1. Linear Search

1. Logic (intuition)

- Goal: find value `x` in an array `A[0..n-1]`.
- Idea: start at the beginning and check every element one by one until you either find `x` or finish the array.

---

2. Steps

For array `A[0..n-1]` and target `x`:

1. Set index `i = 0`.
2. While `i < n`:
    - If `A[i] == x`, return index `i`.
    - Else increment `i` and continue.
3. If loop finishes without finding `x`, return "not found" (e.g., −1).

A. Iterative linear search

3. C++-style code (iterative)

cpp

```cpp
int linearSearchIter(int A[], int n, int x) {
    for (int i = 0; i < n; i++) {      // loop from 0 to n-1
        if (A[i] == x) {               // compare element with x
            return i;                  // found at index i
        }
    }
    return -1;                         // not found
}
```

---

4. Time complexity – equation → Big-O → words

Let

$\mathrm{Titer(n)}$

T

iter

(n) be the number of basic operations in worst case (x at end or not present).

Per iteration:

- `i < n` check → 1
- `A[i] == x` comparison → 1
- `i++` increment (except last) → ~1

Total per iteration ≈ constant c.

Loop executes for `i = 0..n-1` → n iterations.

So:

$$\mathrm{T(n)=c\cdot n+k}$$

$$(n)=c\cdot n+k$$

for some constants

$\mathrm{c,k}$

$c,k.$

Asymptotically:

- $\mathrm{T(n)\in O(n)}$
- T
- iter
- $(n)\in O(n).$
- In words: linear time (average and worst case).

Best case (x at first position):

- 1 comparison →
- $\mathrm{Tbest(n)=O(1)}$
- T
- best
- $(n)=O(1)$ → constant time.

---

5. Space complexity – equation → words

Extra space beyond the array:

- Local variables: `i` and function parameters (`n`, `x`, pointer to `A`).
- Count of extra integer variables is constant.

Let

$\mathrm{S(n)}$

S

(n) be extra space:

$$\mathrm{S(n)=c}$$

$$(n)=c$$

for some constant

c

c.

So:

- $\mathrm{S(n)\in O(1)}$
- $(n)\in O(1)$.
- In words: constant extra space, algorithm is in-place.

B. Recursive linear search

This is not commonly used in practice but is logically possible and can be examined.

3. C++-style code (recursive)

Version that searches from index `i` onwards:

cpp

```cpp
int linearSearchRec(int A[], int n, int x, int i) {
    if (i >= n) {                // base case: reached end, not found
        return -1;
    }
    if (A[i] == x) {             // found at index i
        return i;
    }
    // recursive call on the remaining part of the array
    return linearSearchRec(A, n, x, i + 1);
}
```

Typical call:

cpp

```cpp
int index = linearSearchRec(A, n, x, 0);
```

4r. Time complexity – recurrence → Big-O → words

Let

$$T_{rec}(n)$$

(n) be time to search in an array segment of length

n

n.

In each call:

- Check base case `i >= n` → constant.
- Compare `A[i] == x` → constant.
- One recursive call on index `i+1`.

At each step, the remaining segment shrinks by 1:

- First call: size
- n
- n
- Next:
- n−1
- $n-1$, then
- n−2
- $n-2$, … until 0.

So recurrence:

$$\text{Trec}(n)=\text{Trec}(n-1)+c$$

T

rec

$(n)=T$

rec

$$(n-1)+c$$

with base case

$\text{Trec}(0)=O(1)$

T

rec

$(0)=O(1)$.

Solving by expansion:

$$\mathrm{Trec(n)=c+c+\cdots+c+Trec(0)=c\cdot n+k}$$

T

rec

$(n)=c+c+\cdots+c+T$

rec

$$(0)=c\cdot n+k$$

So:

- $\mathrm{Trec(n)\in O(n)}$
- T
- rec
- $(n)\in O(n)$.
- In words: linear time (same as iterative).

Best case (found at first index):

- Only one call $\rightarrow$
- $\mathrm{O(1)}$
- $O(1)\rightarrow$ constant time.

---

5. Space complexity – recursion stack

Each call adds one frame on the call stack.

Per call space:

- Local variables: parameters `A`, `n`, `x`, `i`, plus return address → constant.

Maximum depth:

- In worst case, you go from `i = 0` to `i = n`, so n recursive calls on the stack.

Let

$\mathrm{Srec(n)}$

S

rec

(n) be extra space:

- Each level: constant
- c
- c.
- Depth:
- n
- n.

$$\mathrm{Srec(n)} \approx \mathrm{c} \cdot \mathrm{n}$$

S

rec

$$(n) \approx c \cdot n$$

So:

- $\mathrm{Srec(n)} \in \mathrm{O(n)}$
- S
- rec
- $(n) \in O(n)$.
- In words: linear extra space due to recursion stack.

This is worse than the iterative version's constant space.

---

6. Exam angle (linear search, both forms)

Possible questions in your pattern:

- MCQ about time complexity:
    - "The complexity of linear search algorithm is:"
        - Correct:
        - $\mathrm{O(n)}$

        - $O(n) \rightarrow$ linear time (average and worst case).
- MCQ about average case:
    - “Average case of linear search is:”
        - When item is somewhere in the middle of the array; still linear overall.
- MCQ comparing with binary search:
    - “A characteristic that binary search uses but linear search ignores is:”
        - Correct: the order (sortedness) of the list.
- MCQ about space / recursion:
    - “Space complexity of iterative linear search (excluding input) is:” $\rightarrow$
    - $\mathrm{O}(1)$
    - $O(1)$, constant.
    - “Space complexity of recursive linear search (excluding input) is:” $\rightarrow$
    - $\mathrm{O}(\mathrm{n})$
    - $O(n)$ due to recursion depth.
- Code-based MCQ:
    - They can show either iterative or recursive version and ask:
        - “What does this function do?”
        - “What is its worst-case time complexity?”
        - “Will this function run out of stack for very large arrays?” (recursive only).

2.Binary search

1. Logic (intuition)

- Array must be sorted in non-decreasing order.
- Instead of scanning all elements, binary search looks at the middle, compares with `x`, and then throws away half of the search range each time.
- This “halving” makes it much faster than linear search for large

2. Steps

Given sorted array `A[0..n-1]` and target `x`:

1. Set `low = 0`, `high = n - 1`.
2. While `low <= high`:

- Compute `mid = low + (high - low)/2`.
- If `A[mid] == x`, return `mid`.
- If `x < A[mid]`, set `high = mid - 1` (go left half).
- Else set `low = mid + 1` (go right half).

3. If loop ends without finding `x`, return −1 (not found).

A. Iterative binary search

3. C++-style code (iterative)

```
int binarySearchIter(int A[], int n, int x) {

   int low = 0;

   int high = n - 1;



   while (low <= high) {

      int mid = low + (high - low) / 2;   // avoid overflow



      if (A[mid] == x) {

         return mid;                  // found

      } else if (x < A[mid]) {

         high = mid - 1;              // search left half

      } else {

         low = mid + 1;               // search right half
```

```
        }

    }


    return -1;                  // not found

}
```

4. Time complexity – equation → Big-O → words

Let

$\mathrm{Titer(n)}$

T

iter

(n) be number of basic operations in worst case.

Per loop iteration:

- Compute `mid` → constant.
- Compare `A[mid]` with `x` → constant.
- Possibly update `low` or `high` → constant.

So each iteration ≈ constant

c

c operations.

How many iterations?

- Each iteration halves the search interval size:
- $\mathrm{n,n/2,n/4,...,1}$
- $n,n/2,n/4,...,1.$
- Number of times we can halve
- n

- n until 1 ≈
- $\log 2n$
- $\log$
- 2
- $n.$

So:

$$\mathrm{Titer(n)} \approx \mathrm{c} \cdot \log 2\mathrm{n} + \mathrm{k}$$

T

iter

$(n) \approx c \cdot \log$

2

$$n+k$$

Thus:

- $\mathrm{Titer(n)} \in \mathrm{O(logn)}$
- T
- iter
- $(n) \in O(\log n).$
- In words: logarithmic time (worst and average case).
  Best case: if `A[mid] == x` on first check →
- $\mathrm{O}(1)$
- $O(1)$ constant time.

5. Space complexity – equation → words

Extra space beyond array:

- Local variables: `low`, `high`, `mid`, `n`, `x` → constant count.
- No recursion, no extra arrays.

Let

$\mathrm{Siter(n)}$

S

iter

(n) be extra space:

$$\text{Siter(n)=c}$$

S

iter

$$(n)=c$$

for some constant

c

c.

So:

- $\text{Siter(n)} \in \text{O(1)}$
- S
- iter
- $(n) \in O(1)$.
- In words: constant extra space, algorithm is in-place (for searching).

B. Recursive binary search

3r. C++-style code (recursive)

```
int binarySearchRec(int A[], int low, int high, int x) {
   if (low > high) {                  // base case: not found
      return -1;
   }

   int mid = low + (high - low) / 2;

   if (A[mid] == x) {
      return mid;                     // found
```

```
    } else if (x < A[mid]) {
        return binarySearchRec(A, low, mid - 1, x);     // search left half
    } else {
        return binarySearchRec(A, mid + 1, high, x);    // search right half
    }
}
```

Call:
int index = binarySearchRec(A, 0, n - 1, x);

4r. Time complexity – recurrence → Big-O → words

Let

$\mathrm{Trec(n)}$

T

rec

(n) be time when searching a segment of size

n

n.

In each call:

- Non-recursive work:
  - `low > high` check, compute `mid`, compare `A[mid]` with `x`, choose branch → constant
  - c
  - c.
- Recursive call is on a subarray of size
- n/2
- $n/2$.

Recurrence:

$$\mathrm{Trec(n)=Trec(n/2)+c}$$

T

rec

$(n)=T$

rec

$$(n/2)+c$$

with base case

$\mathrm{Trec}(1)=\mathrm{O}(1)$

T

rec

$(1)=O(1).$

Solve:

- Standard recurrence →
- $\mathrm{Trec}(\mathrm{n})=\mathrm{clog2n+k}$
- T
- rec
- $(n)=c\log$
- 2
- $n+k.$

So:

- $\mathrm{Trec}(\mathrm{n})\in\mathrm{O}(\mathrm{logn})$
- T
- rec
- $(n)\in O(\log n).$
- In words: logarithmic time (same as iterative).

Best case:

- First `mid` already matches `x` → one call →
- $\mathrm{O}(1)$
- $O(1).$

5r. Space complexity – recursion stack

Per call space:

- Parameters (`A`, `low`, `high`, `x`), local `mid`, return address → constant.

Max depth of recursion:

- Each call halves the segment size until 1 → depth ≈
- $\log 2n$
- $\log$
- 2
- $n.$

Let

$\mathrm{Srec}(n)$

S

rec

(n) be extra space:

$$\mathrm{Srec}(n) \approx c \cdot \log 2n$$

S

rec

$(n) \approx c \cdot \log$

2

$$n$$

So:

- $\mathrm{Srec}(n) \in O(\log n)$
- S
- rec
- $(n) \in O(\log n).$
- In words: logarithmic extra space, due to recursion stack.

Compared:

- Iterative binary search: time
- $\mathrm{O}(\log n)$
- $O(\log n)$, space
- $\mathrm{O}(1)$
- $O(1)$.
- Recursive binary search: time
- $\mathrm{O}(\log n)$
- $O(\log n)$, space
- $\mathrm{O}(\log n)$
- $O(\log n)$.

---

6. Exam angle (binary search, both forms)

Based on your pattern + syllabus:

- Conceptual MCQs:
  - "Binary search can be applied only if:" → array is sorted.
  - "What extra property does binary search exploit that linear search ignores?" → order of elements (you have this exact mid MCQ).
- Time complexity MCQs:
  - "Binary search has which time complexity?" →
  - $\mathrm{O}(\log n)$
  - $O(\log n)$ → logarithmic time.
  - Comparing algorithms: "Which of the following is logarithmic?" → binary search.
- Space complexity / recursion MCQs:
  - "Space complexity of iterative binary search (excluding input):" → constant.
  - "Space complexity of recursive binary search (excluding input):" → logarithmic due to recursion depth.
- Code-based MCQs:
  - Show iterative or recursive code and ask:
    - "What does this code do?"
    - "What is its worst-case time complexity?"
    - "What will it return for certain inputs (e.g., x not found, x smaller than all elements)?"

- Comparison with linear search:
    - “Why is binary search faster than linear search for large
    - n
    - n?” → because
    - $\log n$
    - $\log n$ grows much slower than
    - n
    - n (logarithmic vs linear time).

Core comparison: behavior and requirements

- Linear search
    - Checks each element one by one.
    - Does not require the array to be sorted.
    - Works on any sequence: array, list, unsorted data.
- Binary search
    - Each step compares with the middle and halves the search interval.
    - Requires the array (or list) to be sorted in a known order.
    - Only meaningful on random-access structures (arrays) where mid index is easy.

Key property:

- Binary search uses and depends on the order of elements; your mid MCQ explicitly tested this.

Time & space comparison (words)

- Time complexity
    - Linear search:
        - Worst/average:
        - $O(n)$
        - $O(n) \rightarrow$ linear time.
    - Binary search:
        - Worst/average:
        - $O(\log n)$
        - $O(\log n) \rightarrow$ logarithmic time.

- Space complexity (iterative implementations)
    - Both use only a few variables (indices, target, etc.) →
    - $\mathrm{O}(1)$
    - $O(1)$ → constant extra space, in-place.
    - Recursive forms:
        - Linear search recursion:
        - $\mathrm{O}(\mathrm{n})$
        - $O(n)$ space (depth n).
        - Binary search recursion:
        - $\mathrm{O}(\log \mathrm{n})$
        - $O(\log n)$ space (depth log n).

When is each better?

- Use linear search when:
    - Data is unsorted and you don't want to spend time sorting.
    - The array is very small, so simplicity matters more than speed.
    - Data structure is a linked list (binary search is not efficient on linked lists without extra work).
- Use binary search when:
    - Data is sorted (or can be sorted once and searched many times).
    - You will perform many searches on the same dataset; cost of sorting is worth it.
    - You need very fast lookups for large
    - n
    - n: logarithmic time is far better than linear.

Sometimes, if you must search once in an unsorted array, sorting first then using binary search is worse than just doing one linear search (because sorting costs at least

$\mathrm{O}(\mathrm{n}\log \mathrm{n})$

$O(n\log n)$).

Exam angles for "Comparing Linear vs Binary Search"

Typical question styles (matching your mids and syllabus):

- MCQ on property:
    - “A characteristic that binary search uses but linear search ignores is the:”
        - Correct: order of the elements of the list (you have this exact MCQ).
- MCQ on time (words or symbols):
    - “Linear search runs in _____ time and binary search in _____ time.”
        - Answer: linear, logarithmic.
- Conceptual question:
    - “If you have an unsorted array and need just one search, which is more appropriate and why?”
        - Answer: linear search, because sorting then binary search is more expensive.
- Choice question:
    - “For a very large sorted array with many queries, which search is better and why?”

Answer: binary search, because logarithmic time per query is much smaller than linear.