Binary Search

Efficient Searching Through Halves

Recap: Linear Search Complexity

- Best Case
  - target = **3**, List = [**3**, 6, 4, 9, 2]
  - O(1) → Constant time

Recap: Linear Search Complexity

- Best Case
  - target = 3, List = [3, 6, 4, 9, 2]
  - O(1) → Constant time
- Worst Case
  - target = **2 or ?**, List = [3, 6, 4, 9, **2**]

Recap: Linear Search Complexity

- Best Case
  - target = 3, List = [3, 6, 4, 9, 2]
  - O(1) → Constant time
- Worst Case
  - target = **2 or ?**, List = [3, 6, 4, 9, **2**]
  - O(n) → Linear time

Recap: Linear Search Complexity

- Best Case
  - target = 3, List = [3, 6, 4, 9, 2]
  - O(1) → Constant time
- Worst Case
  - target = 2 or ?, List = [3, 6, 4, 9, 2]
  - O(n) → Linear time
- Average Case
  - target = **4**, List = [3, 6, **4**, 9, 2]

Recap: Linear Search Complexity

- Best Case
  - target = 3, List = [3, 6, 4, 9, 2]
  - O(1) → Constant time
- Worst Case
  - target = 2 or ?, List = [3, 6, 4, 9, 2]
  - O(n) → Linear time
- Average Case
  - target = **4**, List = [3, 6, **4**, 9, 2]
  - n/2 ≈ O(n) → Linear time

Recap: Linear Search Complexity

- Best Case
  - target = 3, List = [3, 6, 4, 9, 2]
  - O(1) → Constant time
- **Worst Case**
  - target = 2 or ?, List = [3, 6, 4, 9, 2]
  - O(n) → Linear time
- **Average Case**
  - target = 4, List = [3, 6, 4, 9, 2]
  - n/2 ≈ O(n) → Linear time

From Linear to Binary Search: A Faster Way

- **Recap Linear Search:**
  - Checks each element in a list one by one

From Linear to Binary Search: A Faster Way

- **Recap Linear Search:**
  - Checks each element in a list one by one
  - Inefficient as the list grows larger

From Linear to Binary Search: A Faster Way

- **Recap Linear Search:**
  - Checks each element in a list one by one
  - Inefficient as the list grows larger
  - Linear time complexity: **O(n)**

From Linear to Binary Search: A Faster Way

- **Recap Linear Search:**
  - Checks each element in a list one by one
  - Inefficient as the list grows larger
  - Linear time complexity: **O(n)**
- **Need for Speed:**
  - Imagine searching through 1 million records

From Linear to Binary Search: A Faster Way

- **Recap Linear Search:**
  - Checks each element in a list one by one
  - Inefficient as the list grows larger
  - Linear time complexity: **O(n)**
- **Need for Speed:**
  - Imagine searching through 1 million records
  - 1 million steps in the worst case

From Linear to Binary Search: A Faster Way

- **Recap Linear Search:**
  - Checks each element in a list one by one
  - Inefficient as the list grows larger
  - Linear time complexity: **O(n)**
- **Need for Speed:**
  - Imagine searching through 1 million records
  - 1 million steps in the worst case
  - Is there a faster way?

From Linear to Binary Search: A Faster Way

- **Recap Linear Search:**
  - Checks each element in a list one by one
  - Inefficient as the list grows larger
  - Linear time complexity: **O(n)**
- **Need for Speed:**
  - Imagine searching through 1 million records
  - 1 million steps in the worst case
  - Is there a faster way?
  - Yes! **Binary Search**

From Linear to Binary Search: A Faster Way

- **Recap Linear Search:**
  - Checks each element in a list one by one
  - Inefficient as the list grows larger
  - Linear time complexity: **O(n)**
- **Need for Speed:**
  - Imagine searching through 1 million records
  - 1 million steps in the worst case
  - Is there a faster way?
  - Yes! **Binary Search**
- **What is Binary Search?:**
  - Binary Search halves the list at each step, reducing the total steps

From Linear to Binary Search: A Faster Way

- **Recap Linear Search:**
  - Checks each element in a list one by one
  - Inefficient as the list grows larger
  - Linear time complexity: **O(n)**
- **Need for Speed:**
  - Imagine searching through 1 million records
  - 1 million steps in the worst case
  - Is there a faster way?
  - Yes! **Binary Search**
- **What is Binary Search?:**
  - Binary Search halves the list at each step, reducing the total steps
  - But it only works on **sorted lists**

From Linear to Binary Search: A Faster Way

- **Recap Linear Search:**
  - Checks each element in a list one by one
  - Inefficient as the list grows larger
  - Linear time complexity: **O(n)**
- **Need for Speed:**
  - Imagine searching through 1 million records
  - 1 million steps in the worst case
  - Is there a faster way?
  - Yes! **Binary Search**
- **What is Binary Search?:**
  - Binary Search halves the list at each step, reducing the total steps
  - But it only works on **sorted lists**
- **Efficiency:**
  - Binary Search finishes in log(n) steps instead of n

From Linear to Binary Search: A Faster Way

- **Recap Linear Search:**
  - Checks each element in a list one by one
  - Inefficient as the list grows larger
  - Linear time complexity: **O(n)**
- **Need for Speed:**
  - Imagine searching through 1 million records
  - 1 million steps in the worst case
  - Is there a faster way?
  - Yes! **Binary Search**

- **What is Binary Search?:**
  - Binary Search halves the list at each step, reducing the total steps
  - But it only works on **sorted lists**
- **Efficiency:**
  - Binary Search finishes in log(n) steps instead of n
  - For 1 million items, Binary Search takes around **20 steps at most**

From Linear to Binary Search: A Faster Way

- **Recap Linear Search:**
  - Checks each element in a list one by one
  - Inefficient as the list grows larger
  - Linear time complexity: **O(n)**
- **Need for Speed:**
  - Imagine searching through 1 million records
  - 1 million steps in the worst case
  - Is there a faster way?
  - Yes! **Binary Search**
- **What is Binary Search?:**
  - Binary Search halves the list at each step, reducing the total steps
  - But it only works on **sorted lists**
- **Efficiency:**
  - Binary Search finishes in log(n) steps instead of n
  - For 1 million items, Binary Search takes around **20 steps at most**
- **Setting up the Challenge:**
  - Let’s explore this idea with a number guessing game

From Linear to Binary Search: A Faster Way

- **Recap Linear Search:**
  - Checks each element in a list one by one
  - Inefficient as the list grows larger
  - Linear time complexity: **O(n)**
- **Need for Speed:**
  - Imagine searching through 1 million records
  - 1 million steps in the worst case
  - Is there a faster way?
  - Yes! **Binary Search**
- **What is Binary Search?:**
  - Binary Search halves the list at each step, reducing the total steps
  - But it only works on **sorted lists**
- **Efficiency:**
  - Binary Search finishes in log(n) steps instead of n
  - For 1 million items, Binary Search takes around **20 steps at most**
- **Setting up the Challenge:**
  - Let's explore this idea with a number guessing game
  - Imagine thinking of a number from a list of 16 sorted numbers

From Linear to Binary Search: A Faster Way

- **Recap Linear Search:**
  - Checks each element in a list one by one
  - Inefficient as the list grows larger
  - Linear time complexity: **O(n)**
- **Need for Speed:**
  - Imagine searching through 1 million records
  - 1 million steps in the worst case
  - Is there a faster way?
  - Yes! **Binary Search**
- **What is Binary Search?:**
  - Binary Search halves the list at each step, reducing the total steps
  - But it only works on **sorted lists**
- **Efficiency:**
  - Binary Search finishes in log(n) steps instead of n
  - For 1 million items, Binary Search takes around **20 steps at most**
- **Setting up the Challenge:**
  - Let's explore this idea with a number guessing game
  - Imagine thinking of a number from a list of 16 sorted numbers
  - Can you guess it in no more than **4 attempts?**

Binary Search: Finding a Number in Sorted List

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
List	3	7	12	18	23	27	30	35	40	45	50	54	60	65	70	72

Binary Search: Finding a Number in Sorted List

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
List	3	7	12	18	23	27	30	35	40	45	50	54	60	65	70	72

Binary Search: Finding a Number in Sorted List

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
List	3	7	12	18	23	27	30	35	40	45	50	54	60	65	70	72

Binary Search: Finding a Number in Sorted List

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
List	3	7	12	18	23	27	30	35	40	45	50	54	60	65	70	72

We are thinking of
12

Binary Search: Finding a Number in Sorted List

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
List	3	7	12	18	23	27	30	35	40	45	50	54	60	65	70	72

We are thinking of
12

Calculate middle index
mid = (first + last) // 2

1. mid = (0 + 15) // 2 = **7**

Binary Search: Finding a Number in Sorted List

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
List	3	7	12	18	23	27	30	35	40	45	50	54	60	65	70	72

We are thinking of

12

Calculate middle index

mid = (first + last) // 2

1. mid = (0 + 15) // 2 = **7**

Binary Search: Finding a Number in Sorted List

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
List	3	7	12	18	23	27	30	35	40	45	50	54	60	65	70	72

Is **12 == 35?** Yes, Greater, Less? **Less**

We are thinking of
12

Calculate middle index
mid = (first + last) // 2

1. mid = (0 + 15) // 2 = **7**

Binary Search: Finding a Number in Sorted List

Index	0	1	2	3	4	5	6
List	3	7	12	18	23	27	30

We are thinking of
12

Calculate middle index
mid = (first + last) // 2

1. mid = (0 + 15) // 2 = **7**

Is **12 == 35?** Yes, Greater, Less? **Less**

Binary Search: Finding a Number in Sorted List

Index	0	1	2	3	4	5	6
List	3	7	12	18	23	27	30

Is **12 == 35?** Yes, Greater, Less? **Less**

We are thinking of
12

Calculate middle index
mid = (first + last) // 2

1. mid = (0 + 15) // 2 = **7**

2. mid = (0 + 6) // 2 = **3**

Binary Search: Finding a Number in Sorted List

Index	0	1	2	3	4	5	6
List	3	7	12	18	23	27	30

Is **12 == 35?** Yes, Greater, Less? **Less**

We are thinking of
12

Calculate middle index
mid = (first + last) // 2

1. mid = (0 + 15) // 2 = **7**

2. mid = (0 + 6) // 2 = **3**

Binary Search: Finding a Number in Sorted List

Index	0	1	2	3	4	5	6
List	3	7	12	18	23	27	30

Is **12 == 18?** Yes, Greater, Less? **Less**

We are thinking of
12

Calculate middle index
mid = (first + last) // 2

1. mid = (0 + 15) // 2 = **7**

2. mid = (0 + 6) // 2 = **3**

Binary Search: Finding a Number in Sorted List

Index	0	1	2
List	3	7	12

Is **12 == 18?** Yes, Greater, Less? **Less**

We are thinking of
12

Calculate middle index
mid = (first + last) // 2

1. mid = (0 + 15) // 2 = **7**

2. mid = (0 + 6) // 2 = **3**

Binary Search: Finding a Number in Sorted List

Index	0	1	2
List	3	7	12

Is **12 == 18?** Yes, Greater, Less? **Less**

We are thinking of
12

Calculate middle index
mid = (first + last) // 2

1. mid = (0 + 15) // 2 = **7**

2. mid = (0 + 6) // 2 = **3**

3. mid = (0 + 2) // 2 = **1**

Binary Search: Finding a Number in Sorted List

Index	0	1	2
List	3	7	12

Is **12 == 18?** Yes, Greater, Less? **Less**

We are thinking of
12

Calculate middle index
mid = (first + last) // 2

1. mid = (0 + 15) // 2 = **7**

2. mid = (0 + 6) // 2 = **3**

3. mid = (0 + 2) // 2 = **1**

Binary Search: Finding a Number in Sorted List

Index	0	1	2
List	3	7	12

Is **12 == 7?** Yes, Greater, Less? **Greater**

We are thinking of
12

Calculate middle index
mid = (first + last) // 2

1. **mid** = (0 + 15) // 2 = **7**
2. **mid** = (0 + 6) // 2 = **3**
3. **mid** = (0 + 2) // 2 = **1**

Binary Search: Finding a Number in Sorted List

Index		2	
List		12	

Is **12 == 7?** Yes, Greater, Less? **Greater**

We are thinking of
12

Calculate middle index
mid = (first + last) // 2

1. mid = (0 + 15) // 2 = **7**

2. mid = (0 + 6) // 2 = **3**

3. mid = (0 + 2) // 2 = **1**

Binary Search: Finding a Number in Sorted List

Index		2	
List		12	

Is **12 == 7?** Yes, Greater, Less? **Greater**

We are thinking of
12

Calculate middle index
mid = (first + last) // 2

1. **mid** = (0 + 15) // 2 = **7**
2. **mid** = (0 + 6) // 2 = **3**
3. **mid** = (0 + 2) // 2 = **1**
4. **mid** = (2 + 2) // 2 = **2**

Binary Search: Finding a Number in Sorted List

Index		2
List		12

Is **12 == 7?** Yes, Greater, Less? **Greater**

We are thinking of
12

Calculate middle index
mid = (first + last) // 2

1. mid = (0 + 15) // 2 = **7**

2. mid = (0 + 6) // 2 = **3**

3. mid = (0 + 2) // 2 = **1**

4. mid = (2 + 2) // 2 = **2**

Binary Search: Finding a Number in Sorted List

Index		2	
List		12	

Is **12 == 12?** Yes, Greater, Less? **Yes**

We are thinking of

12

Calculate middle index

mid = (first + last) // 2

1. **mid** = (0 + 15) // 2 = **7**
2. **mid** = (0 + 6) // 2 = **3**
3. **mid** = (0 + 2) // 2 = **1**
4. **mid** = (2 + 2) // 2 = **2**

Binary Search: Finding a Number in Sorted List

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
List	3	7	12	18	23	27	30	35	40	45	50	54	60	65	70	72

Input Size: **16**

We are thinking of
12

Calculate middle index
mid = (first + last) // 2

1. mid = (0 + 15) // 2 = **7**

2. mid = (0 + 6) // 2 = **3**

3. mid = (0 + 2) // 2 = **1**

4. mid = (2 + 2) // 2 = **2**

Binary Search: Finding a Number in Sorted List

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
List	3	7	12	18	23	27	30	35	40	45	50	54	60	65	70	72

Input Size: **16**

Total Comparisons: **4**

We are thinking of

12

Calculate middle index

mid = (first + last) // 2

1. **mid** = (0 + 15) // 2 = **7**
2. **mid** = (0 + 6) // 2 = **3**
3. **mid** = (0 + 2) // 2 = **1**
4. **mid** = (2 + 2) // 2 = **2**

Binary Search: Finding a Number in Sorted List

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
List	3	7	12	18	23	27	30	35	40	45	50	54	60	65	70	72

We are thinking of
12

Calculate middle index
mid = (first + last) // 2

1. **mid** = (0 + 15) // 2 = **7**
2. **mid** = (0 + 6) // 2 = **3**
3. **mid** = (0 + 2) // 2 = **1**
4. **mid** = (2 + 2) // 2 = **2**

Input Size: **16**

Total Comparisons: **4**

Linear Search: Comparisons $\propto$ Input Size

Binary Search: Finding a Number in Sorted List

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
List	3	7	12	18	23	27	30	35	40	45	50	54	60	65	70	72

We are thinking of
12

Calculate middle index
mid = (first + last) // 2

1. mid = (0 + 15) // 2 = **7**

2. mid = (0 + 6) // 2 = **3**

3. mid = (0 + 2) // 2 = **1**

4. mid = (2 + 2) // 2 = **2**

Input Size: **16**

Total Comparisons: **4**

Linear Search: Comparisons ∝ Input Size

1

Binary Search: Finding a Number in Sorted List

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
List	3	7	12	18	23	27	30	35	40	45	50	54	60	65	70	72

We are thinking of
12

Calculate middle index
mid = (first + last) // 2

1. **mid** = (0 + 15) // 2 = **7**
2. **mid** = (0 + 6) // 2 = **3**
3. **mid** = (0 + 2) // 2 = **1**
4. **mid** = (2 + 2) // 2 = **2**

Input Size: **16**

Total Comparisons: **4**

Linear Search: Comparisons $\propto$ Input Size

1 **1**

Binary Search: Finding a Number in Sorted List

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
List	3	7	12	18	23	27	30	35	40	45	50	54	60	65	70	72

We are thinking of
12

Calculate middle index
mid = (first + last) // 2

1. mid = (0 + 15) // 2 = **7**

2. mid = (0 + 6) // 2 = **3**

3. mid = (0 + 2) // 2 = **1**

4. mid = (2 + 2) // 2 = **2**

Input Size: **16**

Total Comparisons: **4**

Linear Search: Comparisons ∝ Input Size

1	**1**
10	**10**

Binary Search: Finding a Number in Sorted List

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
List	3	7	12	18	23	27	30	35	40	45	50	54	60	65	70	72

We are thinking of
12

Calculate middle index
mid = (first + last) // 2

1. **mid** = (0 + 15) // 2 = **7**
2. **mid** = (0 + 6) // 2 = **3**
3. **mid** = (0 + 2) // 2 = **1**
4. **mid** = (2 + 2) // 2 = **2**

Input Size: **16**

Total Comparisons: **4**

Linear Search: Comparisons ∝ Input Size

1	**1**
10	**10**
	n

Binary Search: Finding a Number in Sorted List

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
List	3	7	12	18	23	27	30	35	40	45	50	54	60	65	70	72

We are thinking of
12

Calculate middle index
mid = (first + last) // 2

1. mid = (0 + 15) // 2 = **7**

2. mid = (0 + 6) // 2 = **3**

3. mid = (0 + 2) // 2 = **1**

4. mid = (2 + 2) // 2 = **2**

Input Size: **16**

Total Comparisons: **4**

Linear Search: Comparisons $\propto$ Input Size

1 **1**

10 **10**

n **n**

Binary Search: Finding a Number in Sorted List

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
List	3	7	12	18	23	27	30	35	40	45	50	54	60	65	70	72

We are thinking of
12

Calculate middle index
mid = (first + last) // 2

1. mid = (0 + 15) // 2 = **7**

2. mid = (0 + 6) // 2 = **3**

3. mid = (0 + 2) // 2 = **1**

4. mid = (2 + 2) // 2 = **2**

Input Size: **16**

Total Comparisons: **4**

Linear Search: Comparisons ∝ Input Size

1 **1**

10 **10**

n **n** ⇒ **O(n)**

Binary Search: Finding a Number in Sorted List

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
List	3	7	12	18	23	27	30	35	40	45	50	54	60	65	70	72

We are thinking of
12

Calculate middle index
mid = (first + last) // 2

1. mid = (0 + 15) // 2 = **7**

2. mid = (0 + 6) // 2 = **3**

3. mid = (0 + 2) // 2 = **1**

4. mid = (2 + 2) // 2 = **2**

Input Size: **16**

Total Comparisons: **4**

Linear Search: Comparisons ∝ Input Size

1 | **1**

10 | **10**

n | **n** ⇒ **O(n)**

Binary Search: Comparisons **?** Input Size

Binary Search: Finding a Number in Sorted List

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
List	3	7	12	18	23	27	30	35	40	45	50	54	60	65	70	72

We are thinking of
12

Calculate middle index
mid = (first + last) // 2

1. mid = (0 + 15) // 2 = **7**

2. mid = (0 + 6) // 2 = **3**

3. mid = (0 + 2) // 2 = **1**

4. mid = (2 + 2) // 2 = **2**

Input Size: **16**

Total Comparisons: **4**

Linear Search: Comparisons ∝ Input Size

1 **1**

10 **10**

n **n** ⇒ **O(n)**

Binary Search: Comparisons **?** Input Size

16

Binary Search: Finding a Number in Sorted List

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
List	3	7	12	18	23	27	30	35	40	45	50	54	60	65	70	72

We are thinking of
12

Calculate middle index
mid = (first + last) // 2

1. mid = (0 + 15) // 2 = **7**

2. mid = (0 + 6) // 2 = **3**

3. mid = (0 + 2) // 2 = **1**

4. mid = (2 + 2) // 2 = **2**

Input Size: **16**

Total Comparisons: **4**

Linear Search: Comparisons $\propto$ Input Size

Comparisons	Input Size
1	**1**
10	**10**
n	**n**

$\Rightarrow$ **O(n)**

Binary Search: Comparisons **?** Input Size

Comparisons	Input Size
4	**16**

Binary Search: Finding a Number in Sorted List

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
List	3	7	12	18	23	27	30	35	40	45	50	54	60	65	70	72

We are thinking of
12

Calculate middle index
mid = (first + last) // 2

1. **mid** = (0 + 15) // 2 = **7**
2. **mid** = (0 + 6) // 2 = **3**
3. **mid** = (0 + 2) // 2 = **1**
4. **mid** = (2 + 2) // 2 = **2**

Input Size: **16**

Total Comparisons: **4**

Linear Search: Comparisons ∝ Input Size

Comparisons	Input Size
1	**1**
10	**10**
n	**n** ⇒ **O(n)**

Binary Search: Comparisons **?** Input Size

Comparisons	Input Size
4	**16**
?	**n**

Binary Search: Finding a Number in Sorted List

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
List	3	7	12	18	23	27	30	35	40	45	50	54	60	65	70	72

We are thinking of
12

Calculate middle index
mid = (first + last) // 2

1. **mid** = (0 + 15) // 2 = **7**
2. **mid** = (0 + 6) // 2 = **3**
3. **mid** = (0 + 2) // 2 = **1**
4. **mid** = (2 + 2) // 2 = **2**

Input Size: **16**

Total Comparisons: **4**

Linear Search: Comparisons $\propto$ Input Size

Comparisons	Input Size	
1	**1**	
10	**10**	
n	**n**	$\Rightarrow$ **O(n)**

Binary Search: Comparisons **?** Input Size

Comparisons	Input Size	
4	**16**	
?	**n**	$\Rightarrow$ $\boxed{\textbf{O(lgn)}}$

Binary Search: Finding a Number in Sorted List

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
List	3	7	12	18	23	27	30	35	40	45	50	54	60	65	70	72

We are thinking of
12

Calculate middle index
mid = (first + last) // 2

1. **mid** = (0 + 15) // 2 = **7**
2. **mid** = (0 + 6) // 2 = **3**
3. **mid** = (0 + 2) // 2 = **1**
4. **mid** = (2 + 2) // 2 = **2**

Input Size: **16**

Total Comparisons: **4**

Linear Search: Comparisons ∝ Input Size

1 **1**

10 **10**

n **n** ⇒ **O(n)**

What is n?

Binary Search: Comparisons **?** Input Size

4 **16**

? **n** ⇒ $\boxed{\textbf{O(lgn)}}$

Binary Search: Finding a Number in Sorted List

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
List	3	7	12	18	23	27	30	35	40	45	50	54	60	65	70	72

We are thinking of
12

Calculate middle index
mid = (first + last) // 2

1. **mid** = (0 + 15) // 2 = **7**
2. **mid** = (0 + 6) // 2 = **3**
3. **mid** = (0 + 2) // 2 = **1**
4. **mid** = (2 + 2) // 2 = **2**

What is n? → **16**

Input Size: **16**

Total Comparisons: **4**

Linear Search: Comparisons ∝ Input Size

1 **1**

10 **10**

n **n** ⇒ **O(n)**

Binary Search: Comparisons **?** Input Size

4 **16**

? **n** ⇒ **O(lgn)**

Binary Search: Finding a Number in Sorted List

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
List	3	7	12	18	23	27	30	35	40	45	50	54	60	65	70	72

We are thinking of
12

Calculate middle index
mid = (first + last) // 2

1. mid = (0 + 15) // 2 = **7**

2. mid = (0 + 6) // 2 = **3**

3. mid = (0 + 2) // 2 = **1**

4. mid = (2 + 2) // 2 = **2**

Input Size: **16**

Total Comparisons: **4**

Linear Search: Comparisons ∝ Input Size

1 **1**

10 **10**

n **n** ⇒ **O(n)**

What is n? → **16**
Comparisons?

Binary Search: Comparisons **?** Input Size

4 **16**

? **n** ⇒ **O(lgn)**

Binary Search: Finding a Number in Sorted List

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
List	3	7	12	18	23	27	30	35	40	45	50	54	60	65	70	72

We are thinking of
12

Calculate middle index
mid = (first + last) // 2

1. mid = (0 + 15) // 2 = **7**
2. mid = (0 + 6) // 2 = **3**
3. mid = (0 + 2) // 2 = **1**
4. mid = (2 + 2) // 2 = **2**

Input Size: **16**
Total Comparisons: **4**

Linear Search: Comparisons ∝ Input Size

1 **1**
10 **10**
n **n** ⇒ **O(n)**

What is n? → **16**
Comparisons? → **4**

Binary Search: Comparisons **?** Input Size

4 **16**
? **n** ⇒ $\boxed{\textbf{O(lgn)}}$

Binary Search: Finding a Number in Sorted List

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
List	3	7	12	18	23	27	30	35	40	45	50	54	60	65	70	72

We are thinking of
12

Calculate middle index
mid = (first + last) // 2

1. **mid** = (0 + 15) // 2 = **7**
2. **mid** = (0 + 6) // 2 = **3**
3. **mid** = (0 + 2) // 2 = **1**
4. **mid** = (2 + 2) // 2 = **2**

Input Size: **16**

Total Comparisons: **4**

Linear Search: Comparisons ∝ Input Size

1 **1**

10 **10**

n **n** ⇒ **O(n)**

What is n? → **16**
Comparisons? → **4**
What is lgn → **lg16**?

Binary Search: Comparisons **?** Input Size

4 **16**

? **n** ⇒ $\boxed{\textbf{O(lgn)}}$

Binary Search: Finding a Number in Sorted List

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
List	3	7	12	18	23	27	30	35	40	45	50	54	60	65	70	72

We are thinking of
12

Calculate middle index
mid = (first + last) // 2

1. **mid** = (0 + 15) // 2 = **7**
2. **mid** = (0 + 6) // 2 = **3**
3. **mid** = (0 + 2) // 2 = **1**
4. **mid** = (2 + 2) // 2 = **2**

Input Size: **16**

Total Comparisons: **4**

Linear Search: Comparisons ∝ Input Size

Comparisons	Input Size	
1	1	
10	10	
n	n	⇒ O(n)

What is n? → **16**
Comparisons? → **4**
What is lgn → **lg16**?
Must know base?

Binary Search: Comparisons **?** Input Size

Comparisons	Input Size	
4	16	
?	n	⇒ $\boxed{O(\lg n)}$

Binary Search: Finding a Number in Sorted List

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
List	3	7	12	18	23	27	30	35	40	45	50	54	60	65	70	72

We are thinking of
12

Calculate middle index
mid = (first + last) // 2

1. mid = (0 + 15) // 2 = **7**
2. mid = (0 + 6) // 2 = **3**
3. mid = (0 + 2) // 2 = **1**
4. mid = (2 + 2) // 2 = **2**

Input Size: **16**
Total Comparisons: **4**

Linear Search: Comparisons ∝ Input Size

Comparisons	Input Size	
1	**1**	
10	**10**	
n	**n**	⇒ **O(n)**

What is n? → **16**
Comparisons? → **4**
What is lgn → **lg16**?
Must know base? → **2**

Binary Search: Comparisons **?** Input Size

Comparisons	Input Size	
4	**16**	
?	**n**	⇒ $\boxed{\textbf{O(lgn)}}$

Binary Search: Finding a Number in Sorted List

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
List	3	7	12	18	23	27	30	35	40	45	50	54	60	65	70	72

We are thinking of
12

Calculate middle index
mid = (first + last) // 2

1. **mid** = (0 + 15) // 2 = **7**
2. **mid** = (0 + 6) // 2 = **3**
3. **mid** = (0 + 2) // 2 = **1**
4. **mid** = (2 + 2) // 2 = **2**

Input Size: **16**

Total Comparisons: **4**

Linear Search: Comparisons ∝ Input Size

1 **1**

10 **10**

n **n** ⇒ **O(n)**

Binary Search: Comparisons **?** Input Size

4 **16**

? **n** ⇒ $\boxed{\textbf{O(lgn)}}$

What is n? → **16**
Comparisons? → **4**
What is lgn → **lg16**?
Must know base? → **2**
What is lgn → $\mathbf{lg_2 16}$?

Binary Search: Finding a Number in Sorted List

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
List	3	7	12	18	23	27	30	35	40	45	50	54	60	65	70	72

We are thinking of
12

Calculate middle index
mid = (first + last) // 2

1. mid = (0 + 15) // 2 = **7**

2. mid = (0 + 6) // 2 = **3**

3. mid = (0 + 2) // 2 = **1**

4. mid = (2 + 2) // 2 = **2**

Input Size: **16**

Total Comparisons: **4**

Linear Search: Comparisons ∝ Input Size

Comparisons	Input Size	
1	**1**	
10	**10**	
n	**n**	⇒ **O(n)**

Binary Search: Comparisons **?** Input Size

Comparisons	Input Size	
4	**16**	
?	**n**	⇒ $\boxed{\textbf{O(lgn)}}$

What is n? → **16**
Comparisons? → **4**
What is lgn → **lg16**?
Must know base? → **2**
What is lgn → **$lg_2 16$**? ⇒
What **power of base** (2) will give **16**?

Binary Search: Finding a Number in Sorted List

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
List	3	7	12	18	23	27	30	35	40	45	50	54	60	65	70	72

We are thinking of
12

Calculate middle index
mid = (first + last) // 2

1. mid = (0 + 15) // 2 = **7**

2. mid = (0 + 6) // 2 = **3**

3. mid = (0 + 2) // 2 = **1**

4. mid = (2 + 2) // 2 = **2**

Input Size: **16**

Total Comparisons: **4**

Linear Search: Comparisons ∝ Input Size

1 **1**

10 **10**

n **n** ⇒ **O(n)**

Binary Search: Comparisons **?** Input Size

4 **16**

? **n** ⇒ $\boxed{\textbf{O(lgn)}}$

What is n? → **16**
Comparisons? → **4**
What is lgn → **lg16**?
Must know base? → **2**
What is lgn → $\mathbf{lg_2 16}$? ⇒
What **power of base** (2)
will give **16**? ⇒ **4**

Binary Search: Finding a Number in Sorted List

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
List	3	7	12	18	23	27	30	35	40	45	50	54	60	65	70	72

We are thinking of
12

Calculate middle index
mid = (first + last) // 2

1. mid = (0 + 15) // 2 = **7**
2. mid = (0 + 6) // 2 = **3**
3. mid = (0 + 2) // 2 = **1**
4. mid = (2 + 2) // 2 = **2**

Input Size: **16**
Total Comparisons: **4**

Linear Search: Comparisons ∝ Input Size

Comparisons	Input Size	
1	1	
10	10	
n	n	⇒ O(n)

Binary Search: Comparisons **?** Input Size

Comparisons	Input Size	
4	16	
?	n	⇒ $\boxed{O(\lg n)}$

What is n? → **16**
Comparisons? → **4**
What is lgn → **lg16**?
Must know base? → **2**
What is lgn → $\lg_2 16$? ⇒
What **power of base** (2) will give **16**? ⇒ **4**, i.e. $2^4 = 16$

Binary Search: Finding a Number in Sorted List

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
List	3	7	12	18	23	27	30	35	40	45	50	54	60	65	70	72

We are thinking of
12

Calculate middle index
mid = (first + last) // 2

1. mid = (0 + 15) // 2 = **7**

2. mid = (0 + 6) // 2 = **3**

3. mid = (0 + 2) // 2 = **1**

4. mid = (2 + 2) // 2 = **2**

Input Size: **16**

Total Comparisons: **4**

Linear Search: Comparisons ∝ Input Size

Comparisons	Input Size
1	**1**
10	**10**
n	**n**

⇒ **O(n)**

Binary Search: Comparisons **?** Input Size

Comparisons	Input Size
4	**16**
?	**n**

⇒ $\boxed{O(\lg n)}$

What is n? → **16**
Comparisons? → **4**
What is lgn → **lg16**?
Must know base? → **2**
What is lgn → $\lg_2 16$? ⇒
What **power of base** (2) will give **16**? ⇒ **4**, i.e. $2^4 = 16$

$\therefore \lg_2 n = \lg_2 16 = 4$

Binary Search: Finding a Number in Sorted List

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
List	3	7	12	18	23	27	30	35	40	45	50	54	60	65	70	72

We are thinking of
12

Calculate middle index
mid = (first + last) // 2

1. mid = (0 + 15) // 2 = **7**
2. mid = (0 + 6) // 2 = **3**
3. mid = (0 + 2) // 2 = **1**
4. mid = (2 + 2) // 2 = **2**

Input Size: **16**
Total Comparisons: **4**

Linear Search: Comparisons ∝ Input Size

1 **1**
10 **10**
n **n** ⇒ **O(n)**

Binary Search: Comparisons **?** Input Size

4 **16**
? **n** ⇒ $\boxed{\textbf{O(lgn)}}$

log is also written as lg

What is n? → **16**
Comparisons? → **4**
What is lgn → **lg16**?
Must know base? → **2**
What is lgn → $\lg_2 16$? ⇒
What **power of base** (2) will give **16**? ⇒ **4**, i.e. $2^4 = 16$

$\therefore \lg_2 n = \lg_2 16 = 4$

Binary Search: Finding a Number in Sorted List

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
List	3	7	12	18	23	27	30	35	40	45	50	54	60	65	70	72

We are thinking of
12

Calculate middle index
mid = (first + last) // 2

1. mid = (0 + 15) // 2 = **7**
2. mid = (0 + 6) // 2 = **3**
3. mid = (0 + 2) // 2 = **1**
4. mid = (2 + 2) // 2 = **2**

Input Size: **16**
Total Comparisons: **4**

Linear Search: Comparisons ∝ Input Size

Comparisons	Input Size	
1	1	
10	10	
n	n	⇒ O(n)

lgn vs lgn +1

Binary Search: Comparisons **?** Input Size

Comparisons	Input Size	
4	16	
?	n	⇒ $\boxed{O(\lg n)}$

log is also written as lg

What is n? → **16**
Comparisons? → **4**
What is lgn → **lg16**?
Must know base? → **2**
What is lgn → $\lg_2 16$? ⇒
What **power of base** (2) will give **16**? ⇒ **4**, i.e. $2^4 = 16$

$\therefore \lg_2 n = \lg_2 16 = 4$

Linear vs. Binary Search: Comparing Steps with Input Growth

Input Size (n)	Linear Search	Binary Search
8	8	$\lg_2 8$ = **3**
16	16	$\lg_2 16$ =
32	32	$\lg_2 32$ =
64	64	$\lg_2 64$ =
...	...	...
1024	1024	$\lg_2 1024$ =
2048	2048	$\lg_2 2048$ =
...	...	...
1048576	1048576	$\lg_2 1048576$ =

Linear vs. Binary Search: Comparing Steps with Input Growth

Input Size (n)	Linear Search	Binary Search
8	8	$\lg_2 8$ = **3**
16	16	$\lg_2 16$ =
32	32	$\lg_2 32$ =
64	64	$\lg_2 64$ =
...	...	...
1024	1024	$\lg_2 1024$ =
2048	2048	$\lg_2 2048$ =
...	...	...
1048576	1048576	$\lg_2 1048576$ =

Linear vs. Binary Search: Comparing Steps with Input Growth

Input Size (n)	Linear Search	Binary Search
8	8	$\lg_2 8 =$ **3**
16	16	$\lg_2 16 =$ **4**
32	32	$\lg_2 32 =$ **5**
64	64	$\lg_2 64 =$ **6**
…	…	…
1024	1024	$\lg_2 1024 =$
2048	2048	$\lg_2 2048 =$
…	…	…
1048576	1048576	$\lg_2 1048576 =$

Linear vs. Binary Search: Comparing Steps with Input Growth

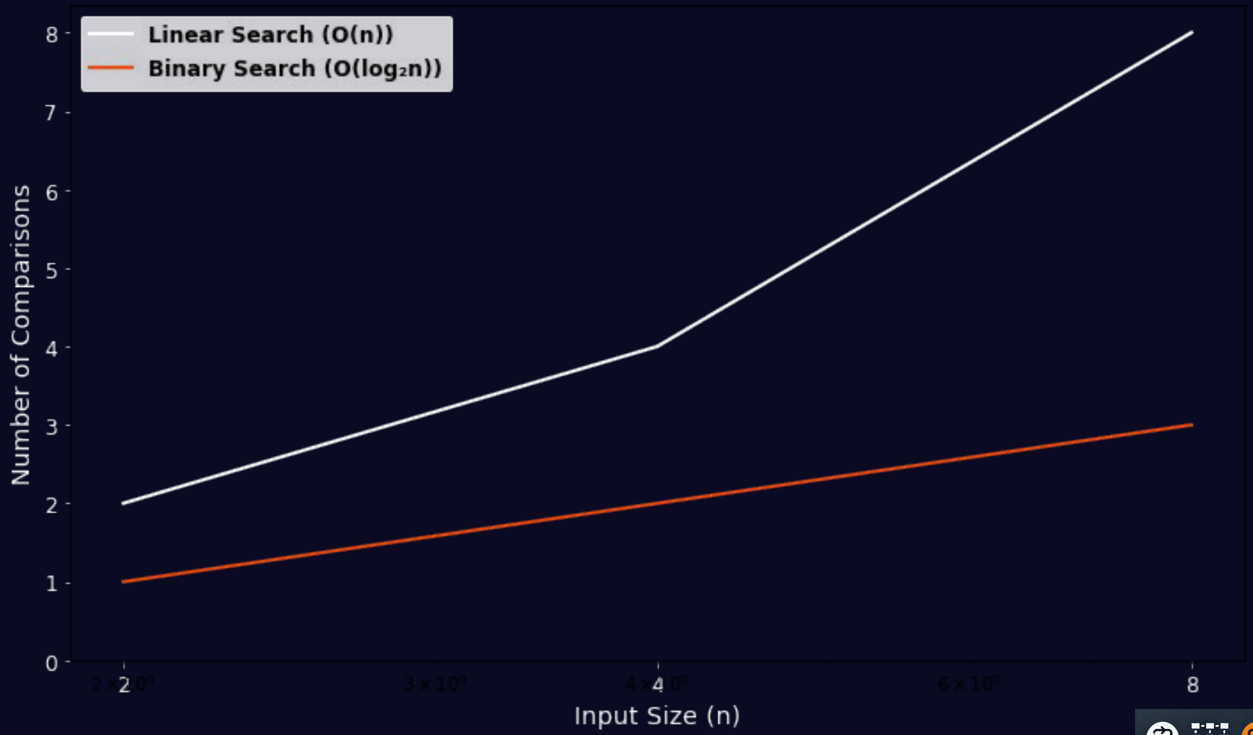
Input Size (n)	Linear Search	Binary Search
8	8	$lg_2 8 =$ **3**
16	16	$lg_2 16 =$ **4**
32	32	$lg_2 32 =$ **5**
64	64	$lg_2 64 =$ **6**
…	…	…
1024	1024	$lg_2 1024 =$ **10**
2048	2048	$lg_2 2048 =$ **11**
…	…	…
1048576	1048576	$lg_2 1048576 =$

Linear vs. Binary Search: Comparing Steps with Input Growth

Input Size (n)	Linear Search	Binary Search
8	8	$lg_2 8$ = **3**
16	16	$lg_2 16$ = **4**
32	32	$lg_2 32$ = **5**
64	64	$lg_2 64$ = **6**
…	…	…
1024	1024	$lg_2 1024$ = **10**
2048	2048	$lg_2 2048$ = **11**
…	…	…
1048576	1048576	$lg_2 1048576$ = **20**

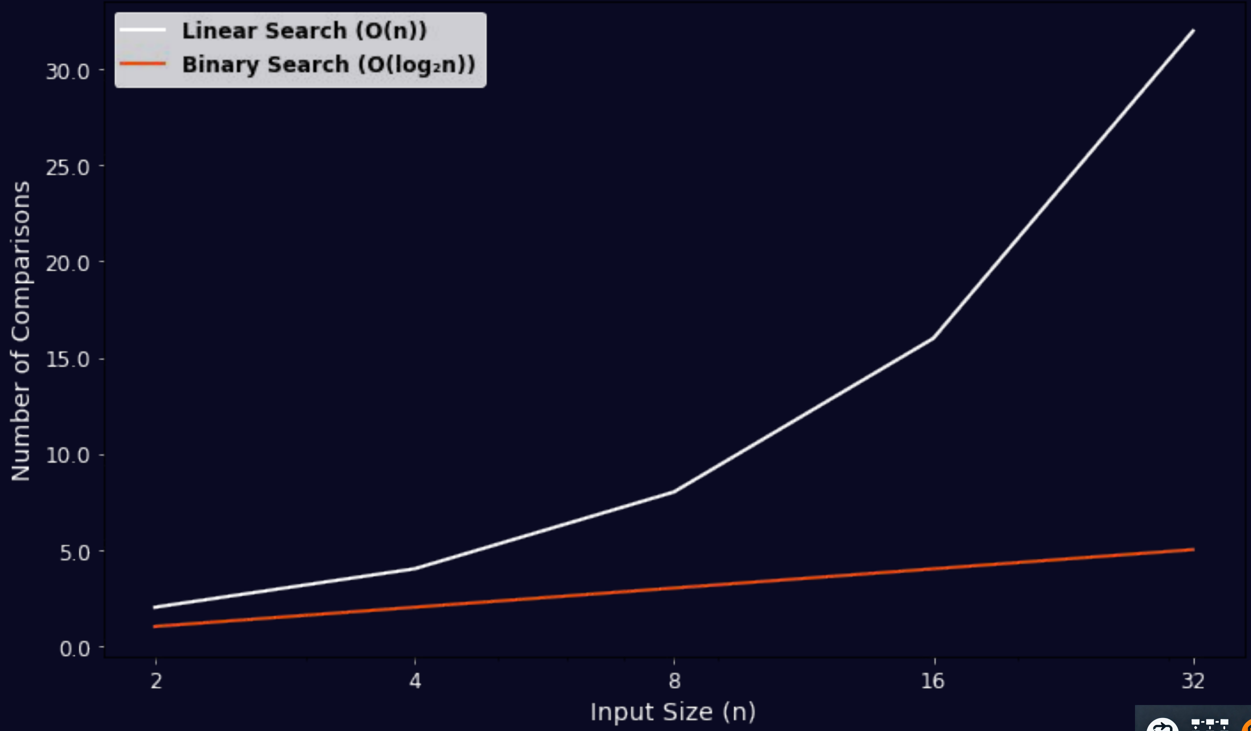
Linear vs. Binary Search: Comparing Steps with Input Growth

Input Size (n)	Linear Search	Binary Search
8	8	$\lg_2 8 =$ **3**
16	16	$\lg_2 16 =$ **4**
32	32	$\lg_2 32 =$ **5**
64	64	$\lg_2 64 =$ **6**
...	...	...
1024	1024	$\lg_2 1024 =$ **10**
2048	2048	$\lg_2 2048 =$ **11**
...	...	...
1048576	1048576	$\lg_2 1048576 =$ **20**



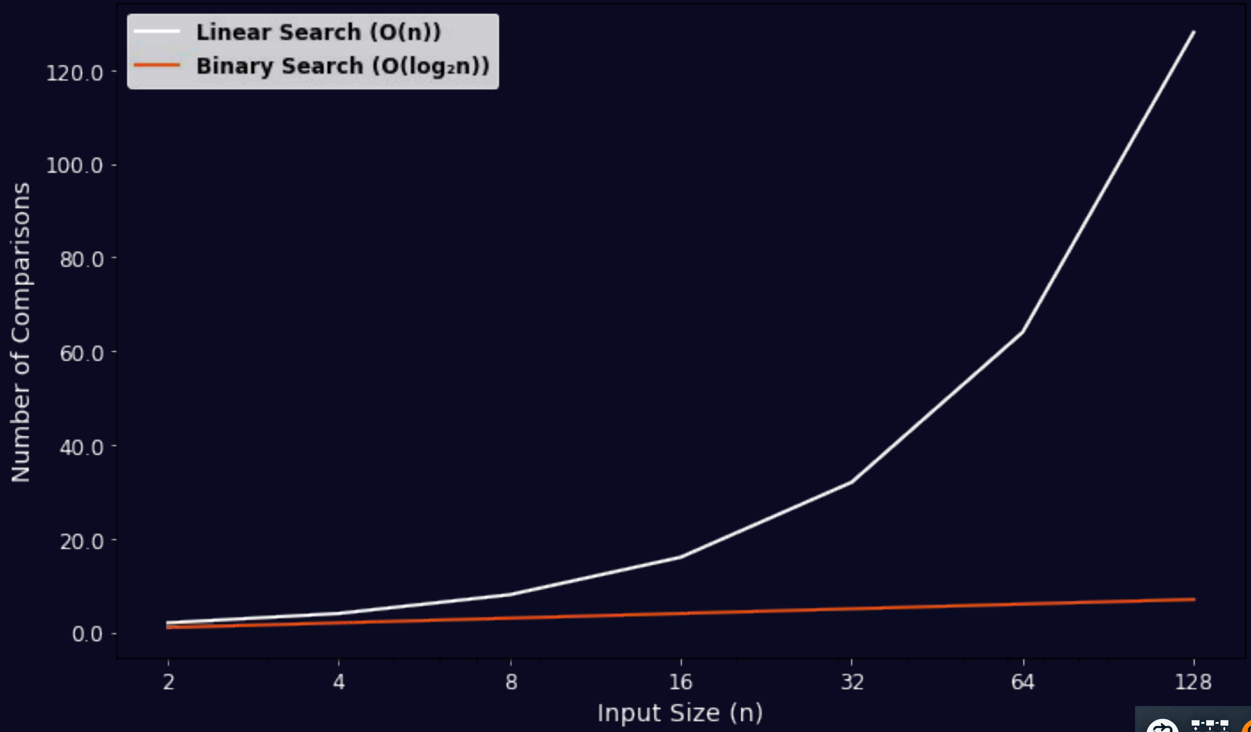
Linear vs. Binary Search: Comparing Steps with Input Growth

Input Size (n)	Linear Search	Binary Search
8	8	$\lg_2 8$ = **3**
16	16	$\lg_2 16$ = **4**
32	32	$\lg_2 32$ = **5**
64	64	$\lg_2 64$ = **6**
...	...	...
1024	1024	$\lg_2 1024$ = **10**
2048	2048	$\lg_2 2048$ = **11**
...	...	...
1048576	1048576	$\lg_2 1048576$ = **20**



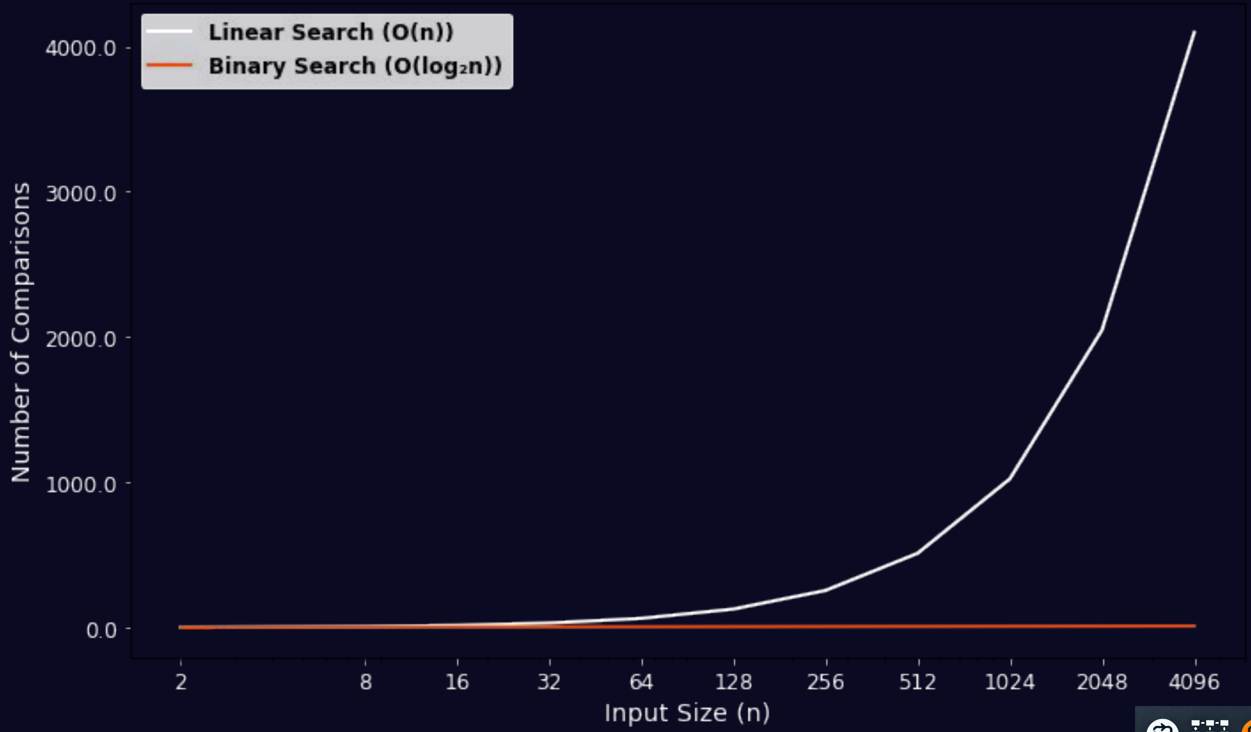
Linear vs. Binary Search: Comparing Steps with Input Growth

Input Size (n)	Linear Search	Binary Search
8	8	$\lg_2 8$ = **3**
16	16	$\lg_2 16$ = **4**
32	32	$\lg_2 32$ = **5**
64	64	$\lg_2 64$ = **6**
...	...	...
1024	1024	$\lg_2 1024$ = **10**
2048	2048	$\lg_2 2048$ = **11**
...	...	...
1048576	1048576	$\lg_2 1048576$ = **20**



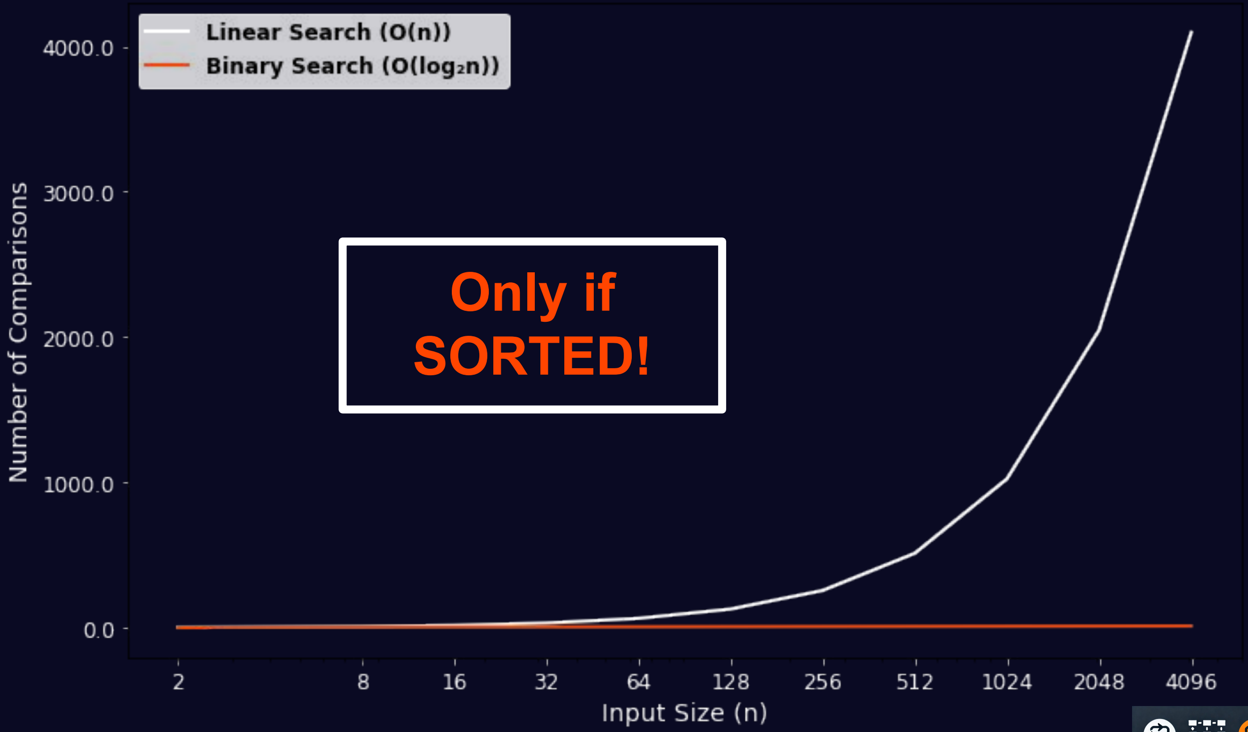
Linear vs. Binary Search: Comparing Steps with Input Growth

Input Size (n)	Linear Search	Binary Search
8	8	$\lg_2 8$ = **3**
16	16	$\lg_2 16$ = **4**
32	32	$\lg_2 32$ = **5**
64	64	$\lg_2 64$ = **6**
…	…	…
1024	1024	$\lg_2 1024$ = **10**
2048	2048	$\lg_2 2048$ = **11**
…	…	…
1048576	1048576	$\lg_2 1048576$ = **20**



Linear vs. Binary Search: Comparing Steps with Input Growth

Input Size (n)	Linear Search	Binary Search
8	8	$lg_2 8$ = **3**
16	16	$lg_2 16$ = **4**
32	32	$lg_2 32$ = **5**
64	64	$lg_2 64$ = **6**
...	...	...
1024	1024	$lg_2 1024$ = **10**
2048	2048	$lg_2 2048$ = **11**
...	...	...
1048576	1048576	$lg_2 1048576$ = **20**



Binary Search Algorithm

Algorithm BinarySearch

Binary Search Algorithm

Algorithm BinarySearch
Input: A sorted list A, a target value

Binary Search Algorithm

Algorithm BinarySearch
Input: A sorted list A, a target value

Index	0	1	2	3	4	5	6	7
List	3	7	12	18	23	27	30	35

Target to Search: **13**

Binary Search Algorithm

Algorithm BinarySearch
Input: A sorted list A, a target value
Output: Index of target in A, or -1 if target is not found

Index	0	1	2	3	4	5	6	7
List	3	7	12	18	23	27	30	35

Target to Search: **13**

Binary Search Algorithm

Algorithm BinarySearch
Input: A sorted list A, a target value
Output: Index of target in A, or -1 if target is not found

Index	0	1	2	3	4	5	6	7
List	3	7	12	18	23	27	30	35

Target to Search: **13**
Expected Output: **-1**

Binary Search Algorithm

Algorithm BinarySearch
Input: A sorted list A, a target value
Output: Index of target in A, or -1 if target is not found

1. Initialize low ← 0, high ← length(A) - 1

low high

Index	0	1	2	3	4	5	6	7
List	3	7	12	18	23	27	30	35

Target to Search: **13**
Expected Output: **-1**
Length of List: **8**

Binary Search Algorithm

low

high

Index	0	1	2	3	4	5	6	7
List	3	7	12	18	23	27	30	35

Algorithm BinarySearch
Input: A sorted list A, a target value
Output: **Index** of **target** in A, or -1 if target is not found

1. Initialize low ← 0, high ← length(A) - 1
2. While low ≤ high do:

Target to Search: **13**
Expected Output: **-1**
Length of List: **8**

Binary Search Algorithm

low high

Index	0	1	2	3	4	5	6	7
List	3	7	12	18	23	27	30	35

Algorithm BinarySearch
Input: A sorted list A, a target value
Output: Index of target in A, or -1 if target is not found

1. Initialize low ← 0, high ← length(A) - 1
2. While low ≤ high do:
    a. Set mid ← (low + high) // 2

Target to Search: **13**
Expected Output: **-1**
Length of List: **8**

Binary Search Algorithm

Algorithm BinarySearch
Input: A sorted list A, a target value
Output: Index of target in A, or -1 if target is not found

1. Initialize low ← 0, high ← length(A) - 1
2. While low ≤ high do:
    a. Set mid ← (low + high) // 2

low mid high

Index	0	1	2	**3**	4	5	6	7
List	3	7	12	18	23	27	30	35

Target to Search: **13**
Expected Output: **-1**
Length of List: **8**

1. mid = (0 + 7) // 2 = **3**

Binary Search Algorithm

low mid high

Index	0	1	2	**3**	4	5	6	7
List	3	7	12	**18**	23	27	30	35

Algorithm BinarySearch
Input: A sorted list A, a target value
Output: **Index** of **target** in A, or **-1** if target is not found

1. Initialize low ← 0, high ← length(A) - 1
2. While low ≤ high do:
    a. Set mid ← (low + high) // 2

Target to Search: **13**
Expected Output: **-1**
Length of List: **8**

1. mid = (0 + 7) // 2 = **3**

Binary Search Algorithm

Algorithm BinarySearch
Input: A sorted list A, a target value
Output: Index of target in A, or -1 if target is not found

1. Initialize low ← 0, high ← length(A) - 1
2. While low ≤ high do:
   a. Set mid ← (low + high) // 2
   b. If A[mid] = target then
      return mid

low mid high

Index	0	1	2	**3**	4	5	6	7
List	3	7	12	**18**	23	27	30	35

Target to Search: **13**
Expected Output: **-1**
Length of List: **8**

1. mid = (0 + 7) // 2 = **3**

Binary Search Algorithm

low mid high

Index	0	1	2	3	4	5	6	7
List	3	7	12	**18**	23	27	30	35

Algorithm BinarySearch
Input: A sorted list A, a target value
Output: **Index** of **target** in A, or **-1** if target is not found

1. Initialize low ← 0, high ← length(A) - 1
2. While low ≤ high do:
   a. Set mid ← (low + high) // 2
   b. If A[mid] = target then
      return mid

Is 18 == 13?

Target to Search: **13**
Expected Output: **-1**
Length of List: **8**

1. mid = (0 + 7) // 2 = **3**

Binary Search Algorithm

low　　　mid　　　high

Index	0	1	2	3	4	5	6	7
List	3	7	12	**18**	23	27	30	35

Algorithm BinarySearch
Input: A sorted list A, a target value
Output: **Index** of **target** in A, or **-1** if target is not found

1. Initialize low ← 0, high ← length(A) - 1
2. While low ≤ high do:
   a. Set mid ← (low + high) // 2
   b. If A[mid] = target then
      return mid
   c. Else If A[mid] > target then
      high ← mid - 1

Target to Search: **13**
Expected Output: **-1**
Length of List: **8**

1. mid = (0 + 7) // 2 = **3**

Binary Search Algorithm

low mid high

Index	0	1	2	**3**	4	5	6	7
List	3	7	12	**18**	23	27	30	35

Algorithm BinarySearch
Input: A sorted list A, a target value
Output: **Index** of **target** in A, or **-1** if target is not found

1. Initialize low ← 0, high ← length(A) - 1
2. While low ≤ high do:
   a. Set mid ← (low + high) // 2
   b. If A[mid] = target then
      return mid
   c. Else If A[mid] > target then
      high ← mid - 1

Is 18 > 13?, YES

Target to Search: **13**
Expected Output: **-1**
Length of List: **8**

1. mid = (0 + 7) // 2 = **3**

Binary Search Algorithm

Algorithm BinarySearch
Input: A sorted list A, a target value
Output: Index of target in A, or -1 if target is not found

1. Initialize low ← 0, high ← length(A) - 1
2. While low ≤ high do:
   a. Set mid ← (low + high) // 2
   b. If A[mid] = target then
      return mid
   c. Else If A[mid] > target then
      high ← mid - 1

Is 18 > 13?, YES

low high

Index	0	1	2
List	3	7	12

Target to Search: **13**
Expected Output: **-1**
Length of List: **8**

1. mid = (0 + 7) // 2 = **3**

Binary Search Algorithm

Index	0	1	2
List	3	7	12

Algorithm BinarySearch
Input: A sorted list A, a target value
Output: Index of target in A, or -1 if target is not found

1. Initialize low ← 0, high ← length(A) - 1
2. While low ≤ high do:
   a. Set mid ← (low + high) // 2
   b. If A[mid] = target then
      return mid
   c. Else If A[mid] > target then
      high ← mid - 1

Target to Search: **13**
Expected Output: **-1**
Length of List: **8**

1. mid = (0 + 7) // 2 = **3**
2. mid = (0 + 2) // 2 = **1**

Binary Search Algorithm

Index	0	1	2
List	3	7	12

low → 0, mid → 1, high → 2

Algorithm BinarySearch
Input: A sorted list A, a target value
Output: Index of target in A, or -1 if target is not found

1. Initialize low ← 0, high ← length(A) - 1
2. While low ≤ high do:
    a. Set mid ← (low + high) // 2
    b. If A[mid] = target then
        return mid
    c. Else If A[mid] > target then
        high ← mid - 1

Target to Search: **13**
Expected Output: **-1**
Length of List: **8**

1. mid = (0 + 7) // 2 = **3**
2. mid = (0 + 2) // 2 = **1**

Binary Search Algorithm

	low	mid	high
Index	0	**1**	2
List	3	**7**	12

Algorithm BinarySearch
Input: A sorted list A, a target value
Output: **Index** of **target** in A, or **-1** if target is not found

1. Initialize low ← 0, high ← length(A) - 1
2. While low ≤ high do:
   a. Set mid ← (low + high) // 2
   b. If A[mid] = target then
      return mid
   c. Else If A[mid] > target then
      high ← mid - 1

Is 7 == 13?

Target to Search: **13**
Expected Output: **-1**
Length of List: **8**

1. mid = (0 + 7) // 2 = **3**
2. mid = (0 + 2) // 2 = **1**

Binary Search Algorithm

mid	low		high

Index	0	1	2
List	3	**7**	12

Algorithm BinarySearch
Input: A sorted list A, a target value
Output: Index of target in A, or -1 if target is not found

1. Initialize low ← 0, high ← length(A) - 1
2. While low ≤ high do:
   a. Set mid ← (low + high) // 2
   b. If A[mid] = target then
      return mid
   c. Else If A[mid] > target then
      high ← mid - 1

Is 7 > 13?

Target to Search: **13**
Expected Output: **-1**
Length of List: **8**

1. mid = (0 + 7) // 2 = **3**
2. mid = (0 + 2) // 2 = **1**

Binary Search Algorithm

mid

low high

Index	0	1	2
List	3	**7**	12

Algorithm BinarySearch
Input: A sorted list A, a target value
Output: Index of target in A, or -1 if target is not found

1. Initialize low ← 0, high ← length(A) - 1
2. While low ≤ high do:
   a. Set mid ← (low + high) // 2
   b. If A[mid] = target then
      return mid
   c. Else If A[mid] > target then
      high ← mid - 1

Target to Search: **13**
Expected Output: **-1**
Length of List: **8**

1. mid = (0 + 7) // 2 = **3**

2. mid = (0 + 2) // 2 = **1**

Is 7 < 13?, **YES**

Binary Search Algorithm

	low	mid	high
Index	0	**1**	2
List	3	**7**	12

Algorithm BinarySearch
Input: A sorted list A, a target value
Output: Index of target in A, or -1 if target is not found

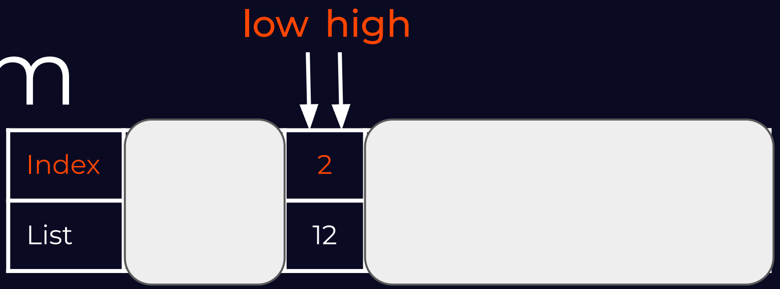
1. Initialize low ← 0, high ← length(A) - 1
2. While low ≤ high do:
   a. Set mid ← (low + high) // 2
   b. If A[mid] = target then
      return mid
   c. Else If A[mid] > target then
      high ← mid - 1
   d. Else
      low ← mid + 1

Is 7 < 13?, **YES**

Target to Search: **13**
Expected Output: **-1**
Length of List: **8**

1. mid = (0 + 7) // 2 = **3**
2. mid = (0 + 2) // 2 = **1**

Binary Search Algorithm



Algorithm BinarySearch
Input: A sorted list A, a target value
Output: Index of target in A, or -1 if target is not found

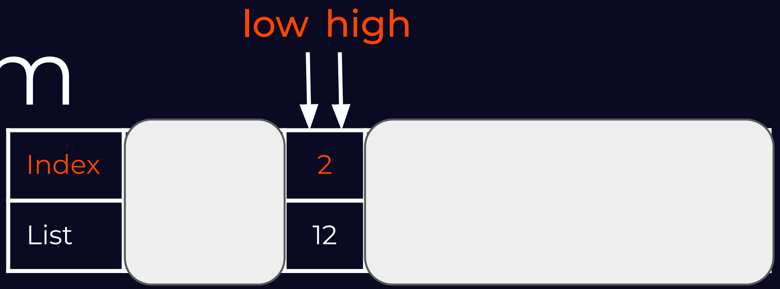
1. Initialize low ← 0, high ← length(A) - 1
2. While low ≤ high do:
   a. Set mid ← (low + high) // 2
   b. If A[mid] = target then
      return mid
   c. Else If A[mid] > target then
      high ← mid - 1
   d. Else
      low ← mid + 1

Target to Search: **13**
Expected Output: **-1**
Length of List: **8**

1. mid = (0 + 7) // 2 = **3**
2. mid = (0 + 2) // 2 = **1**

Is 7 < 13?, **YES**

Binary Search Algorithm



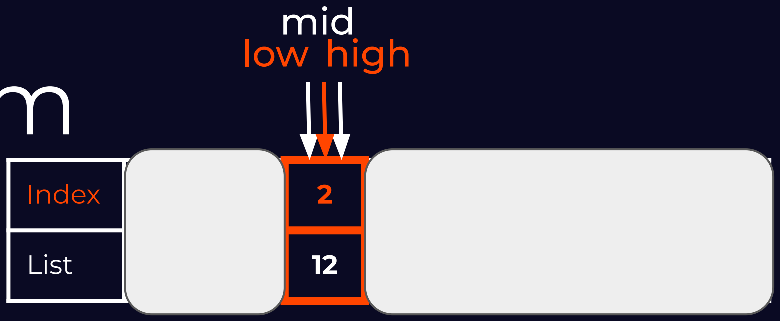
Algorithm BinarySearch
Input: A sorted list A, a target value
Output: Index of target in A, or -1 if target is not found

1. Initialize low ← 0, high ← length(A) - 1
2. While low ≤ high do:
    a. Set mid ← (low + high) // 2
    b. If A[mid] = target then
        return mid
    c. Else If A[mid] > target then
        high ← mid - 1
    d. Else
        low ← mid + 1

Target to Search: **13**
Expected Output: **-1**
Length of List: **8**

1. mid = (0 + 7) // 2 = **3**
2. mid = (0 + 2) // 2 = **1**
3. mid = (2 + 2) // 2 = **2**

Binary Search Algorithm



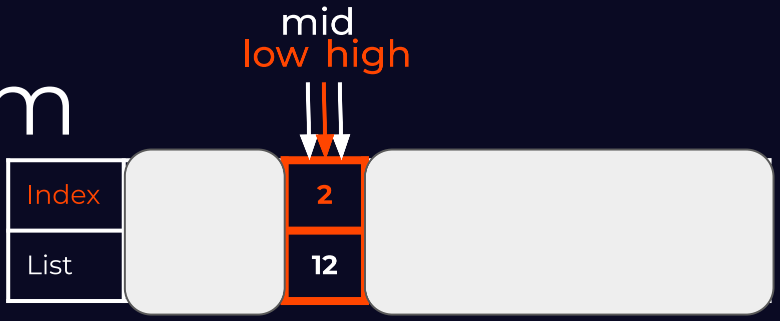
Algorithm BinarySearch
Input: A sorted list A, a target value
Output: Index of target in A, or -1 if target is not found

1. Initialize low ← 0, high ← length(A) - 1
2. While low ≤ high do:
    a. Set mid ← (low + high) // 2
    b. If A[mid] = target then
        return mid
    c. Else If A[mid] > target then
        high ← mid - 1
    d. Else
        low ← mid + 1

Target to Search: **13**
Expected Output: **-1**
Length of List: **8**

1. mid = (0 + 7) // 2 = **3**
2. mid = (0 + 2) // 2 = **1**
3. mid = (2 + 2) // 2 = **2**

Binary Search Algorithm



Algorithm BinarySearch
Input: A sorted list A, a target value
Output: Index of target in A, or -1 if target is not found

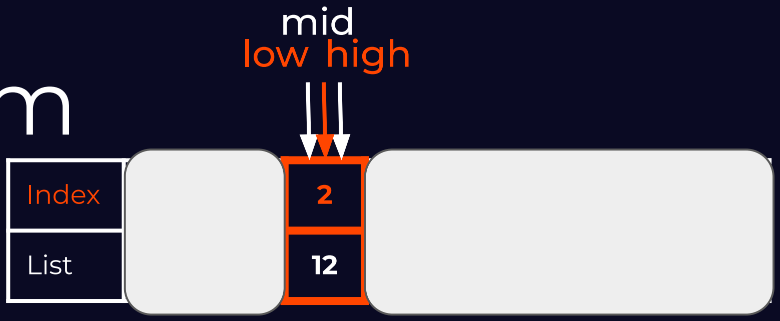
1. Initialize low ← 0, high ← length(A) - 1
2. While low ≤ high do:
   a. Set mid ← (low + high) // 2
   b. If A[mid] = target then
      return mid
   c. Else If A[mid] > target then
      high ← mid - 1
   d. Else
      low ← mid + 1

Is 12 == 13?

Target to Search: **13**
Expected Output: **-1**
Length of List: **8**

1. mid = (0 + 7) // 2 = **3**
2. mid = (0 + 2) // 2 = **1**
3. mid = (2 + 2) // 2 = **2**

Binary Search Algorithm



Algorithm BinarySearch
Input: A sorted list A, a target value
Output: Index of target in A, or -1 if target is not found

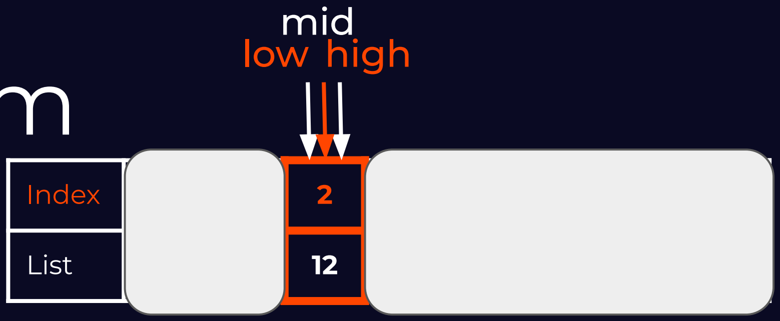
1. Initialize low ← 0, high ← length(A) - 1
2. While low ≤ high do:
   a. Set mid ← (low + high) // 2
   b. If A[mid] = target then
      return mid
   c. Else If A[mid] > target then
      high ← mid - 1
   d. Else
      low ← mid + 1

Is 12 > 13?

Target to Search: **13**
Expected Output: **-1**
Length of List: **8**

1. mid = (0 + 7) // 2 = **3**
2. mid = (0 + 2) // 2 = **1**
3. mid = (2 + 2) // 2 = **2**

Binary Search Algorithm



Algorithm BinarySearch
Input: A sorted list A, a target value
Output: **Index** of **target** in A, or **-1** if target is not found

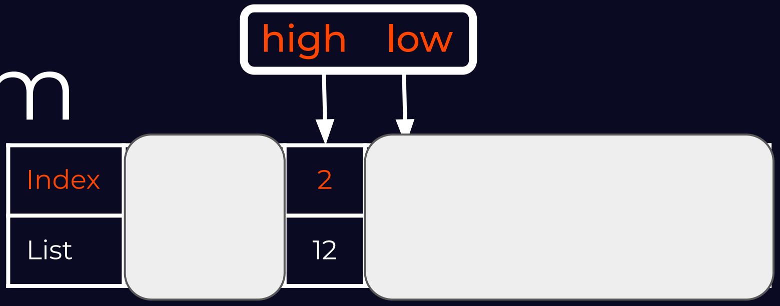
1. Initialize low ← 0, high ← length(A) - 1
2. While low ≤ high do:
    a. Set mid ← (low + high) // 2
    b. If A[mid] = target then
        return mid
    c. Else If A[mid] > target then
        high ← mid - 1
    d. Else
        low ← mid + 1

Is 12 < 13?, **YES**

Target to Search: **13**
Expected Output: **-1**
Length of List: **8**

1. mid = (0 + 7) // 2 = **3**
2. mid = (0 + 2) // 2 = **1**
3. mid = (2 + 2) // 2 = **2**

Binary Search Algorithm



Algorithm BinarySearch
Input: A sorted list A, a target value
Output: Index of target in A, or -1 if target is not found

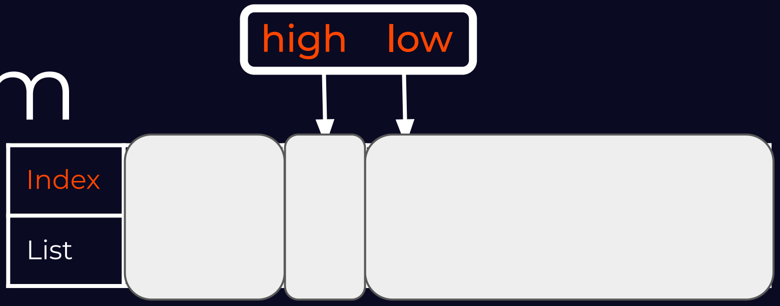
1. Initialize low ← 0, high ← length(A) - 1
2. While low ≤ high do:
    a. Set mid ← (low + high) // 2
    b. If A[mid] = target then
        return mid
    c. Else If A[mid] > target then
        high ← mid - 1
    d. Else
        low ← mid + 1

Is 12 < 13?, **YES**

Target to Search: **13**
Expected Output: **-1**
Length of List: **8**

1. mid = (0 + 7) // 2 = **3**
2. mid = (0 + 2) // 2 = **1**
3. mid = (2 + 2) // 2 = **2**

Binary Search Algorithm



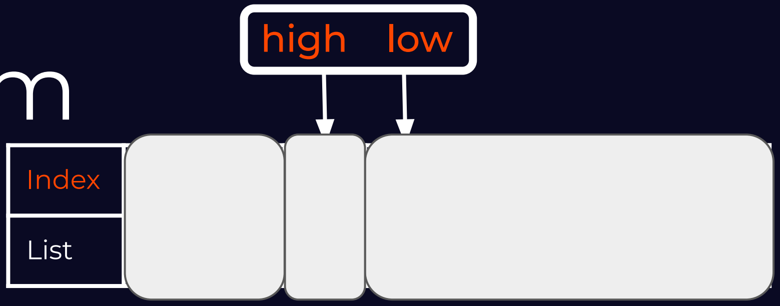
Algorithm BinarySearch
Input: A sorted list A, a target value
Output: Index of target in A, or -1 if target is not found

1. Initialize low ← 0, high ← length(A) - 1
2. While $\boxed{\text{low} \leq \text{high}}$ do:
    a. Set mid ← (low + high) // 2
    b. If A[mid] = target then
        return mid
    c. Else If A[mid] > target then
        high ← mid - 1
    d. Else
        low ← mid + 1

Target to Search: **13**
Expected Output: **-1**
Length of List: **8**

1. mid = (0 + 7) // 2 = **3**
2. mid = (0 + 2) // 2 = **1**
3. mid = (2 + 2) // 2 = **2**

Binary Search Algorithm



Algorithm BinarySearch
Input: A sorted list A, a target value
Output: Index of target in A, or -1 if target is not found

1. Initialize low ← 0, high ← length(A) - 1
2. While low ≤ high do:
   a. Set mid ← (low + high) // 2
   b. If A[mid] = target then
      return mid
   c. Else If A[mid] > target then
      high ← mid - 1
   d. Else
      low ← mid + 1
3. Return -1 // Target not found

Target to Search: **13**
Expected Output: **-1**
Length of List: **8**

1. mid = (0 + 7) // 2 = **3**
2. mid = (0 + 2) // 2 = **1**
3. mid = (2 + 2) // 2 = **2**

Binary Search in Python

Algorithm BinarySearch
Input: A sorted list A, a target value
Output: **Index** of **target**, or **-1** if not found

1. Initialize low ← 0, high ← length(A) - 1
2. While low ≤ high do:
    a. Set mid ← (low + high) // 2
    b. If A[mid] = target then
        return mid
    c. Else If A[mid] > target then
        high ← mid - 1
    d. Else
        low ← mid + 1
3. Return -1 // Target not found

```
def binary_search(A, target):
```

Binary Search in Python

Algorithm BinarySearch
Input: A sorted list A, a target value
Output: **Index** of **target**, or **-1** if not found

1. Initialize low ← 0, high ← length(A) - 1
2. While low ≤ high do:
    a. Set mid ← (low + high) // 2
    b. If A[mid] = target then
        return mid
    c. Else If A[mid] > target then
        high ← mid - 1
    d. Else
        low ← mid + 1
3. Return -1 // Target not found

```
def binary_search(A, target):

    low = 0
    high = len(A) - 1
```

Binary Search in Python

Algorithm BinarySearch
Input: A sorted list A, a target value
Output: **Index** of **target**, or **-1** if not found

1. Initialize low ← 0, high ← length(A) - 1
2. While low ≤ high do:
    a. Set mid ← (low + high) // 2
    b. If A[mid] = target then
        return mid
    c. Else If A[mid] > target then
        high ← mid - 1
    d. Else
        low ← mid + 1
3. Return -1 // Target not found

```
def binary_search(A, target):

    low = 0
    high = len(A) - 1
    while low <= high:
```

Binary Search in Python

Algorithm BinarySearch
Input: A sorted list A, a target value
Output: **Index** of **target**, or **-1** if not found

1. Initialize low ← 0, high ← length(A) - 1
2. While low ≤ high do:
    a. Set mid ← (low + high) // 2
    b. If A[mid] = target then
        return mid
    c. Else If A[mid] > target then
        high ← mid - 1
    d. Else
        low ← mid + 1
3. Return -1 // Target not found

```python
def binary_search(A, target):

    low = 0
    high = len(A) - 1
    while low <= high:
        mid = (low + high) // 2
```

Binary Search in Python

```
Algorithm BinarySearch
Input: A sorted list A, a target value
Output: Index of target, or -1 if not found

1. Initialize low ← 0, high ← length(A) - 1
2. While low ≤ high do:
    a. Set mid ← (low + high) // 2
    b. If A[mid] = target then
        return mid
    c. Else If A[mid] > target then
        high ← mid - 1
    d. Else
        low ← mid + 1
3. Return -1  // Target not found
```

```python
def binary_search(A, target):

    low = 0
    high = len(A) - 1
    while low <= high:
        mid = (low + high) // 2
        if A[mid] == target:
            return mid
```

Binary Search in Python

Algorithm BinarySearch
Input: A sorted list A, a target value
Output: **Index** of **target**, or **-1** if not found

1. Initialize low ← 0, high ← length(A) - 1
2. While low ≤ high do:
   a. Set mid ← (low + high) // 2
   b. If A[mid] = target then
      return mid
   c. Else If A[mid] > target then
      high ← mid - 1
   d. Else
      low ← mid + 1
3. Return -1 // Target not found

```python
def binary_search(A, target):

    low = 0
    high = len(A) - 1
    while low <= high:
        mid = (low + high) // 2
        if A[mid] == target:
            return mid
        elif A[mid] > target:
            high = mid - 1
```

Binary Search in Python

Algorithm BinarySearch
Input: A sorted list A, a target value
Output: **Index** of **target**, or **-1** if not found

1. Initialize low ← 0, high ← length(A) - 1
2. While low ≤ high do:
    a. Set mid ← (low + high) // 2
    b. If A[mid] = target then
        return mid
    c. Else If A[mid] > target then
        high ← mid - 1
    d. Else
        low ← mid + 1
3. Return -1 // Target not found

```python
def binary_search(A, target):

    low = 0
    high = len(A) - 1
    while low <= high:
        mid = (low + high) // 2
        if A[mid] == target:
            return mid
        elif A[mid] > target:
            high = mid - 1
        else:
            low = mid + 1
```

Binary Search in Python

Algorithm BinarySearch
Input: A sorted list A, a target value
Output: **Index** of **target**, or **-1** if not found

1. Initialize low ← 0, high ← length(A) - 1
2. While low ≤ high do:
    a. Set mid ← (low + high) // 2
    b. If A[mid] = target then
        return mid
    c. Else If A[mid] > target then
        high ← mid - 1
    d. Else
        low ← mid + 1
3. Return -1 // Target not found

```python
def binary_search(A, target):

    low = 0
    high = len(A) - 1
    while low <= high:
        mid = (low + high) // 2
        if A[mid] == target:
            return mid
        elif A[mid] > target:
            high = mid - 1
        else:
            low = mid + 1
    return -1
```

Binary Search in Python

Algorithm BinarySearch
Input: A sorted list A, a target value
Output: Index of target, or -1 if not found

1. Initialize low ← 0, high ← length(A) - 1
2. While low ≤ high do:
    a. Set mid ← (low + high) // 2
    b. If A[mid] = target then
        return mid
    c. Else If A[mid] > target then
        high ← mid - 1
    d. Else
        low ← mid + 1
3. Return -1 // Target not found

```python
def binary_search(A, target):

    low = 0
    high = len(A) - 1
    while low <= high:
        mid = (low + high) // 2
        if A[mid] == target:
            return mid
        elif A[mid] > target:
            high = mid - 1
        else:
            low = mid + 1
    return -1
```

```python
A = [3, 7, 12, 18, 23, 27, 30, 35]
target = 12
```

Binary Search in Python

Algorithm BinarySearch
Input: A sorted list A, a target value
Output: Index of target, or -1 if not found

1. Initialize low ← 0, high ← length(A) - 1
2. While low ≤ high do:
    a. Set mid ← (low + high) // 2
    b. If A[mid] = target then
        return mid
    c. Else If A[mid] > target then
        high ← mid - 1
    d. Else
        low ← mid + 1
3. Return -1 // Target not found

```python
def binary_search(A, target):

    low = 0
    high = len(A) - 1
    while low <= high:
        mid = (low + high) // 2
        if A[mid] == target:
            return mid
        elif A[mid] > target:
            high = mid - 1
        else:
            low = mid + 1
    return -1
```

```python
A = [3, 7, 12, 18, 23, 27, 30, 35]
target = 12
result = binary_search(A, target)
```

Binary Search in Python

Algorithm BinarySearch
Input: A sorted list A, a target value
Output: Index of target, or -1 if not found

1. Initialize low ← 0, high ← length(A) - 1
2. While low ≤ high do:
    a. Set mid ← (low + high) // 2
    b. If A[mid] = target then
        return mid
    c. Else If A[mid] > target then
        high ← mid - 1
    d. Else
        low ← mid + 1
3. Return -1 // Target not found

```
def binary_search(A, target):

    low = 0
    high = len(A) - 1
    while low <= high:
        mid = (low + high) // 2
        if A[mid] == target:
            return mid
        elif A[mid] > target:
            high = mid - 1
        else:
            low = mid + 1
    return -1
```

```
A = [3, 7, 12, 18, 23, 27, 30, 35]
target = 12
result = binary_search(A, target)
if result != -1:
    print(f'Target found at index {result}')
```

Binary Search in Python

Algorithm BinarySearch
Input: A sorted list A, a target value
Output: Index of target, or -1 if not found

1. Initialize low ← 0, high ← length(A) - 1
2. While low ≤ high do:
    a. Set mid ← (low + high) // 2
    b. If A[mid] = target then
        return mid
    c. Else If A[mid] > target then
        high ← mid - 1
    d. Else
        low ← mid + 1
3. Return -1 // Target not found

```python
def binary_search(A, target):

    low = 0
    high = len(A) - 1
    while low <= high:
        mid = (low + high) // 2
        if A[mid] == target:
            return mid
        elif A[mid] > target:
            high = mid - 1
        else:
            low = mid + 1
    return -1
```

```python
A = [3, 7, 12, 18, 23, 27, 30, 35]
target = 12
result = binary_search(A, target)
if result != -1:
    print(f'Target found at index {result}')
else:
    print('Target not found')
```

Recursion: An Alternative to Iteration

Recursion: An Alternative to Iteration

Factorial of a number **n** is the **product** of all positive integers from **1 to n**.

Recursion: An Alternative to Iteration

Factorial of a number **n** is the **product** of all positive integers from **1 to n**.

Mathematically: $n! = n \times (n - 1) \times (n - 2) \times \ldots \times 1$

Recursion: An Alternative to Iteration

Factorial of a number **n** is the **product** of all positive integers from **1 to n**.

Mathematically: **n!** = n × (n - 1) × (n - 2) × ... × 1

Example: **4!** = 4 × 3 × 2 × 1 = 24

Recursion: An Alternative to Iteration

Factorial of a number **n** is the **product** of all positive integers from **1 to n**.

Mathematically: **n!** = n × (n - 1) × (n - 2) × ... × 1

Example: **4!** = 4 × 3 × 2 × 1 = 24

```
def iterative_factorial(n):
```

Recursion: An Alternative to Iteration

Factorial of a number **n** is the **product** of all positive integers from **1 to n**.

Mathematically: **n!** = n × (n - 1) × (n - 2) × ... × 1

Example: **4!** = 4 × 3 × 2 × 1 = 24

```
def iterative_factorial(n):
    result = 1
```

Recursion: An Alternative to Iteration

Factorial of a number **n** is the **product** of all positive integers from **1 to n**.

Mathematically: **n!** = n × (n - 1) × (n - 2) × ... × 1

Example: **4!** = 4 × 3 × 2 × 1 = 24

```
def iterative_factorial(n):
    result = 1
    for i in range(1, n + 1):
```

Recursion: An Alternative to Iteration

Factorial of a number **n** is the **product** of all positive integers from **1 to n**.

Mathematically: **n!** = n × (n - 1) × (n - 2) × … × 1

Example: **4!** = 4 × 3 × 2 × 1 = 24

```
def iterative_factorial(n):
    result = 1
    for i in range(1, n + 1):
        result = result * i
```

Recursion: An Alternative to Iteration

Factorial of a number **n** is the **product** of all positive integers from **1 to n**.

Mathematically: $n! = n \times (n - 1) \times (n - 2) \times \ldots \times 1$

Example: $4! = 4 \times 3 \times 2 \times 1 = 24$

```
def iterative_factorial(n):
    result = 1
    for i in range(1, n + 1):
        result = result * i
    return result
```

Recursion: An Alternative to Iteration

Factorial of a number **n** is the **product** of all positive integers from **1 to n**.

Mathematically: **n!** = n × (n - 1) × (n - 2) × … × 1

Example: **4!** = 4 × 3 × 2 × 1 = 24

```
def iterative_factorial(n):
    result = 1
    for i in range(1, n + 1):
        result = result * i
    return result
```

```
def iterative_factorial(n):
    result = 1
    for i in range(n, 1, -1):
        result = result * i
    return result
```

Recursion: An Alternative to Iteration

Factorial of a number **n** is the **product** of all positive integers from **1 to n**.

Mathematically: **n!** = n × (n - 1) × (n - 2) × … × 1

Example: **4!** = 4 × 3 × 2 × 1 = 24

```
def iterative_factorial(n):
    result = 1
    for i in range(1, n + 1):
        result = result * i
    return result
```

```
def iterative_factorial(n):
    result = 1
    for i in range(n, 1, -1):
        result = result * i
    return result
```

Recursion: An Alternative to Iteration

Factorial of a number **n** is the **product** of all positive integers from **1 to n**.

Mathematically: **n!** = n × (n - 1) × (n - 2) × … × 1

Example: **4!** = 4 × 3 × 2 × 1 = 24

Alternative:

Factorial(**n**)

```
def iterative_factorial(n):
    result = 1
    for i in range(1, n + 1):
        result = result * i
    return result
```

```
def iterative_factorial(n):
    result = 1
    for i in range(n, 1, -1):
        result = result * i
    return result
```

Recursion: An Alternative to Iteration

Factorial of a number **n** is the **product** of all positive integers from **1 to n**.

Mathematically: **n!** = n × (n - 1) × (n - 2) × … × 1

Example: **4!** = 4 × 3 × 2 × 1 = 24

```
def iterative_factorial(n):
    result = 1
    for i in range(1, n + 1):
        result = result * i
    return result
```

```
def iterative_factorial(n):
    result = 1
    for i in range(n, 1, -1):
        result = result * i
    return result
```

Alternative:

Factorial(**n**)

n * Factorial(**n-1**)

Recursion: An Alternative to Iteration

Factorial of a number **n** is the **product** of all positive integers from **1 to n**.

Mathematically: **n!** = n × (n - 1) × (n - 2) × … × 1

Example: **4!** = 4 × 3 × 2 × 1 = 24

```
def iterative_factorial(n):
    result = 1
    for i in range(1, n + 1):
        result = result * i
    return result
```

```
def iterative_factorial(n):
    result = 1
    for i in range(n, 1, -1):
        result = result * i
    return result
```

Alternative:

Factorial(**n**)

n * Factorial(**n-1**)

(**n-1) *** Factorial(**n-2**)

Recursion: An Alternative to Iteration

Factorial of a number **n** is the **product** of all positive integers from **1 to n**.

Mathematically: **n!** = n × (n - 1) × (n - 2) × … × 1

Example: **4!** = 4 × 3 × 2 × 1 = 24

```
def iterative_factorial(n):
    result = 1
    for i in range(1, n + 1):
        result = result * i
    return result
```

```
def iterative_factorial(n):
    result = 1
    for i in range(n, 1, -1):
        result = result * i
    return result
```

Alternative:

Factorial(**n**)

n * Factorial(**n-1**)

(**n-1) *** Factorial(**n-2**)

…

(**n-n-2) *** Factorial(**1**)

Recursion: An Alternative to Iteration

Factorial of a number **n** is the **product** of all positive integers from **1 to n**.

Mathematically: **n!** = n × (n - 1) × (n - 2) × … × 1

Example: **4!** = 4 × 3 × 2 × 1 = 24

```
def iterative_factorial(n):
    result = 1
    for i in range(1, n + 1):
        result = result * i
    return result
```

```
def iterative_factorial(n):
    result = 1
    for i in range(n, 1, -1):
        result = result * i
    return result
```

Alternative:

```
Factorial(n)
  n * Factorial(n-1)
    (n-1) * Factorial(n-2)
              …
      (n-n-2) * Factorial(1)
        return 1
```

Recursion: An Alternative to Iteration

Factorial of a number **n** is the **product** of all positive integers from **1 to n**.

Mathematically: **n!** = n × (n - 1) × (n - 2) × … × 1

Example: **4!** = 4 × 3 × 2 × 1 = 24

```
def iterative_factorial(n):
    result = 1
    for i in range(1, n + 1):
        result = result * i
    return result
```

```
def iterative_factorial(n):
    result = 1
    for i in range(n, 1, -1):
        result = result * i
    return result
```

Alternative:

Factorial(**n**)

n * Factorial(**n-1**)

(**n-1) *** Factorial(**n-2**)

…

(**n-n-2) *** Factorial(**1**)

return **1**

Alternative:

Factorial(**4**)

Recursion: An Alternative to Iteration

Factorial of a number **n** is the **product** of all positive integers from **1 to n**.

Mathematically: **n!** = n × (n - 1) × (n - 2) × … × 1

Example: **4!** = 4 × 3 × 2 × 1 = 24

```
def iterative_factorial(n):
    result = 1
    for i in range(1, n + 1):
        result = result * i
    return result
```

```
def iterative_factorial(n):
    result = 1
    for i in range(n, 1, -1):
        result = result * i
    return result
```

Alternative:

Factorial(**n**)

n * Factorial(**n-1**)

(**n-1) *** Factorial(**n-2**)

…

(**n-n-2) *** Factorial(**1**)

return **1**

Alternative:

Factorial(**4**)

4 * Factorial(**3**)

Recursion: An Alternative to Iteration

Factorial of a number **n** is the **product** of all positive integers from **1 to n**.

Mathematically: **n!** = n × (n - 1) × (n - 2) × … × 1

Example: **4!** = 4 × 3 × 2 × 1 = 24

```
def iterative_factorial(n):
    result = 1
    for i in range(1, n + 1):
        result = result * i
    return result
```

```
def iterative_factorial(n):
    result = 1
    for i in range(n, 1, -1):
        result = result * i
    return result
```

Alternative:

Factorial(**n**)

n * Factorial(**n-1**)

(**n-1) *** Factorial(**n-2**)

…

(**n-n-2) *** Factorial(**1**)

return **1**

Alternative:

Factorial(**4**)

4 * Factorial(**3**)

3 * Factorial(**2**)

Recursion: An Alternative to Iteration

Factorial of a number **n** is the **product** of all positive integers from **1 to n**.

Mathematically: **n!** = n × (n - 1) × (n - 2) × … × 1

Example: **4!** = 4 × 3 × 2 × 1 = 24

```
def iterative_factorial(n):
    result = 1
    for i in range(1, n + 1):
        result = result * i
    return result
```

```
def iterative_factorial(n):
    result = 1
    for i in range(n, 1, -1):
        result = result * i
    return result
```

Alternative:

Factorial(**n**)

n * Factorial(**n-1**)

(**n-1**) ***** Factorial(**n-2**)

…

(**n-n-2**) ***** Factorial(**1**)

return **1**

Alternative:

Factorial(**4**)

4 * Factorial(**3**)

3 * Factorial(**2**)

2 * Factorial(**1**)

Recursion: An Alternative to Iteration

Factorial of a number **n** is the **product** of all positive integers from **1 to n**.

Mathematically: **n!** = n × (n - 1) × (n - 2) × … × 1

Example: **4!** = 4 × 3 × 2 × 1 = 24

```
def iterative_factorial(n):
    result = 1
    for i in range(1, n + 1):
        result = result * i
    return result
```

```
def iterative_factorial(n):
    result = 1
    for i in range(n, 1, -1):
        result = result * i
    return result
```

Alternative:

Factorial(**n**)

n * Factorial(**n-1**)

(**n-1) *** Factorial(**n-2**)

…

(**n-n-2) *** Factorial(**1**)

return **1**

Alternative:

Factorial(**4**)

4 * Factorial(**3**)

3 * Factorial(**2**)

2 * Factorial(**1**)

return **1**

Recursion: An Alternative to Iteration

Factorial of a number **n** is the **product** of all positive integers from **1 to n**.

Mathematically: **n!** = n × (n - 1) × (n - 2) × … × 1

Example: **4!** = 4 × 3 × 2 × 1 = 24

```
def iterative_factorial(n):
    result = 1
    for i in range(1, n + 1):
        result = result * i
    return result
```

```
def iterative_factorial(n):
    result = 1
    for i in range(n, 1, -1):
        result = result * i
    return result
```

Alternative:

Factorial(**n**)

n * Factorial(**n-1**)

(**n-1**) * Factorial(**n-2**)

…

(**n-n-2**) * Factorial(**1**)

return **1**

Alternative:

Factorial(**4**)

4 * Factorial(**3**)

3 * Factorial(**2**)

2 * Factorial(**1**)

return **1**

Recursive Case

Recursion: An Alternative to Iteration

Factorial of a number **n** is the **product** of all positive integers from **1 to n**.

Mathematically: **n!** = n × (n - 1) × (n - 2) × … × 1

Example: **4!** = 4 × 3 × 2 × 1 = 24

```
def iterative_factorial(n):
    result = 1
    for i in range(1, n + 1):
        result = result * i
    return result
```

```
def iterative_factorial(n):
    result = 1
    for i in range(n, 1, -1):
        result = result * i
    return result
```

Alternative:

Factorial(**n**)

n * Factorial(**n-1**)

(**n-1**) * Factorial(**n-2**)

…

(**n-n-2**) * Factorial(**1**)

return **1**

Alternative:

Factorial(4)

4 * Factorial(3)

3 * Factorial(2)

Recursive Case

2 * Factorial(1)

return **1**

Base Case

Recursion: An Alternative to Iteration

Factorial of a number **n** is the **product** of all positive integers from **1 to n**.

Mathematically: **n!** = n × (n - 1) × (n - 2) × … × 1

Example: **4!** = 4 × 3 × 2 × 1 = 24

```
def iterative_factorial(n):
    result = 1
    for i in range(1, n + 1):
        result = result * i
    return result
```

```
def iterative_factorial(n):
    result = 1
    for i in range(n, 1, -1):
        result = result * i
    return result
```

Alternative:

Factorial(**n**)

n * Factorial(**n-1**)

(**n-1**) ***** Factorial(**n-2**)

…

(**n-n-2**) ***** Factorial(**1**)

return **1**

Alternative:

Factorial(**4**)

4 * Factorial(**3**)

3 * Factorial(**2**)

2 * Factorial(**1**)

return **1**

Recursion: An Alternative to Iteration

Factorial of a number **n** is the **product** of all positive integers from **1 to n**.

Mathematically: **n!** = n × (n - 1) × (n - 2) × … × 1

Example: **4!** = 4 × 3 × 2 × 1 = 24

```
def iterative_factorial(n):
    result = 1
    for i in range(1, n + 1):
        result = result * i
    return result
```

```
def iterative_factorial(n):
    result = 1
    for i in range(n, 1, -1):
        result = result * i
    return result
```

Alternative:

Factorial(**n**)

n * Factorial(**n-1**)

(**n-1) *** Factorial(**n-2**)

…

(**n-n-2) *** Factorial(**1**)

return **1**

Alternative:

Factorial(**4**)

4 * Factorial(**3**)

3 * Factorial(**2**)

2 * Factorial(**1**)

return **1**

Recursion: An Alternative to Iteration

Factorial of a number **n** is the **product** of all positive integers from **1 to n**.

Mathematically: **n!** = n × (n - 1) × (n - 2) × … × 1

Example: **4!** = 4 × 3 × 2 × 1 = 24

```
def iterative_factorial(n):
    result = 1
    for i in range(1, n + 1):
        result = result * i
    return result
```

```
def iterative_factorial(n):
    result = 1
    for i in range(n, 1, -1):
        result = result * i
    return result
```

Alternative:

Factorial(**n**)

n * Factorial(**n-1**)

(**n-1) *** Factorial(**n-2**)

…

(**n-n-2) *** Factorial(**1**)

return **1**

Alternative:

Factorial(**4**)

4 * Factorial(**3**)

3 * Factorial(**2**)

2 * → **1 = 2**

~~return 1~~

Recursion: An Alternative to Iteration

Factorial of a number **n** is the **product** of all positive integers from **1 to n**.

Mathematically: **n!** = n × (n - 1) × (n - 2) × ... × 1

Example: **4!** = 4 × 3 × 2 × 1 = 24

```
def iterative_factorial(n):
    result = 1
    for i in range(1, n + 1):
        result = result * i
    return result
```

```
def iterative_factorial(n):
    result = 1
    for i in range(n, 1, -1):
        result = result * i
    return result
```

Alternative:

Factorial(**n**)

n * Factorial(**n-1**)

(**n-1) *** Factorial(**n-2**)

...

(**n-n-2) *** Factorial(**1**)

return **1**

Alternative:

Factorial(**4**)

4 * Factorial(**3**)

3 * Factorial(**2**)

2 * → **1 = 2**

~~return 1~~

Recursion: An Alternative to Iteration

Factorial of a number **n** is the **product** of all positive integers from **1 to n**.

Mathematically: **n!** = n × (n - 1) × (n - 2) × … × 1

Example: **4!** = 4 × 3 × 2 × 1 = 24

```
def iterative_factorial(n):
    result = 1
    for i in range(1, n + 1):
        result = result * i
    return result
```

```
def iterative_factorial(n):
    result = 1
    for i in range(n, 1, -1):
        result = result * i
    return result
```

Alternative:

Factorial(**n**)

n * Factorial(**n-1**)

(**n-1) *** Factorial(**n-2**)

…

(**n-n-2) *** Factorial(**1**)

return **1**

Alternative:

Factorial(**4**)

4 * Factorial(**3**)

3 * **→ 2 = 6**

2 * ~~Factorial(1)~~ **→ 1 = 2**

~~return 1~~

Recursion: An Alternative to Iteration

Factorial of a number **n** is the **product** of all positive integers from **1 to n**.

Mathematically: **n!** = n × (n - 1) × (n - 2) × … × 1

Example: **4!** = 4 × 3 × 2 × 1 = 24

```
def iterative_factorial(n):
    result = 1
    for i in range(1, n + 1):
        result = result * i
    return result
```

```
def iterative_factorial(n):
    result = 1
    for i in range(n, 1, -1):
        result = result * i
    return result
```

Alternative:

Factorial(**n**)

n * Factorial(**n-1**)

(**n-1**) ***** Factorial(**n-2**)

…

(**n-n-2**) ***** Factorial(**1**)

return **1**

Alternative:

Factorial(**4**)

4 * Factorial(**3**)

3 * [redacted] → **2 = 6**

2 * ~~Factorial(1)~~ → **1 = 2**

~~return 1~~

Recursion: An Alternative to Iteration

Factorial of a number **n** is the **product** of all positive integers from **1 to n**.

Mathematically: **n!** = n × (n - 1) × (n - 2) × … × 1

Example: **4!** = 4 × 3 × 2 × 1 = 24

```
def iterative_factorial(n):
    result = 1
    for i in range(1, n + 1):
        result = result * i
    return result
```

```
def iterative_factorial(n):
    result = 1
    for i in range(n, 1, -1):
        result = result * i
    return result
```

Alternative:

Factorial(**n**)

n * Factorial(**n-1**)

(**n-1**) ***** Factorial(**n-2**)

…

(**n-n-2**) ***** Factorial(**1**)

return **1**

Alternative:

Factorial(**4**)

4 * [redacted] **→ 6 = 24**

3 * ~~Factorial(2)~~ **→ 2 = 6**

2 * ~~Factorial(1)~~ **→ 1 = 2**

~~return 1~~

Recursion: An Alternative to Iteration

Factorial of a number **n** is the **product** of all positive integers from **1 to n**.

Mathematically: **n!** = n × (n - 1) × (n - 2) × … × 1

Example: **4!** = 4 × 3 × 2 × 1 = 24

```
def iterative_factorial(n):
    result = 1
    for i in range(1, n + 1):
        result = result * i
    return result
```

```
def iterative_factorial(n):
    result = 1
    for i in range(n, 1, -1):
        result = result * i
    return result
```

Alternative:

Factorial(**n**)

n * Factorial(**n-1**)

(**n-1**) ***** Factorial(**n-2**)

…

(**n-n-2**) ***** Factorial(**1**)

return **1**

Alternative:

$\boxed{\text{Factorial(4)}}$

4 * → **6 = 24**

3 * ~~Factorial(2)~~ → **2 = 6**

2 * ~~Factorial(1)~~ → **1 = 2**

~~return 1~~

Recursion: An Alternative to Iteration

Factorial of a number **n** is the **product** of all positive integers from **1 to n**.

Mathematically: **n!** = n × (n - 1) × (n - 2) × … × 1

Example: **4!** = 4 × 3 × 2 × 1 = 24

```
def iterative_factorial(n):
    result = 1
    for i in range(1, n + 1):
        result = result * i
    return result
```

```
def iterative_factorial(n):
    result = 1
    for i in range(n, 1, -1):
        result = result * i
    return result
```

Alternative:

Factorial(**n**)

n * Factorial(**n-1**)

(**n-1**) * Factorial(**n-2**)

…

(**n-n-2**) * Factorial(**1**)

return **1**

Alternative:

[redacted] → **24**

4 * ~~Factorial(3)~~ → **6 = 24**

3 * ~~Factorial(2)~~ → **2 = 6**

2 * ~~Factorial(1)~~ → **1 = 2**

~~return 1~~

Recursion: An Alternative to Iteration

Factorial of a number **n** is the **product** of all positive integers from **1 to n**.

Mathematically: **n!** = n × (n - 1) × (n - 2) × … × 1

Example: **4!** = 4 × 3 × 2 × 1 = 24

```
def iterative_factorial(n):
    result = 1
    for i in range(1, n + 1):
        result = result * i
    return result
```

```
def iterative_factorial(n):
    result = 1
    for i in range(n, 1, -1):
        result = result * i
    return result
```

Alternative:

Factorial(**n**)

n * Factorial(**n-1**)

(**n-1**) ***** Factorial(**n-2**)

…

(**n-n-2**) ***** Factorial(**1**)

return **1**

Alternative:

Factorial(**4**)

4 * Factorial(**3**)

3 * Factorial(**2**)

Recursive Case

2 * Factorial(**1**)

return **1**

Base Case

```
def recursive_factorial(n):
```

Recursion: An Alternative to Iteration

Factorial of a number **n** is the **product** of all positive integers from **1 to n**.

Mathematically: **n!** = n × (n - 1) × (n - 2) × … × 1

Example: **4!** = 4 × 3 × 2 × 1 = 24

```
def iterative_factorial(n):
    result = 1
    for i in range(1, n + 1):
        result = result * i
    return result
```

```
def iterative_factorial(n):
    result = 1
    for i in range(n, 1, -1):
        result = result * i
    return result
```

Alternative:

```
Factorial(n)
  n * Factorial(n-1)
    (n-1) * Factorial(n-2)
      ...
        (n-n-2) * Factorial(1)
          return 1
```

Alternative:

```
Factorial(4)
  4 * Factorial(3)
    3 * Factorial(2)
      2 * Factorial(1)
        return 1
```

Recursive Case

Base Case

```
def recursive_factorial(n):
    if n == 1:
        return 1
```

Recursion: An Alternative to Iteration

Factorial of a number **n** is the **product** of all positive integers from **1 to n**.

Mathematically: **n!** = n × (n - 1) × (n - 2) × … × 1

Example: **4!** = 4 × 3 × 2 × 1 = 24

```
def iterative_factorial(n):
    result = 1
    for i in range(1, n + 1):
        result = result * i
    return result
```

```
def iterative_factorial(n):
    result = 1
    for i in range(n, 1, -1):
        result = result * i
    return result
```

Alternative:

Factorial(**n**)
n * Factorial(**n-1**)
(**n-1**) * Factorial(**n-2**)
…
(**n-n-2**) * Factorial(**1**)
return **1**

Alternative:

Factorial(**4**)
4 * Factorial(**3**)
3 * Factorial(**2**)

Recursive Case

2 * Factorial(**1**)
return **1**

Base Case

```
def recursive_factorial(n):
    if n == 1:
        return 1
```

Base Case

Recursion: An Alternative to Iteration

Factorial of a number **n** is the **product** of all positive integers from **1 to n**.

Mathematically: **n!** = n × (n - 1) × (n - 2) × … × 1

Example: **4!** = 4 × 3 × 2 × 1 = 24

```
def iterative_factorial(n):
    result = 1
    for i in range(1, n + 1):
        result = result * i
    return result
```

```
def iterative_factorial(n):
    result = 1
    for i in range(n, 1, -1):
        result = result * i
    return result
```

Alternative:

Factorial(**n**)
  n * Factorial(**n-1**)
    (**n-1**) * Factorial(**n-2**)
      …
        (**n-n-2**) * Factorial(**1**)
          return **1**

Alternative:

Factorial(**4**)
  4 * Factorial(**3**)
    3 * Factorial(**2**)

Recursive Case

      2 * Factorial(**1**)
        return **1**

Base Case

```
def recursive_factorial(n):
    if n == 1:
        return 1
    else:
        return n * recursive_factorial(n - 1)
```

Base Case

Recursive Case

Recursion: An Alternative to Iteration

Factorial of a number **n** is the **product** of all positive integers from **1 to n**.

Mathematically: **n!** = n × (n - 1) × (n - 2) × ... × 1

Example: **4!** = 4 × 3 × 2 × 1 = 24

```
def iterative_factorial(n):
    result = 1
    for i in range(1, n + 1):
        result = result * i
    return result
```

```
def iterative_factorial(n):
    result = 1
    for i in range(n, 1, -1):
        result = result * i
    return result
```

Alternative:

Factorial(**n**)
n * Factorial(**n-1**)
(**n-1**) * Factorial(**n-2**)
...
(**n-n-2**) * Factorial(**1**)
return **1**

Alternative:

Factorial(**4**)
4 * Factorial(**3**)
3 * Factorial(**2**)
2 * Factorial(**1**)
return **1**

Recursive Case

Base Case

```
def recursive_factorial(n):
    if n == 1:
        return 1
    else:
        return n * recursive_factorial(n - 1)
```

Base Case

Recursive Case

Recursion: An Alternative to Iteration

- **Iteration:**
  - Most common approach to problem-solving.

Recursion: An Alternative to Iteration

- **Iteration:**
  - Most common approach to problem-solving.
  - Involves using loops (for, while) to repeatedly execute a block of code.

Recursion: An Alternative to Iteration

- **Iteration:**
  - Most common approach to problem-solving.
  - Involves using loops (for, while) to repeatedly execute a block of code.
  - Efficient in terms of memory but can sometimes be more verbose or harder to follow in complex problems.

Recursion: An Alternative to Iteration

- **Iteration:**
  - Most common approach to problem-solving.
  - Involves using loops (for, while) to repeatedly execute a block of code.
  - Efficient in terms of memory but can sometimes be more verbose or harder to follow in complex problems.
- **Recursion:**
  - Usually solves the problem by redefining the original problem with smaller input size (recursive case) until base case is met.

Recursion: An Alternative to Iteration

- **Iteration:**
  - Most common approach to problem-solving.
  - Involves using loops (for, while) to repeatedly execute a block of code.
  - Efficient in terms of memory but can sometimes be more verbose or harder to follow in complex problems.
- **Recursion:**
  - Usually solves the problem by redefining the original problem with smaller input size (recursive case) until base case is met.
  - Typically requires fewer lines of code and can make complex problems easier to express.

Recursion: An Alternative to Iteration

- **Iteration:**
  - Most common approach to problem-solving.
  - Involves using loops (for, while) to repeatedly execute a block of code.
  - Efficient in terms of memory but can sometimes be more verbose or harder to follow in complex problems.
- **Recursion:**
  - Usually solves the problem by redefining the original problem with smaller input size (recursive case) until base case is met.
  - Typically requires fewer lines of code and can make complex problems easier to express.
  - Function calls and can lead to stack overflow if the depth of recursion is too large.

Converting Iterative Binary Search to Recursive

```
def binary_search(A, target):
    low = 0
    high = len(A) - 1
    while low <= high:
        mid = (low + high) // 2

        if A[mid] == target:
            return mid
        elif A[mid] > target:
            high = mid - 1
        else:
            low = mid + 1

    return -1
```

Converting Iterative Binary Search to Recursive

```
def binary_search(A, target):
    low = 0
    high = len(A) - 1
    while low <= high:
        mid = (low + high) // 2
        if A[mid] == target:
            return mid
        elif A[mid] > target:
            high = mid - 1
        else:
            low = mid + 1
    return -1
```

```
A = [3, 7, 12, 18, 23, 27, 30, 35]
target = 12
result = binary_search(A, target)
if result != -1:
    print(f'Target found at index {result}')
else:
    print('Target not found')
```

Converting Iterative Binary Search to Recursive

```
def binary_search(A, target):
    low = 0
    high = len(A) - 1
    while low <= high:
        mid = (low + high) // 2

        if A[mid] == target:
            return mid
        elif A[mid] > target:
            high = mid - 1
        else:
            low = mid + 1

    return -1
```

```
A = [3, 7, 12, 18, 23, 27, 30, 35]
target = 12
result = binary_search(A, target)
if result != -1:
    print(f'Target found at index {result}')
else:
    print('Target not found')
```

Converting Iterative Binary Search to Recursive

```
def binary_search(A, target):
    low = 0
    high = len(A) - 1
    while low <= high:
        mid = (low + high) // 2

        if A[mid] == target:
            return mid
        elif A[mid] > target:
            high = mid - 1
        else:
            low = mid + 1

    return -1
```

```
A = [3, 7, 12, 18, 23, 27, 30, 35]
target = 12
result = binary_search(A, target, 0, len(A)-1)
if result != -1:
    print(f'Target found at index {result}')
else:
    print('Target not found')
```

Converting Iterative Binary Search to Recursive

```
def binary_search(A, target, low, high):
    while low <= high:
        mid = (low + high) // 2
        if A[mid] == target:
            return mid
        elif A[mid] > target:
            high = mid - 1
        else:
            low = mid + 1
    return -1
```

```
A = [3, 7, 12, 18, 23, 27, 30, 35]
target = 12
result = binary_search(A, target, 0, len(A)-1)
if result != -1:
    print(f'Target found at index {result}')
else:
    print('Target not found')
```

Converting Iterative Binary Search to Recursive

```
def binary_search(A, target, low, high):
    while low <= high:
        mid = (low + high) // 2
        if A[mid] == target:
            return mid
        elif A[mid] > target:
            high = mid - 1
        else:
            low = mid + 1
    return -1
```

```
A = [3, 7, 12, 18, 23, 27, 30, 35]
target = 12
result = binary_search(A, target, 0, len(A)-1)
if result != -1:
    print(f'Target found at index {result}')
else:
    print('Target not found')
```

Converting Iterative Binary Search to Recursive

```
def binary_search(A, target, low, high):

    while low <= high:
        mid = (low + high) // 2

        if A[mid] == target:
            return mid
        elif A[mid] > target:
            high = mid - 1
        else:
            low = mid + 1

    return -1
```

```
A = [3, 7, 12, 18, 23, 27, 30, 35]
target = 12
result = binary_search(A, target, 0, len(A)-1)
if result != -1:
    print(f'Target found at index {result}')
else:
    print('Target not found')
```

```
A = [3, 7, 12, 18, 23, 27, 30, 35]
target = 12
result = recursive_binary_search(A, target, 0, len(A)-1)
if result != -1:
    print(f'Target found at index {result}')
else:
    print('Target not found')
```

Converting Iterative Binary Search to Recursive

```
def binary_search(A, target, low, high):
    while low <= high:
        mid = (low + high) // 2
        if A[mid] == target:
            return mid
        elif A[mid] > target:
            high = mid - 1
        else:
            low = mid + 1
    return -1
```

```
def recursive_binary_search(A, target, low, high):
```

```
A = [3, 7, 12, 18, 23, 27, 30, 35]
target = 12
result = binary_search(A, target, 0, len(A)-1)
if result != -1:
    print(f'Target found at index {result}')
else:
    print('Target not found')
```

```
A = [3, 7, 12, 18, 23, 27, 30, 35]
target = 12
result = recursive_binary_search(A, target, 0, len(A)-1)
if result != -1:
    print(f'Target found at index {result}')
else:
    print('Target not found')
```

Converting Iterative Binary Search to Recursive

```
def binary_search(A, target, low, high):
    while low <= high:
        mid = (low + high) // 2

        if A[mid] == target:
            return mid
        elif A[mid] > target:
            high = mid - 1
        else:
            low = mid + 1

    return -1
```

```
def recursive_binary_search(A, target, low, high):
    if low > high:
        return -1
```

```
A = [3, 7, 12, 18, 23, 27, 30, 35]
target = 12
result = binary_search(A, target, 0, len(A)-1)
if result != -1:
    print(f'Target found at index {result}')
else:
    print('Target not found')
```

```
A = [3, 7, 12, 18, 23, 27, 30, 35]
target = 12
result = recursive_binary_search(A, target, 0, len(A)-1)
if result != -1:
    print(f'Target found at index {result}')
else:
    print('Target not found')
```

Converting Iterative Binary Search to Recursive

```python
def binary_search(A, target, low, high):
    while low <= high:
        mid = (low + high) // 2

        if A[mid] == target:
            return mid
        elif A[mid] > target:
            high = mid - 1
        else:
            low = mid + 1

    return -1
```

```python
def recursive_binary_search(A, target, low, high):
    if low > high:
        return -1
```

```python
A = [3, 7, 12, 18, 23, 27, 30, 35]
target = 12
result = binary_search(A, target, 0, len(A)-1)
if result != -1:
    print(f'Target found at index {result}')
else:
    print('Target not found')
```

```python
A = [3, 7, 12, 18, 23, 27, 30, 35]
target = 12
result = recursive_binary_search(A, target, 0, len(A)-1)
if result != -1:
    print(f'Target found at index {result}')
else:
    print('Target not found')
```

Converting Iterative Binary Search to Recursive

```
def binary_search(A, target, low, high):
    while low <= high:
        mid = (low + high) // 2
        if A[mid] == target:
            return mid
        elif A[mid] > target:
            high = mid - 1
        else:
            low = mid + 1
    return -1
```

```
def recursive_binary_search(A, target, low, high):
    if low > high:
        return -1
    mid = (low + high) // 2
```

```
A = [3, 7, 12, 18, 23, 27, 30, 35]
target = 12
result = binary_search(A, target, 0, len(A)-1)
if result != -1:
    print(f'Target found at index {result}')
else:
    print('Target not found')
```

```
A = [3, 7, 12, 18, 23, 27, 30, 35]
target = 12
result = recursive_binary_search(A, target, 0, len(A)-1)
if result != -1:
    print(f'Target found at index {result}')
else:
    print('Target not found')
```

Converting Iterative Binary Search to Recursive

```
def binary_search(A, target, low, high):
    while low <= high:
        mid = (low + high) // 2
        if A[mid] == target:
            return mid
        elif A[mid] > target:
            high = mid - 1
        else:
            low = mid + 1
    return -1
```

```
def recursive_binary_search(A, target, low, high):
    if low > high:
        return -1
    mid = (low + high) // 2
    if A[mid] == target:
        return mid
```

```
A = [3, 7, 12, 18, 23, 27, 30, 35]
target = 12
result = binary_search(A, target, 0, len(A)-1)
if result != -1:
    print(f'Target found at index {result}')
else:
    print('Target not found')
```

```
A = [3, 7, 12, 18, 23, 27, 30, 35]
target = 12
result = recursive_binary_search(A, target, 0, len(A)-1)
if result != -1:
    print(f'Target found at index {result}')
else:
    print('Target not found')
```

Converting Iterative Binary Search to Recursive

```
def binary_search(A, target, low, high):
    while low <= high:
        mid = (low + high) // 2
        if A[mid] == target:
            return mid
        elif A[mid] > target:
            high = mid - 1
        else:
            low = mid + 1
    return -1
```

```
def recursive_binary_search(A, target, low, high):
    if low > high:
        return -1
```

Base Case 1

```
    mid = (low + high) // 2
    if A[mid] == target:
        return mid
```

Base Case 2

```
A = [3, 7, 12, 18, 23, 27, 30, 35]
target = 12
result = binary_search(A, target, 0, len(A)-1)
if result != -1:
    print(f'Target found at index {result}')
else:
    print('Target not found')
```

```
A = [3, 7, 12, 18, 23, 27, 30, 35]
target = 12
result = recursive_binary_search(A, target, 0, len(A)-1)
if result != -1:
    print(f'Target found at index {result}')
else:
    print('Target not found')
```

Converting Iterative Binary Search to Recursive

```
def binary_search(A, target, low, high):

    while low <= high:
        mid = (low + high) // 2

        if A[mid] == target:
            return mid
        elif A[mid] > target:
            high = mid - 1
        else:
            low = mid + 1

    return -1
```

```
def recursive_binary_search(A, target, low, high):

    if low > high:
        return -1

    mid = (low + high) // 2

    if A[mid] == target:
        return mid
    elif A[mid] > target:
        return recursive_binary_search(A, target, ___, ___)
```

```
A = [3, 7, 12, 18, 23, 27, 30, 35]
target = 12
result = binary_search(A, target, 0, len(A)-1)
if result != -1:
    print(f'Target found at index {result}')
else:
    print('Target not found')
```

```
A = [3, 7, 12, 18, 23, 27, 30, 35]
target = 12
result = recursive_binary_search(A, target, 0, len(A)-1)
if result != -1:
    print(f'Target found at index {result}')
else:
    print('Target not found')
```

Converting Iterative Binary Search to Recursive

```python
def binary_search(A, target, low, high):

    while low <= high:
        mid = (low + high) // 2

        if A[mid] == target:
            return mid
        elif A[mid] > target:
            high = mid - 1
        else:
            low = mid + 1

    return -1
```

```python
def recursive_binary_search(A, target, low, high):

    if low > high:
        return -1

    mid = (low + high) // 2

    if A[mid] == target:
        return mid
    elif A[mid] > target:
        return recursive_binary_search(A, target, ___, ___)
```

```python
A = [3, 7, 12, 18, 23, 27, 30, 35]
target = 12
result = binary_search(A, target, 0, len(A)-1)
if result != -1:
    print(f'Target found at index {result}')
else:
    print('Target not found')
```

```python
A = [3, 7, 12, 18, 23, 27, 30, 35]
target = 12
result = recursive_binary_search(A, target, 0, len(A)-1)
if result != -1:
    print(f'Target found at index {result}')
else:
    print('Target not found')
```

Converting Iterative Binary Search to Recursive

```
def binary_search(A, target, low, high):
    while low <= high:
        mid = (low + high) // 2

        if A[mid] == target:
            return mid
        elif A[mid] > target:
            high = mid - 1
        else:
            low = mid + 1

    return -1
```

```
def recursive_binary_search(A, target, low, high):
    if low > high:
        return -1

    mid = (low + high) // 2

    if A[mid] == target:
        return mid
    elif A[mid] > target:
        return recursive_binary_search(A, target, ___, ___)
```

```
A = [3, 7, 12, 18, 23, 27, 30, 35]
target = 12
result = binary_search(A, target, 0, len(A)-1)
if result != -1:
    print(f'Target found at index {result}')
else:
    print('Target not found')
```

```
A = [3, 7, 12, 18, 23, 27, 30, 35]
target = 12
result = recursive_binary_search(A, target, 0, len(A)-1)
if result != -1:
    print(f'Target found at index {result}')
else:
    print('Target not found')
```

Converting Iterative Binary Search to Recursive

```
def binary_search(A, target, low, high):
    while low <= high:
        mid = (low + high) // 2

        if A[mid] == target:
            return mid
        elif A[mid] > target:
            high = mid - 1
        else:
            low = mid + 1

    return -1
```

#A[mid] < target:
(mistake in video)

```
def recursive_binary_search(A, target, low, high):
    if low > high:
        return -1

    mid = (low + high) // 2

    if A[mid] == target:
        return mid
    elif A[mid] > target:
        return recursive_binary_search(A, target, ___, ___)
    else:
        return recursive_binary_search(A, target, ___, ___)
```

```
A = [3, 7, 12, 18, 23, 27, 30, 35]
target = 12
result = binary_search(A, target, 0, len(A)-1)
if result != -1:
    print(f'Target found at index {result}')
else:
    print('Target not found')
```

```
A = [3, 7, 12, 18, 23, 27, 30, 35]
target = 12
result = recursive_binary_search(A, target, 0, len(A)-1)
if result != -1:
    print(f'Target found at index {result}')
else:
    print('Target not found')
```

Converting Iterative Binary Search to Recursive

```
def binary_search(A, target, low, high):
    while low <= high:
        mid = (low + high) // 2
        if A[mid] == target:
            return mid
        elif A[mid] > target:
            high = mid - 1
        else:
            low = mid + 1
    return -1
```

```
def recursive_binary_search(A, target, low, high):
    if low > high:
        return -1
```

Base Case 1

```
    mid = (low + high) // 2
    if A[mid] == target:
        return mid
```

Base Case 2

```
    elif A[mid] > target:
        return recursive_binary_search(A, target, ___, ___)
    else:
        return recursive_binary_search(A, target, ___, ___)
```

```
A = [3, 7, 12, 18, 23, 27, 30, 35]
target = 12
result = binary_search(A, target, 0, len(A)-1)
if result != -1:
    print(f'Target found at index {result}')
else:
    print('Target not found')
```

```
A = [3, 7, 12, 18, 23, 27, 30, 35]
target = 12
result = recursive_binary_search(A, target, 0, len(A)-1)
if result != -1:
    print(f'Target found at index {result}')
else:
    print('Target not found')
```

Converting Iterative Binary Search to Recursive

```
def binary_search(A, target, low, high):
    while low <= high:
        mid = (low + high) // 2

        if A[mid] == target:
            return mid
        elif A[mid] > target:
            high = mid - 1
        else:
            low = mid + 1

    return -1
```

```
def recursive_binary_search(A, target, low, high):
    if low > high:
        return -1

    mid = (low + high) // 2

    if A[mid] == target:
        return mid
    elif A[mid] > target:
        return recursive_binary_search(A, target, ___, ___)
    else:
        return recursive_binary_search(A, target, ___, ___)
```

Base Case 1

Base Case 2

Recursive Cases

```
A = [3, 7, 12, 18, 23, 27, 30, 35]
target = 12
result = binary_search(A, target, 0, len(A)-1)
if result != -1:
    print(f'Target found at index {result}')
else:
    print('Target not found')
```

```
A = [3, 7, 12, 18, 23, 27, 30, 35]
target = 12
result = recursive_binary_search(A, target, 0, len(A)-1)
if result != -1:
    print(f'Target found at index {result}')
else:
    print('Target not found')
```

Summary and Next Steps

- What We Covered:
  - Reviewed best, worst, and average cases for Linear Search.

Summary and Next Steps

- What We Covered:
  - Reviewed best, worst, and average cases for Linear Search.
  - Introduced and analyzed Binary Search, emphasizing its efficiency.

Summary and Next Steps

- What We Covered:
  - Reviewed best, worst, and average cases for Linear Search.
  - Introduced and analyzed Binary Search, emphasizing its efficiency.
  - Discussed Binary Search's time complexity and why it's faster than Linear Search.

Summary and Next Steps

- What We Covered:
  - Reviewed best, worst, and average cases for Linear Search.
  - Introduced and analyzed Binary Search, emphasizing its efficiency.
  - Discussed Binary Search’s time complexity and why it’s faster than Linear Search.
  - Defined the Binary Search algorithm and implemented it iteratively.

Summary and Next Steps

- What We Covered:
  - Reviewed best, worst, and average cases for Linear Search.
  - Introduced and analyzed Binary Search, emphasizing its efficiency.
  - Discussed Binary Search’s time complexity and why it’s faster than Linear Search.
  - Defined the Binary Search algorithm and implemented it iteratively.
  - Introduced recursion as an alternative design approach.

Summary and Next Steps

- What We Covered:
  - Reviewed best, worst, and average cases for Linear Search.
  - Introduced and analyzed Binary Search, emphasizing its efficiency.
  - Discussed Binary Search’s time complexity and why it’s faster than Linear Search.
  - Defined the Binary Search algorithm and implemented it iteratively.
  - Introduced recursion as an alternative design approach.
  - Provided a detailed outline of the recursive Binary Search algorithm.

Summary and Next Steps

- What We Covered:
  - Reviewed best, worst, and average cases for Linear Search.
  - Introduced and analyzed Binary Search, emphasizing its efficiency.
  - Discussed Binary Search's time complexity and why it's faster than Linear Search.
  - Defined the Binary Search algorithm and implemented it iteratively.
  - Introduced recursion as an alternative design approach.
  - Provided a detailed outline of the recursive Binary Search algorithm.
- Exercise:
  - Action Required: Visit our GitHub page (link in the description) to complete the exercise.

Summary and Next Steps

- What We Covered:
  - Reviewed best, worst, and average cases for Linear Search.
  - Introduced and analyzed Binary Search, emphasizing its efficiency.
  - Discussed Binary Search's time complexity and why it's faster than Linear Search.
  - Defined the Binary Search algorithm and implemented it iteratively.
  - Introduced recursion as an alternative design approach.
  - Provided a detailed outline of the recursive Binary Search algorithm.
- Exercise:
  - Action Required: Visit our GitHub page (link in the description) to complete the exercise.

Coming Up Next:

Topic: Sorting Algorithms

Summary and Next Steps

- What We Covered:
  - Reviewed best, worst, and average cases for Linear Search.
  - Introduced and analyzed Binary Search, emphasizing its efficiency.
  - Discussed Binary Search's time complexity and why it's faster than Linear Search.
  - Defined the Binary Search algorithm and implemented it iteratively.
  - Introduced recursion as an alternative design approach.
  - Provided a detailed outline of the recursive Binary Search algorithm.
- Exercise:
  - Action Required: Visit our GitHub page (link in the description) to complete the exercise.

Coming Up Next:

Topic: Sorting Algorithms

Focus: Exploring foundational sorting techniques and their importance in efficient data organization

Thank You for Watching!