

Binary search student handout

B2.4 search algorithms • Next lesson after linear search • One or two 55 minute classes

Name _____ Date _____

Binary search uses sorted data to rule out about half of the remaining candidates at each step. By the end, you should be able to explain when it works, trace it accurately, and implement and test it in Python.

Recap of linear search

Linear search checks items one at a time, usually from the start, until it finds the target or reaches the end. It works on unsorted data. Our search functions return a matching index, or -1 if the target is absent.

For $[18, 5, 27, 9]$, searching for 27 checks 18, then 5, then 27 and returns index 2. An unsuccessful search checks all four items. Remember that an index identifies a position; it is not the value stored there.

When binary search works

The list must already be sorted by the same key you are searching. This handout assumes ascending order and comparable values. Equal values are allowed. Python lists let us access the middle item directly by index.

Sorting first takes extra work and may change positions. For one search in a short unsorted list, linear search may be the better choice. Repeated searches of a large, already sorted list make binary search particularly useful.

How the algorithm works

1. Set low to 0 and high to the last index. Both bounds are included in the search range.
2. While $\text{low} \leq \text{high}$, calculate $\text{mid} = (\text{low} + \text{high}) // 2$. Integer division chooses the lower middle index when there are two middle positions.
3. If the middle value equals the target, return mid.
4. If the middle value is smaller than the target, discard it and everything to its left: set $\text{low} = \text{mid} + 1$.
5. Otherwise, discard it and everything to its right: set $\text{high} = \text{mid} - 1$.
6. Repeat with the remaining range. If $\text{low} > \text{high}$, no candidates remain; return -1 .

Why is discarding safe? In ascending order, every value to the left of the middle is no larger than it, and every value to the right is no smaller.

Worked example and pseudocode

Use this same list for the worked example and Problems 2 and 3. All indices are zero based.

Index	0	1	2	3	4	5	6	7	8	9	10
Value	3	8	12	17	23	31	38	44	56	62	75

Find the value 56

Record low and high at the start of each iteration. The final column shows the decision and bounds for the next iteration.

Step	low	high	mid	A[mid]	Decision and next bounds
1	0	10	5	31	Go right; low = 6, high = 10
2	6	10	8	56	Found; return 8

The search returns index 8 after two middle-item checks. We never needed to inspect all the earlier values.

Pseudocode

This is classroom pseudocode using zero-based indices, inclusive bounds and DIV for integer division. LENGTH gives the number of items.

```
FUNCTION binarySearch(A, target)
  low ← 0
  high ← LENGTH(A) - 1
  WHILE low <= high
    mid ← (low + high) DIV 2
    IF A[mid] = target THEN
      RETURN mid
    ELSE IF A[mid] < target THEN
      low ← mid + 1
    ELSE
      high ← mid - 1
    END IF
  END WHILE
  RETURN -1
END FUNCTION
```

The range changes; the list itself does not. Each unsuccessful middle-item check removes at least that middle item. This is why the loop eventually stops.

Explain to your partner

Why must the middle item be excluded from the next range after it fails to match? Point to the two lines that do this.

Practice with applicability and traces

Problem 1 Choose an appropriate search

For each case, state whether the ascending-order algorithm can be used directly and justify your answer. If it cannot, suggest a suitable action.

a. Find 14 in [2, 6, 14, 20, 29].

b. Find 14 in [20, 2, 29, 14, 6].

c. Find a student ID in records sorted by surname, not by ID.

d. Find 14 in [29, 20, 14, 6, 2].

Problem 2 Trace a successful search

Using the list on the previous page, search for 23. Show every middle-item check. State the returned index and total number of checks. Leave unused rows blank.

Step	low	high	mid	A[mid]	Decision and next bounds

Returned index _____ Number of checks _____

Problem 3 Trace an unsuccessful search

Using the same list, search for 24. After the last check, state the final low and high, explain why the loop stops, and give the return value.

Step	low	high	mid	A[mid]	Decision and next bounds

Python implementation and testing

Problem 4 Complete the implementation

Fill each blank using the pseudocode. Run your completed function in Python. It assumes values is already sorted in ascending order; it does not sort or validate the input.

```
def binary_search(values, target):
    low = 0
    high = _____ # A
    while _____: # B
        mid = _____ # C
        if values[mid] == target:
            return _____ # D
        elif values[mid] < target:
            low = _____ # E
        else:
            high = _____ # F
    return _____ # G
```

Work as a pair: one student types and explains each update; the other predicts the result and checks the bounds. Swap roles halfway through the tests.

Predict each result before running it. Record the actual result. Add one new test of your own.

values	target	Predicted return	Actual return
[3, 8, 12, 17, 23]	3		
[3, 8, 12, 17, 23]	23		
[3, 8, 12, 17, 23]	10		
[]	7		
[7]	7		
[7]	9		
[2, 4, 4, 4, 9]	4		
Your test			

For the duplicate-value test, does this version guarantee the first matching index? Explain from the code.

Using the result safely: -1 means not found in this function. Do not use it as a list index without checking it; in Python, values[-1] refers to the last item.

Debugging and choosing an algorithm

Problem 5 Debug the loop

This version contains three faulty lines. Trace it on paper before running it, because it may loop forever.

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

a. What does it return for values = [7], target = 7? Why is that wrong?

b. For values = [3, 8], target = 8, record the first two sets of low, high and mid. Explain the failure.

c. Correct all three faulty lines. Explain how each change supports a correct search.

Linear search compared with binary search

Count one inspected list item as one check. This is not a count of individual Python comparison operators.

Feature	Linear search	Binary search
Input order	Any order	Sorted by the searched key
Best case on nonempty input	1 check; first item matches	1 check; first middle item matches
Worst case intuition	May inspect every item	Repeatedly halves the candidate range
Example with 1024 items	Up to 1024 checks	Up to 11 middle-item checks
Preparation	No sorting required	Sorting or maintaining order costs work

An empty list needs zero item checks. Binary search is not always faster for every target: linear search finds the first item immediately.

Extension and exit check

Problem 6 Find the first occurrence

The basic binary search may return any matching index. Modify it so it returns the first occurrence when duplicates exist, or -1 if there is no match.

Start with $\text{result} = -1$. When you find a match, save its index and keep searching to the left. Return result when the loop finishes. Complete this task in your editor and outline your changes here.

Test $[2, 4, 4, 4, 9]$ with target 4; $[4, 4, 4]$ with target 4; and $[2, 3, 5]$ with target 4. Predict the three return values, then verify them.

Exit check

1. A list has 1024 items and is already sorted. State the maximum item checks for linear and binary search using this handout's implementations.

2. You will search a short unsorted list only once. Which search would you choose, and why?

3. In one sentence, explain why $\text{low} = \text{mid} + 1$ is safe when $\text{values}[\text{mid}] < \text{target}$.

Success check: I can justify the sorted-data precondition, trace both outcomes, distinguish an index from a value, and test boundary cases.