b = 10

```
def fun( a, b = 10 ):
    print(a)
    print(b)

fun(100)
fun(100, 200)
fun(100, b = 200)
```

Output:

```
100
10
100
200
100
200
```

```
def fun( a = 3 ):
    print(a)
    if a > 0:
        fun( a - 1)

fun()
fun( 5 )
```

Output:

3

2

1

0

5

4

3

2

1

0