ANSWERS

A5.3 Linear and binary search

Algorithms and complexity · A level · OCR H446 2.3.1, AQA 7517 4.3.4.1, Eduqas A500QS 1.3 · about 25 min

BugBotLab

What this lesson is about

Tracing both searches in pseudocode, recursive binary search, O(n) against O(log n), and when sorting first pays off.

- Linear search
- Binary search
- Binary search by recursion
- Is sorting first worth it?

Questions

6 marks in all

1. What is the worst-case time complexity of binary search? [1 mark]

- ☐ **A** O(log n)
- ☐ **B** O(n)
- ☐ **C** O(1)
- ☐ **D** O(n log n)

> **Answer:** A. Each look halves what is left, so about $\log_2 n$ looks are needed.

2. Binary search looks at the middle item and halves what is left each time. What is the largest number of items it must look at to search a sorted list of 1,000,000 items? [1 mark]

> **Answer:** 20. $\lfloor \log_2 1{,}000{,}000 \rfloor + 1 = 19 + 1 = 20$.

3. Which statement about linear search is true? [1 mark]

- ☐ **A** It works on unsorted data and is O(n)
- ☐ **B** It needs sorted data and is O(n)
- ☐ **C** It works on unsorted data and is O(log n)
- ☐ **D** It is O(1) in the worst case

> **Answer:** A. Linear search checks each item in turn, so order does not matter, and the worst case checks all n.

4. This binary search prints each middle index it looks at. What does it print? [1 mark]

```
items = [3, 9, 14, 21, 30, 38, 45, 52, 60, 71, 88]
low, high = 0, len(items) - 1
target = 60
while low <= high:
    mid = (low + high) // 2
    print(mid)
    if items[mid] == target:
        break
    elif target < items[mid]:
        high = mid - 1
    else:
        low = mid + 1
```

Answer:

```
5
8
```

It looks at index 5 (38), so low becomes 6; then index 8, which holds 60.

5. Recursive binary search has the same time complexity as the iterative version. What is its space complexity? [1 mark]

- ☐ **A** O(log n), because of the stack frames of the waiting calls
- ☐ **B** O(1), like the iterative version
- ☐ **C** O(n), because it copies the list
- ☐ **D** O(n log n)

Answer: A. Each call waits for the one it made, so up to about $\log_2 n$ frames are on the stack at once.

6. A list of 1,000,000 unsorted readings will be searched just once. Which is the better choice? [1 mark]

- ☐ **A** Linear search, because sorting first costs more than the search saves
- ☐ **B** Sort it and then use binary search, because binary search is always faster
- ☐ **C** Binary search on the unsorted list
- ☐ **D** Neither can search a list that large

Answer: A. Sorting is O(n log n), far more work than one O(n) linear search. Sorting only pays off when searching many times.

The task: recursive binary search

Write `binary_search(items, target, low, high)` **recursively** (it must call itself, with no `while` or `for` loop inside it). It searches the sorted list `items` between indexes `low` and `high` inclusive, finds the middle with `(low + high) // 2`, and **returns a tuple** `(index, looks)`: the index of `target`, or -1 if it is not there, and the number of middle items it looked at. `items` is the 1,000 even numbers 0, 2, 4, ... 1998. Print exactly three lines: - `734: index <index> after <looks> looks` - `735: not found after <looks> looks` - `most looks: <n>`, the largest number of looks needed to find any one of the 1,000 items in the list (search for every item to find out).

```
# the two lines every program starts with: the commands, then the robot
from bugbot import *
connect()

items = list(range(0, 2000, 2))

def binary_search(items, target, low, high):
    return -1, 0
```

The hint students can ask for: Two base cases come first: low has passed high (not there), or the middle item is the target. Otherwise call the function again on the half that could hold the target, and add one look to whatever that call reports. To find the most looks, search for each item in turn and keep the largest count.

A solution

```
from bugbot import *
connect()

items = list(range(0, 2000, 2))

def binary_search(items, target, low, high):
    if low > high:
        return -1, 0
    mid = (low + high) // 2
    if items[mid] == target:
        return mid, 1
    if target < items[mid]:
        index, looks = binary_search(items, target, low, mid - 1)
    else:
        index, looks = binary_search(items, target, mid + 1, high)
    return index, looks + 1

index, looks = binary_search(items, 734, 0, len(items) - 1)
print("734: index", index, "after", looks, "looks")
index, looks = binary_search(items, 735, 0, len(items) - 1)
print("735: not found after", looks, "looks")
most = 0
for x in items:
    most = max(most, binary_search(items, x, 0, len(items) - 1)[1])
print("most looks:", most)
```

Any program that meets the task's checks is marked correct in the simulator; this is one way, not the only way.