Chapter 2: Algorithm Analysis

Data Structures

Algorithms and efficiency | Experimental analysis | Counting primitive operations | O, Ω, Θ | Loop patterns | Linear and binary search | Space complexity | Correctness and robustness

Sercan KÜLCÜ · Giresun University · Computer Engineering

© Sercan Külcü. All rights reserved. · Interactive HTML version of these slides:
https://sercankulcu.github.io/files/data_structures/slides/Chapter_02_Algorithm_Analysis.html

INTRODUCTION

Learning Outcomes

Measuring and counting

Time an algorithm experimentally and analyze it by counting its primitive operations.

Asymptotic notation

Express time and space complexity with O, Ω, and Θ.

From code to complexity

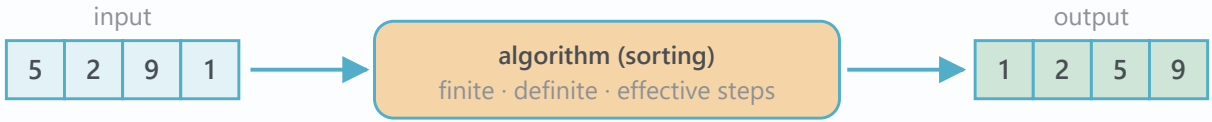
Read best-, average-, and worst-case cost off loop patterns.

Comparing algorithms

Compare algorithms for the same problem by cost, correctness, and robustness.

Algorithms

- An algorithm is a step-by-step procedure for solving a problem: **input** → **steps** → **output**.
- One problem can have many algorithms, and some are more efficient than others.
- Choosing the right one matters: a good algorithm does the job with fewer resources.



Some common algorithms:

Sorting

Puts data in a given order.

`bubble sort, merge sort`

Searching

Finds a given item in a data set.

`linear search, binary search`

Graph

Solves problems on graphs.

`shortest paths, network flow`

What Makes a Good Algorithm

A good algorithm solves its problem effectively. We judge it by four criteria:

Time complexity
Measured by running time or by the number of operations.

Space complexity
Measured by the amount of memory the algorithm uses.

Correctness
It must give the right result for every valid input; wrong answers make it useless.

Robustness
It should cope with a wide range of inputs and situations.

Experimental Analysis: `System.nanoTime()`

```
for (int n = 1000; n <= 64000; n *= 2) {
  int[] a = new int[n];
  for (int i = 0; i < n; i++) a[i] = i; // distinct
  long start = System.nanoTime();
  hasDuplicates(a);                          // timed work
  long elapsed = System.nanoTime() - start;  // ns
  System.out.println(n + " " + elapsed / 1e6 + " ms");
}
```

Measure

Subtracting two `System.nanoTime()` readings gives elapsed ns (not wall-clock time).

Repeat

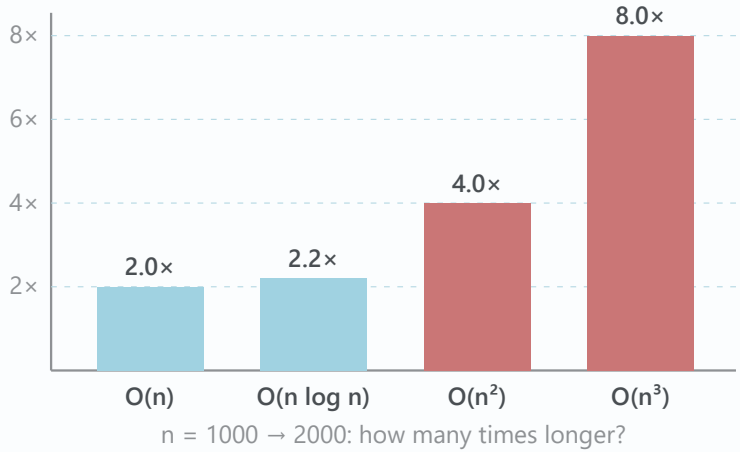
JIT warm-up, garbage collection, and other processes add noise: run several times, take the median.

Plot

Plot time against n and see how the time changes when n **doubles**.

Careful: `new int[n]` is all zeros → `hasDuplicates` (an $O(n^2)$ duplicate check) returns `true` on its first comparison. To time the worst case, use distinct data.

Doubling Experiments and the Limits of Measurement



Limits of experimental analysis (Goodrich §4.1):

- Results depend on the hardware, the JVM, and the load; compare two algorithms **in the same environment**.
- Only a limited set of inputs can be tried; an important one may be missed.
- The algorithm must be **fully implemented** before it can be timed.

Doubling experiment: if $T(2n)/T(n) \approx 2^k$, then $T(n) \approx c \cdot n^k$. To get past these limits, analyze by **counting primitive operations**: hardware-independent, valid for all inputs, no code to run.

Counting Primitive Operations: `arrayMax()`

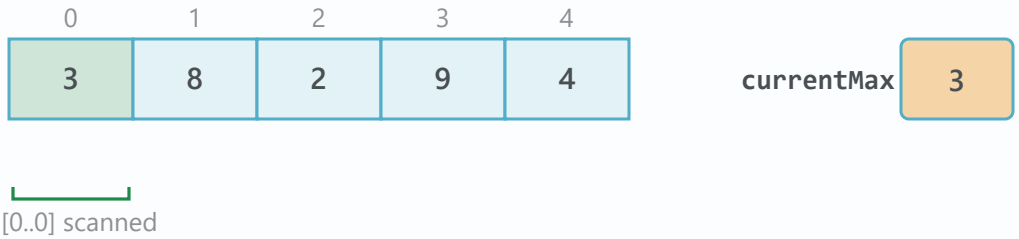
```
int arrayMax(int[] a) {
  int currentMax = a[0];
  for (int i = 1; i < a.length; i++) {
    if (a[i] > currentMax)
      currentMax = a[i];
  }
  return currentMax;
}
```

- **Primitive operation**: assignment, comparison, arithmetic, array indexing, method call / return.
- Each takes constant time → we only count **how many times** each one runs.

Line	Statement	Count
2	`currentMax = a[0]`	1
3	`i = 1`	1
3	`i < a.length`	n
4	`a[i] > currentMax`	n − 1
5	`currentMax = a[i]`	0 … n − 1
3	`i++`	n − 1
7	`return`	1

Best: **3n + 1** (a[0] is the largest, no updates) · Worst: **4n** (increasing array) → **Θ(n)** either way. Constants depend on the hardware, so we drop them.

Code Trace: `arrayMax(a)`



currentMax = a[0] = 3: the first item is the largest so far.

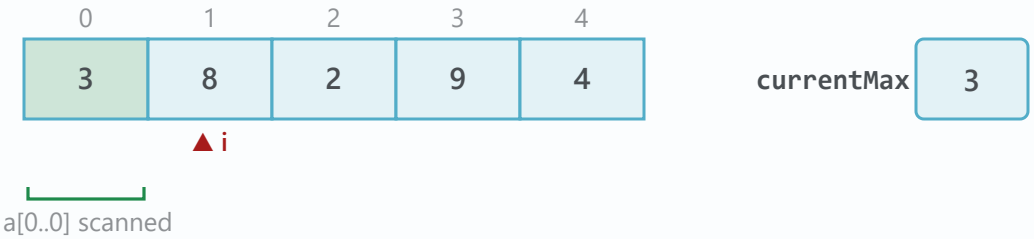
Step 1 / 13

```
1 int arrayMax(int[] a) {
▶   int currentMax = a[0];
3   for (int i = 1; i < a.length; i++) {
4     if (a[i] > currentMax)
5       currentMax = a[i];
6   }
7   return currentMax;
8 }
```

VARIABLES

`a = [3, 8, 2, 9, 4]` `currentMax = 3`

Code Trace: `arrayMax(a)`



i = 1, 1 < 5 → continue. Invariant: currentMax = max(a[0..0]).

Step 2 / 13

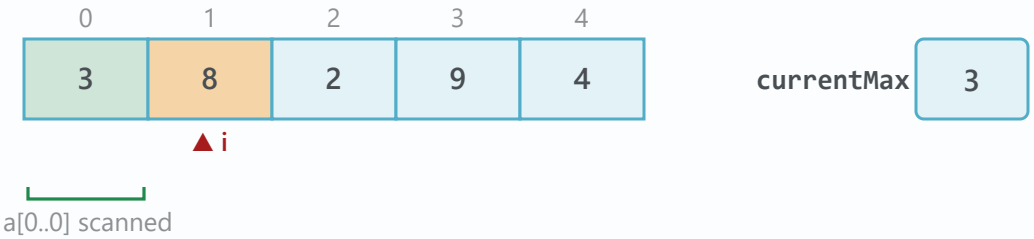
```
1 int arrayMax(int[] a) {
2   int currentMax = a[0];
▶   for (int i = 1; i < a.length; i++) {
4     if (a[i] > currentMax)
5       currentMax = a[i];
6   }
7   return currentMax;
8 }
```

VARIABLES

a = [3, 8, 2, 9, 4] currentMax = 3

i = 1 comparisons = 0 updates = 0

Code Trace: `arrayMax(a)`



a[1] = 8 > 3 → **true**.

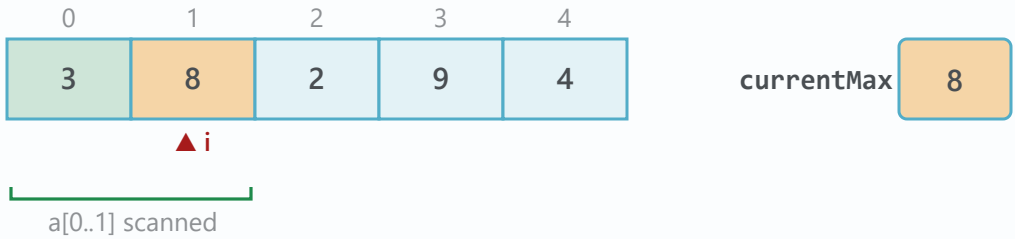
Step 3 / 13

```
1 int arrayMax(int[] a) {
2   int currentMax = a[0];
3   for (int i = 1; i < a.length; i++) {
▶     if (a[i] > currentMax)
5       currentMax = a[i];
6   }
7   return currentMax;
8 }
```

VARIABLES

`a = [3, 8, 2, 9, 4]` `currentMax = 3` `i = 1` `comparisons = 1` `updates = 0`

Code Trace: `arrayMax(a)`



currentMax = 8: a new maximum.

Step 4 / 13

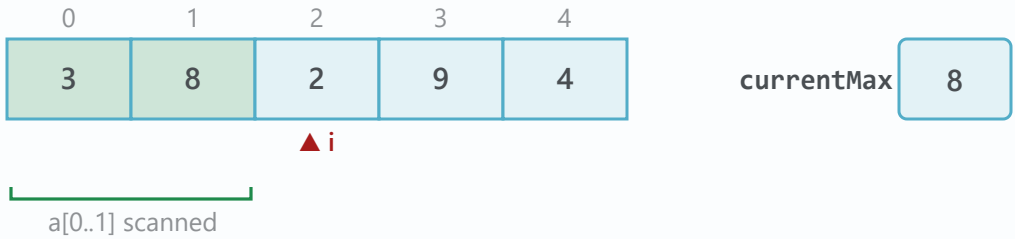
```
1 int arrayMax(int[] a) {
2   int currentMax = a[0];
3   for (int i = 1; i < a.length; i++) {
4     if (a[i] > currentMax)
▶       currentMax = a[i];
6   }
7   return currentMax;
8 }
```

VARIABLES

a = [3, 8, 2, 9, 4] currentMax = 8

i = 1 comparisons = 1 updates = 1

Code Trace: `arrayMax(a)`



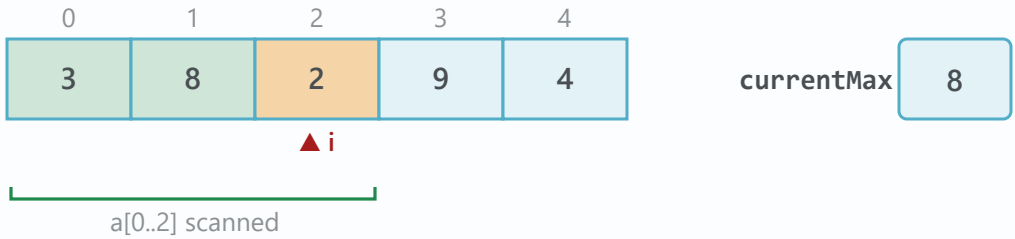
i = 2, 2 < 5 → continue. Invariant: currentMax = max(a[0..1]).


```
1 int arrayMax(int[] a) {
2   int currentMax = a[0];
▶   for (int i = 1; i < a.length; i++) {
4     if (a[i] > currentMax)
5       currentMax = a[i];
6   }
7   return currentMax;
8 }
```

VARIABLES

`a = [3, 8, 2, 9, 4]` `currentMax = 8` `i = 2` `comparisons = 1` `updates = 1`

Code Trace: `arrayMax(a)`



a[2] = 2 > 8 → **false**; currentMax is unchanged.

Step 6 / 13

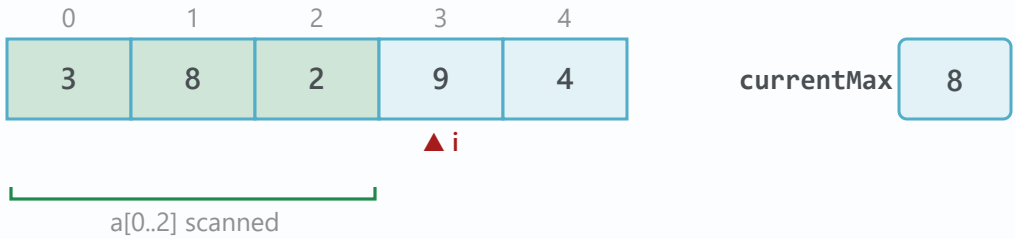
```
1 int arrayMax(int[] a) {
2   int currentMax = a[0];
3   for (int i = 1; i < a.length; i++) {
▶     if (a[i] > currentMax)
5       currentMax = a[i];
6   }
7   return currentMax;
8 }
```

VARIABLES

`a = [3, 8, 2, 9, 4]` `currentMax = 8`

`i = 2` `comparisons = 2` `updates = 1`

Code Trace: `arrayMax(a)`



i = 3, 3 < 5 → continue. Invariant: currentMax = max(a[0..2]).

Step 7 / 13

```
1 int arrayMax(int[] a) {
2   int currentMax = a[0];
▶   for (int i = 1; i < a.length; i++) {
4     if (a[i] > currentMax)
5       currentMax = a[i];
6   }
7   return currentMax;
8 }
```

VARIABLES

a = [3, 8, 2, 9, 4] | currentMax = 8 | i = 3 | comparisons = 2 | updates = 1

Code Trace: `arrayMax(a)`



a[3] = 9 > 8 → **true**.

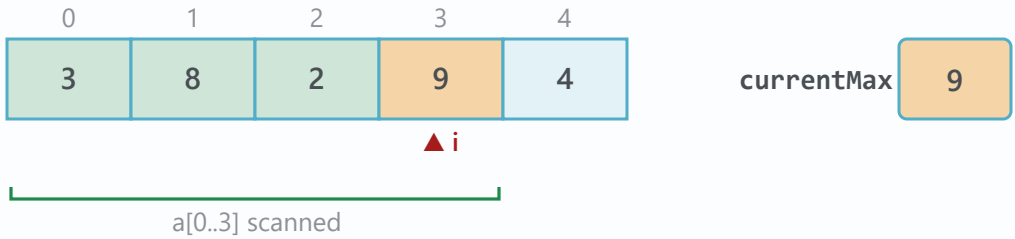
Step 8 / 13

```
1 int arrayMax(int[] a) {
2   int currentMax = a[0];
3   for (int i = 1; i < a.length; i++) {
▶     if (a[i] > currentMax)
5       currentMax = a[i];
6   }
7   return currentMax;
8 }
```

VARIABLES

`a = [3, 8, 2, 9, 4]` `currentMax = 8` `i = 3` `comparisons = 3` `updates = 1`

Code Trace: `arrayMax(a)`



currentMax = 9: a new maximum.

Step 9 / 13

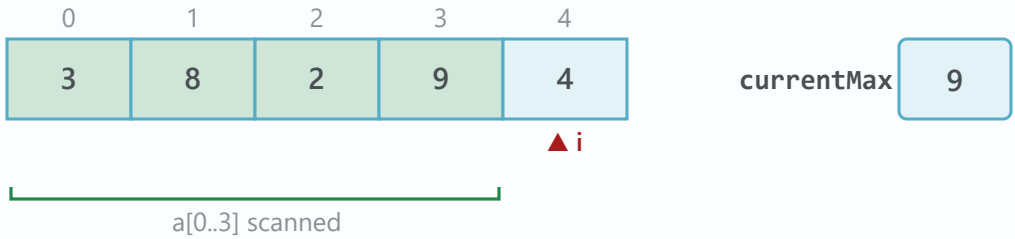
```
1 int arrayMax(int[] a) {
2   int currentMax = a[0];
3   for (int i = 1; i < a.length; i++) {
4     if (a[i] > currentMax)
▶       currentMax = a[i];
6   }
7   return currentMax;
8 }
```

VARIABLES

`a = [3, 8, 2, 9, 4]` `currentMax = 9`

`i = 3` `comparisons = 3` `updates = 2`

Code Trace: `arrayMax(a)`



i = 4, 4 < 5 → continue. Invariant: currentMax = max(a[0..3]).

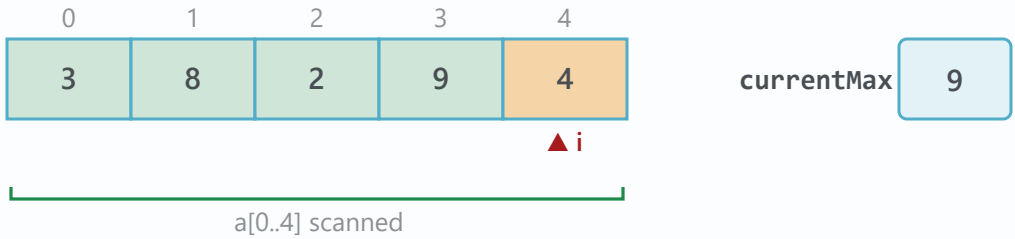
Step 10 / 13

```
1 int arrayMax(int[] a) {
2   int currentMax = a[0];
▶   for (int i = 1; i < a.length; i++) {
4     if (a[i] > currentMax)
5       currentMax = a[i];
6   }
7   return currentMax;
8 }
```

VARIABLES

a = [3, 8, 2, 9, 4] | currentMax = 9 | i = 4 | comparisons = 3 | updates = 2

Code Trace: `arrayMax(a)`



a[4] = 4 > 9 → **false**; currentMax is unchanged.

Step 11 / 13

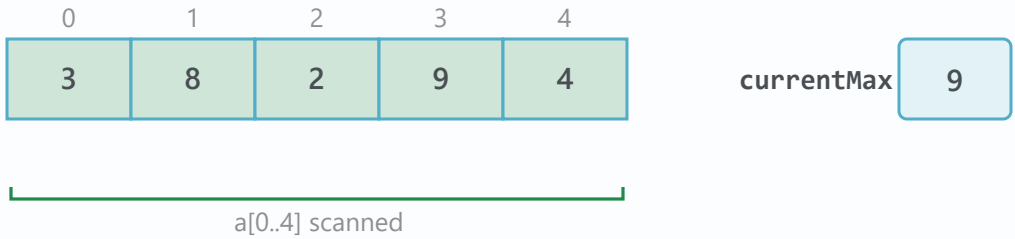
```
1 int arrayMax(int[] a) {
2   int currentMax = a[0];
3   for (int i = 1; i < a.length; i++) {
▶     if (a[i] > currentMax)
5       currentMax = a[i];
6   }
7   return currentMax;
8 }
```

VARIABLES

`a = [3, 8, 2, 9, 4]` `currentMax = 9`

`i = 4` `comparisons = 4` `updates = 2`

Code Trace: `arrayMax(a)`



i = 5: 5 < 5 is false → loop ends. The whole array has been scanned.

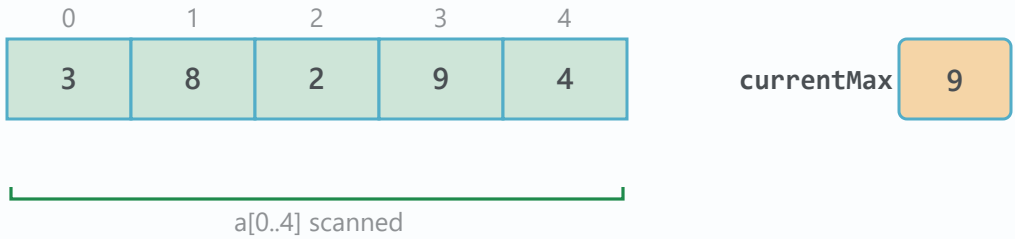
Step 12 / 13

```
1 int arrayMax(int[] a) {
2   int currentMax = a[0];
▶   for (int i = 1; i < a.length; i++) {
4     if (a[i] > currentMax)
5       currentMax = a[i];
6   }
7   return currentMax;
8 }
```

VARIABLES

`a = [3, 8, 2, 9, 4]` `currentMax = 9` `i = 5` `comparisons = 4` `updates = 2`

Code Trace: `arrayMax(a)`



return 9. n − 1 = 4 comparisons for n = 5 → **Θ(n)**.

Step 13 / 13

```
1 int arrayMax(int[] a) {
2   int currentMax = a[0];
3   for (int i = 1; i < a.length; i++) {
4     if (a[i] > currentMax)
5       currentMax = a[i];
6   }
▶   return currentMax;
8 }
```

VARIABLES

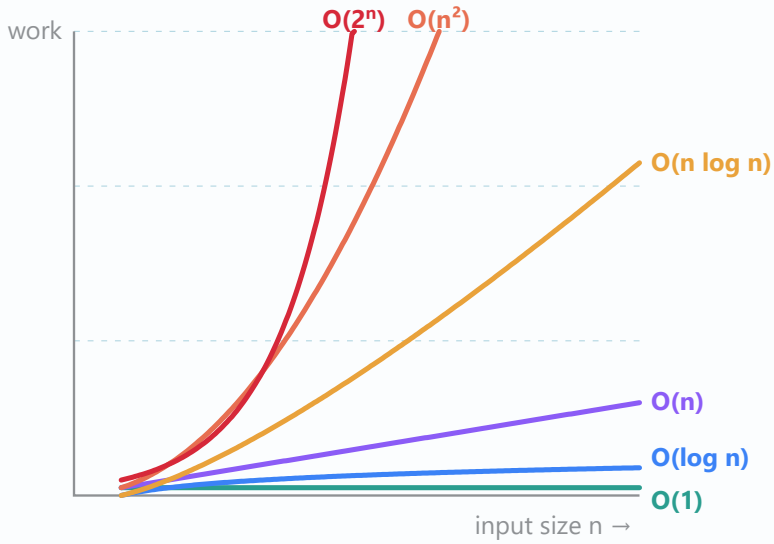
`a = [3, 8, 2, 9, 4]` `currentMax = 9`

`comparisons = 4` `updates = 2`

ASYMPTOTIC NOTATION

Time Complexity and Big O

- Expresses an algorithm's running time **as a function of the input size n**.
- Big O gives an **upper bound** on running time; it is usually quoted for the worst case.
- **O(1)**: the operation count does not grow with the input size, though the time need not be identical for every input.
- **O(n)**: time grows linearly with the input.



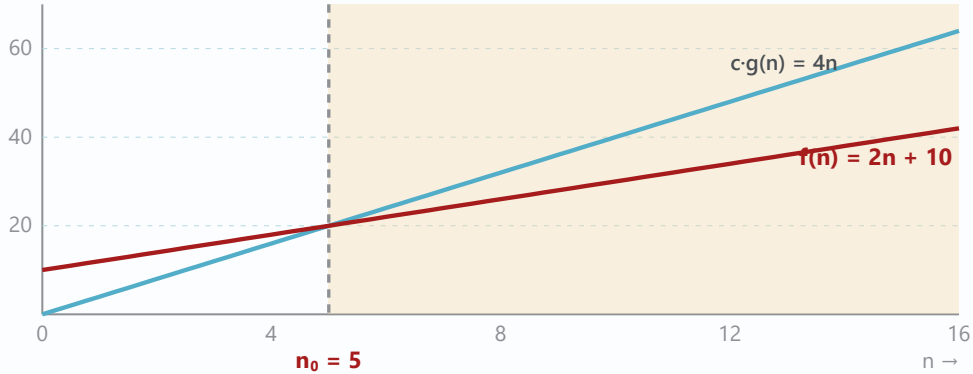
Operations as n grows

Only the fastest-growing term matters in Big O; constants and lower-order terms are dropped: **$3n^2 + 5n + 2 \rightarrow O(n^2)$** · **$4n + 100 \rightarrow O(n)$** · **$7 \rightarrow O(1)$**

ASYMPTOTIC NOTATION

O, Ω, and Θ: Definition and Graphs

Definition: f(n) = O(g(n)) ⇔ there exist c > 0 and $n_0 \geq 1$ such that **f(n) ≤ c·g(n) for every n ≥ n_0**. For Ω use ≥; for Θ, both.

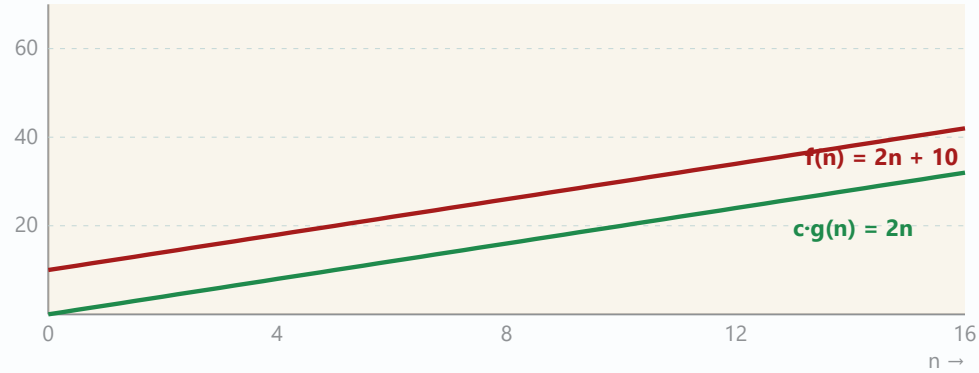


Big O: 2n + 10 ≤ **4**·n for n ≥ n_0 = 5 → f(n) = **O(n)** (c = 4). For small n (n < 5) f(n) is larger, which does not matter.

ASYMPTOTIC NOTATION

O, Ω, and Θ: Definition and Graphs

Definition: f(n) = O(g(n)) ⇔ there exist c > 0 and $n_0 \geq 1$ such that **f(n) ≤ c·g(n) for every n ≥ n_0**. For Ω use ≥; for Θ, both.

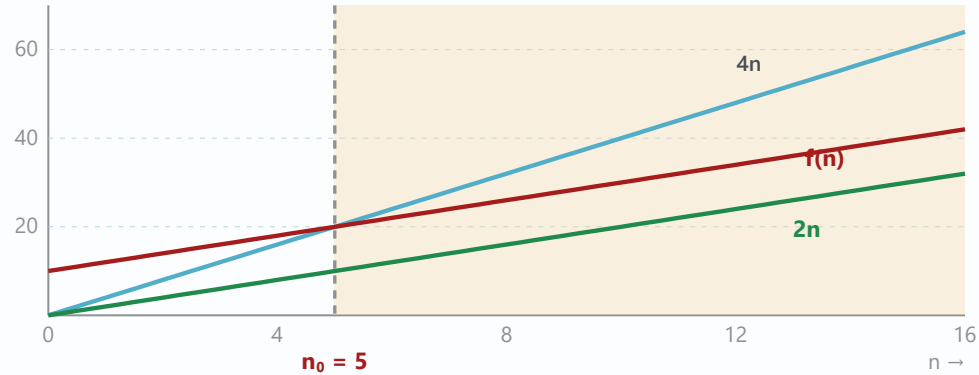


Big Omega: 2n + 10 ≥ **2**·n for every n ≥ 1 → f(n) = **Ω(n)**: f grows no more slowly than n (lower bound).

ASYMPTOTIC NOTATION

O, Ω, and Θ: Definition and Graphs

Definition: f(n) = O(g(n)) ⇔ there exist c > 0 and $n_0 \geq 1$ such that **f(n) ≤ c·g(n) for every n ≥ n_0**. For Ω use ≥; for Θ, both.



Big Theta: 2n ≤ f(n) ≤ 4n for n ≥ 5 → f(n) = **Θ(n)**: squeezed between two lines, it grows exactly like n.

ASYMPTOTIC NOTATION

Asymptotic Bounds and Cases

O(g(n)) · Upper bound
For large enough n, the work grows no faster than g(n).

Ω(g(n)) · Lower bound
For large enough n, the work grows no more slowly than g(n).

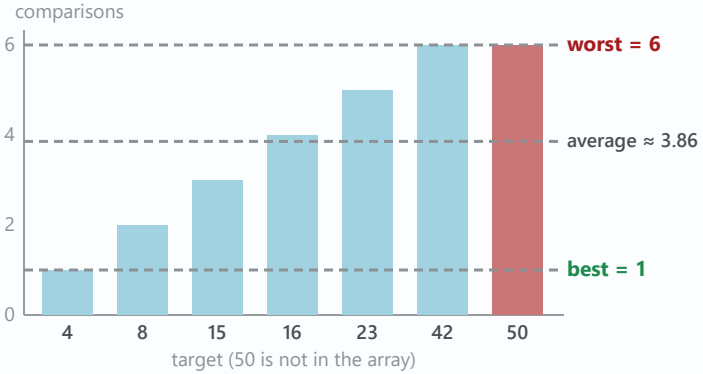
Θ(g(n)) · Tight bound
The upper and lower bounds match; the growth rate is in the same class as g(n).

Analysis	Question	Linear search
Best case	How many operations on the most favorable valid input?	Θ(1) if the first item is the target
Average case	What is the expected cost under an assumed input distribution?	Θ(n) if all positions are equally likely
Worst case	What cost is guaranteed for every valid input?	Θ(n) if the target is last or absent

Amortized analysis: A single operation may be expensive, so we divide the total cost of a long sequence of operations by their number. In a growable array, for example, a resize costs O(n), yet appending is amortized O(1).

Best, Average, and Worst Case

Linear search on [4, 8, 15, 16, 23, 42]: comparisons for each possible target:



- For the same n, the cost depends on the input.
- The **worst case** is a **guarantee** and easy to compute → it is what we usually report.
- The **average** assumes a probability distribution (here 7 equally likely targets, including the absent 50: 27/7 ≈ 3.86).
- The **best case** alone is misleading.

Big O Examples

Notation	Name	Example	Why?
O(1)	Constant	Accessing the first array element	Access time is the same whatever the array size
O(log n)	Logarithmic	Binary search in a sorted array	Each step halves the search range
O(n)	Linear	Scanning an array end to end	Each item is processed once
O(n log n)	Linearithmic	Merge sort	Divide and conquer: log n levels, n work per level
O(n^2)	Quadratic	Comparing all pairs of items	Two nested loops
O(2^n)	Exponential	Listing all subsets	A set of n items has 2^n subsets

What Happens as n Grows?

Complexity	n = 10	n = 100	n = 1,000
$\log_2 n$	3.3	6.6	10
n	10	100	1,000
$n \log_2 n$	33	664	9,966
n^2	100	10,000	1,000,000
2^n	1,024	$\approx 1.3 \cdot 10^{30}$	$\approx 1.1 \cdot 10^{301}$

At 10^9 operations per second, 2^n steps for n = 100 take about **$4 \cdot 10^{13}$ years**. The universe is $\approx 1.4 \cdot 10^{10}$ years old.

From Loop Patterns to Complexity

Pattern	Body runs	n = 16	Class
`for (i = 0; i < n; i++)`	n	16	O(n)
`for (i = 0; i < n; i += 2)`	n / 2	8	O(n)
`for (i = 1; i < n; i *= 2)`	$\lceil \log_2 n \rceil$	4	O(log n)
nested: `i < n`, `j < n`	n^2	256	$O(n^2)$
nested: `i < n`, `j < i`	$n(n-1)/2$	120	$O(n^2)$
two consecutive `i < n` loops	2n	32	O(n)

Consecutive blocks

costs **add**, and the largest wins: $O(n) + O(n^2) = O(n^2)$

Nested loops

costs **multiply**: outer × inner (sum instead if the inner loop depends on the outer)

Code Trace: `countSteps(16)` — i doubles every round



countSteps(16): how many doublings does it take to reach n?

Step 1 / 12

```
▶ int countSteps(int n) {
2   int k = 0;
3   for (int i = 1; i < n; i *= 2)
4     k++;
5   return k;
6 }
```

VARIABLES

n = **16**

Code Trace: `countSteps(16)` — i doubles every round



Counter **k = 0**.

Step 2 / 12

```
1 int countSteps(int n) {
▶   int k = 0;
3   for (int i = 1; i < n; i *= 2)
4     k++;
5   return k;
6 }
```

VARIABLES

n = 16 k = 0

Code Trace: `countSteps(16)` — i doubles every round



i = 1 < 16 → body runs.

Step 3 / 12

```
1  int countSteps(int n) {
2    int k = 0;
▶    for (int i = 1; i < n; i *= 2)
4      k++;
5    return k;
6  }
```

VARIABLES

`n = 16` `k = 0` `i = 1`

Code Trace: `countSteps(16)` — i doubles every round



k++ → k = 1.

Step 4 / 12

```
1  int countSteps(int n) {
2    int k = 0;
3    for (int i = 1; i < n; i *= 2)
▶      k++;
5    return k;
6  }
```

VARIABLES

`n = 16` `k = 1` `i = 1`

Code Trace: `countSteps(16)` — i doubles every round



i = 2 < 16 → body runs.

Step 5 / 12

```
1 int countSteps(int n) {
2   int k = 0;
▶   for (int i = 1; i < n; i *= 2)
4     k++;
5   return k;
6 }
```

VARIABLES

`n = 16` `k = 1` `i = 2`

Code Trace: `countSteps(16)` — i doubles every round



k++ → k = 2.

Step 6 / 12

```
1  int countSteps(int n) {
2    int k = 0;
3    for (int i = 1; i < n; i *= 2)
▶      k++;
5    return k;
6  }
```

VARIABLES

`n = 16` `k = 2` `i = 2`

Code Trace: `countSteps(16)` — i doubles every round



i = 4 < 16 → body runs.

Step 7 / 12

```
1 int countSteps(int n) {
2   int k = 0;
▶   for (int i = 1; i < n; i *= 2)
4     k++;
5   return k;
6 }
```

VARIABLES

`n = 16` `k = 2` `i = 4`

Code Trace: `countSteps(16)` — i doubles every round



k++ → k = 3.

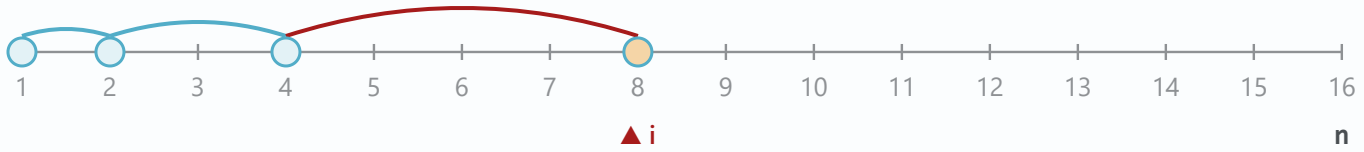
Step 8 / 12

```
1 int countSteps(int n) {
2   int k = 0;
3   for (int i = 1; i < n; i *= 2)
▶     k++;
5   return k;
6 }
```

VARIABLES

`n = 16` `k = 3` `i = 4`

Code Trace: `countSteps(16)` — i doubles every round



i = 8 < 16 → body runs.

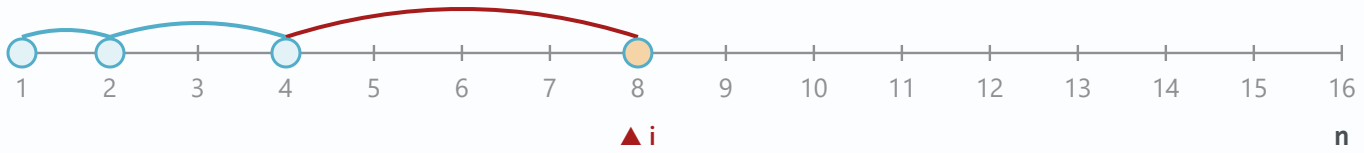
Step 9 / 12

```
1 int countSteps(int n) {
2   int k = 0;
▶   for (int i = 1; i < n; i *= 2)
4     k++;
5   return k;
6 }
```

VARIABLES

`n = 16` `k = 3` `i = 8`

Code Trace: `countSteps(16)` — i doubles every round



k++ → k = 4.

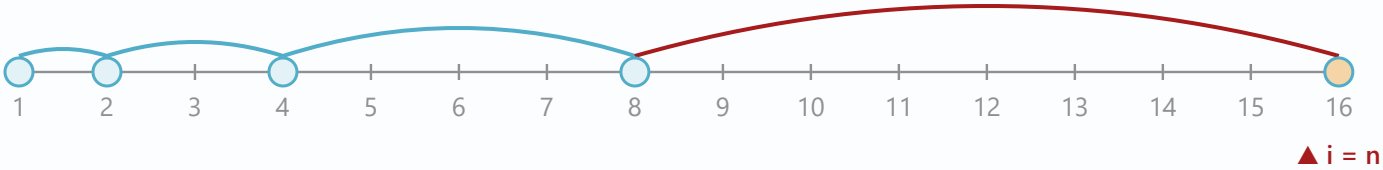
Step 10 / 12

```
1 int countSteps(int n) {
2   int k = 0;
3   for (int i = 1; i < n; i *= 2)
▶     k++;
5   return k;
6 }
```

VARIABLES

n = **16** | k = **4** | i = **8**

Code Trace: `countSteps(16)` — i doubles every round



i *= 2 → **i = 16**; 16 < 16 is false → loop ends.

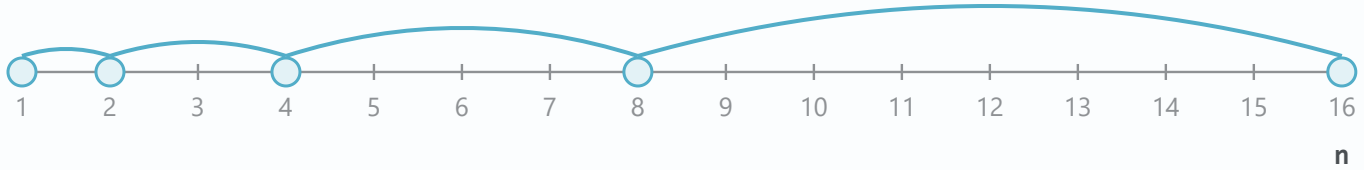
Step 11 / 12

```
1 int countSteps(int n) {
2   int k = 0;
▶   for (int i = 1; i < n; i *= 2)
4     k++;
5   return k;
6 }
```

VARIABLES

`n = 16` `k = 4` `i = 16`

Code Trace: `countSteps(16)` — i doubles every round



return 4 = $\log_2 16$. Making n 1000 times larger adds only ~10 rounds → **O(log n)**.

Step 12 / 12

```
1 int countSteps(int n) {
2   int k = 0;
3   for (int i = 1; i < n; i *= 2)
4     k++;
▶   return k;
6 }
```

VARIABLES

`n = 16` `k = 4`

From Code to Complexity

O(1)

```
int first(int[] a) {
    return a[0];
}
```

O(n)

```
int sum(int[] a) {
    int s = 0;
    for (int x : a)  // n times
        s += x;
    return s;
}
```

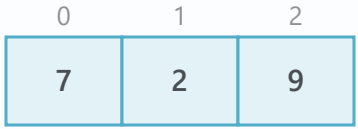
O(n^2)

```
boolean hasDuplicates(int[] a) {
    for (int i = 0; i < a.length; i++)
        for (int j = i + 1; j < a.length; j++)
            if (a[i] == a[j])
                return true;
    return false;
}
// outer loop n, inner ≈ n/2
// → n(n-1)/2 comparisons
```

Rule: drop constants and lower-order terms.

$n(n-1)/2 = n^2/2 - n/2 \rightarrow O(n^2)$

Code Trace: `sum()`



main calls `sum(a)`, and a new call frame is pushed.

Step 1 / 12

```
1  int sum(int[] a) {
2      int s = 0;
3      for (int x : a) {
4          s += x;
5      }
6      return s;
7  }
8
9  // in main:
▶  System.out.println(sum(a));
```

VARIABLES

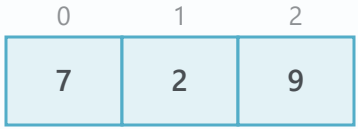
`a = [7, 2, 9]`

CALL STACK

`main()`

OUTPUT

Code Trace: `sum()`



Inside the method, `a` refers to the same array (no copy is made).

Step 2 / 12

```
▶ int sum(int[] a) {
2     int s = 0;
3     for (int x : a) {
4         s += x;
5     }
6     return s;
7 }
8
9 // in main:
10 System.out.println(sum(a));
```

VARIABLES

`a = [7, 2, 9]`

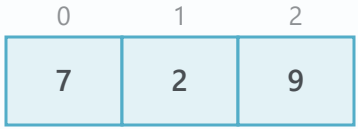
CALL STACK

`sum(a)`

`main()`

OUTPUT

Code Trace: `sum()`



The running total starts at **s = 0**.

Step 3 / 12

```
1  int sum(int[] a) {
▶      int s = 0;
3      for (int x : a) {
4          s += x;
5      }
6      return s;
7  }
8
9  // in main:
10 System.out.println(sum(a));
```

VARIABLES

`a = [7, 2, 9]` `s = 0`

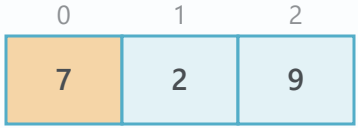
CALL STACK

`sum(a)`

`main()`

OUTPUT

Code Trace: `sum()`



Round 1: **x = a[0] = 7**.

Step 4 / 12

```
1  int sum(int[] a) {
2      int s = 0;
▶      for (int x : a) {
4          s += x;
5      }
6      return s;
7  }
8
9  // in main:
10 System.out.println(sum(a));
```

VARIABLES

`a = [7, 2, 9]` `s = 0` `x = 7`

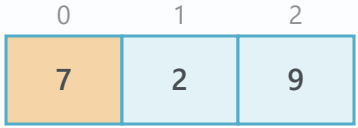
CALL STACK

`sum(a)`

`main()`

OUTPUT

Code Trace: `sum()`



s += x → s = 0 + 7 = **7**.

Step 5 / 12

```
1  int sum(int[] a) {
2      int s = 0;
3      for (int x : a) {
▶          s += x;
5      }
6      return s;
7  }
8
9  // in main:
10 System.out.println(sum(a));
```

VARIABLES

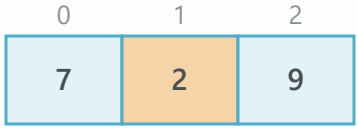
`a = [7, 2, 9]` `s = 7` `x = 7`

CALL STACK

- `sum(a)`
- `main()`

OUTPUT

Code Trace: `sum()`



Round 2: **x = a[1] = 2**.

Step 6 / 12

```
1  int sum(int[] a) {
2      int s = 0;
▶      for (int x : a) {
4          s += x;
5      }
6      return s;
7  }
8
9  // in main:
10 System.out.println(sum(a));
```

VARIABLES

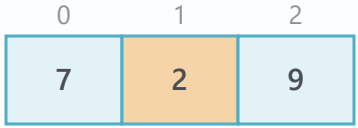
`a = [7, 2, 9]` `s = 7` `x = 2`

CALL STACK

- `sum(a)`
- `main()`

OUTPUT

Code Trace: `sum()`



s += x → s = 7 + 2 = **9**.

Step 7 / 12

```
1  int sum(int[] a) {
2      int s = 0;
3      for (int x : a) {
▶          s += x;
5      }
6      return s;
7  }
8
9  // in main:
10 System.out.println(sum(a));
```

VARIABLES

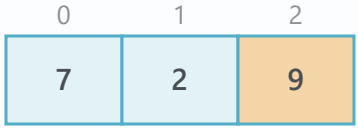
`a = [7, 2, 9]` `s = 9` `x = 2`

CALL STACK

- **sum(a)**
- main()

OUTPUT

Code Trace: `sum()`



Round 3: **x = a[2] = 9**.

Step 8 / 12

```
1  int sum(int[] a) {
2      int s = 0;
▶      for (int x : a) {
4          s += x;
5      }
6      return s;
7  }
8
9  // in main:
10 System.out.println(sum(a));
```

VARIABLES

`a = [7, 2, 9]` `s = 9` `x = 9`

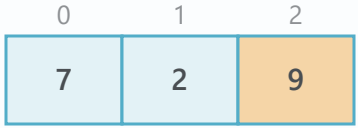
CALL STACK

`sum(a)`

`main()`

OUTPUT

Code Trace: `sum()`



s += x → s = 9 + 9 = **18**.

Step 9 / 12

```
1  int sum(int[] a) {
2      int s = 0;
3      for (int x : a) {
▶          s += x;
5      }
6      return s;
7  }
8
9  // in main:
10 System.out.println(sum(a));
```

VARIABLES

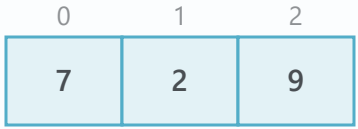
`a = [7, 2, 9]` `s = 18` `x = 9`

CALL STACK

- `sum(a)`
- `main()`

OUTPUT

Code Trace: `sum()`



No items left; the loop exits.

Step 10 / 12

```
1  int sum(int[] a) {
2      int s = 0;
▶      for (int x : a) {
4          s += x;
5      }
6      return s;
7  }
8
9  // in main:
10 System.out.println(sum(a));
```

VARIABLES

`a = [7, 2, 9]` `s = 18`

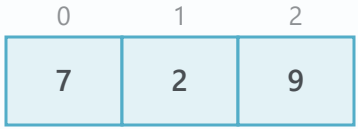
CALL STACK

`sum(a)`

`main()`

OUTPUT

Code Trace: `sum()`



return 18: the frame is popped and the value goes back to the caller.

Step 11 / 12

```
1  int sum(int[] a) {
2      int s = 0;
3      for (int x : a) {
4          s += x;
5      }
▶      return s;
7  }
8
9  // in main:
10 System.out.println(sum(a));
```

VARIABLES

`a = [7, 2, 9]` `s = 18`

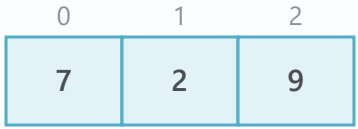
CALL STACK

`sum(a)`

`main()`

OUTPUT

Code Trace: `sum()`



`println` prints 18. The loop body ran **n = 3** times → **O(n)**.

Step 12 / 12

```
1 int sum(int[] a) {
2     int s = 0;
3     for (int x : a) {
4         s += x;
5     }
6     return s;
7 }
8
9 // in main:
▶ System.out.println(sum(a));
```

VARIABLES

`a = [7, 2, 9]`

CALL STACK

`main()`

OUTPUT

```
18
```

Code Trace: `linearSearch(a, 23)`



Searching a sorted array for **23**. Linear search **ignores** the order and checks items one by one from the start.

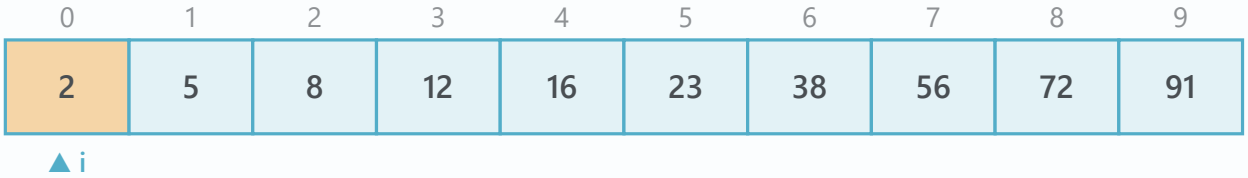
Step 1 / 7

```
▶ int linearSearch(int[] a, int target) {
2     for (int i = 0; i < a.length; i++)
3         if (a[i] == target) return i;
4     return -1;
5 }
```

VARIABLES

`target = 23`

Code Trace: `linearSearch(a, 23)`



a[0] = 2 ≠ 23 → i moves on. Each step rules out only **one** item.

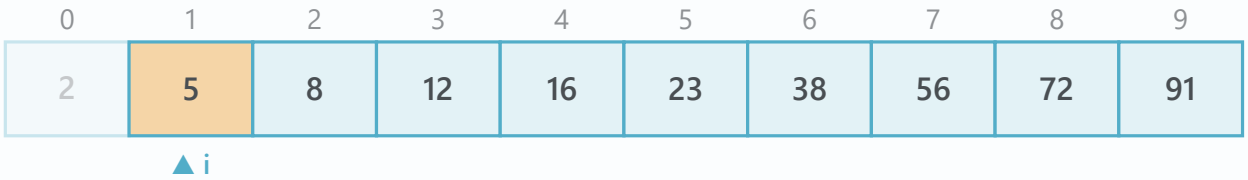
Step 2 / 7

```
1 int linearSearch(int[] a, int target) {
2     for (int i = 0; i < a.length; i++)
▶         if (a[i] == target) return i;
4     return -1;
5 }
```

VARIABLES

`target = 23` `i = 0` `a[i] = 2` `comparisons = 1`

Code Trace: `linearSearch(a, 23)`



a[1] = 5 ≠ 23 → i moves on.

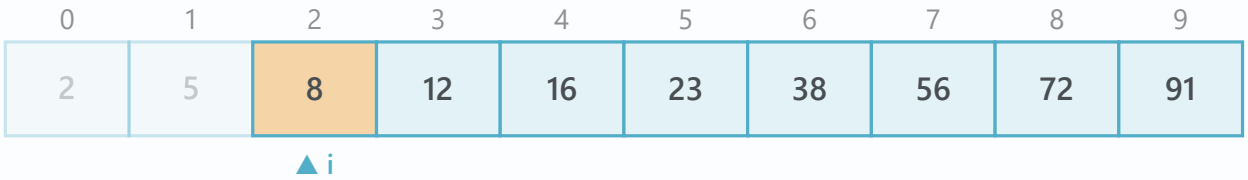
Step 3 / 7

```
1 int linearSearch(int[] a, int target) {
2     for (int i = 0; i < a.length; i++)
▶         if (a[i] == target) return i;
4     return -1;
5 }
```

VARIABLES

`target = 23` `i = 1` `a[i] = 5` `comparisons = 2`

Code Trace: `linearSearch(a, 23)`



a[2] = 8 ≠ 23 → i moves on.

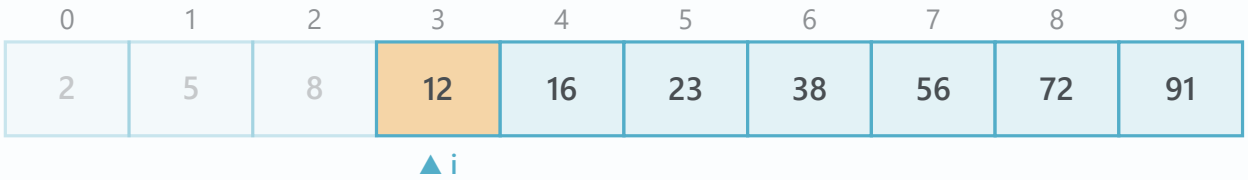
Step 4 / 7

```
1 int linearSearch(int[] a, int target) {
2     for (int i = 0; i < a.length; i++)
▶         if (a[i] == target) return i;
4     return -1;
5 }
```

VARIABLES

`target = 23` `i = 2` `a[i] = 8` `comparisons = 3`

Code Trace: `linearSearch(a, 23)`



a[3] = 12 ≠ 23 → i moves on.

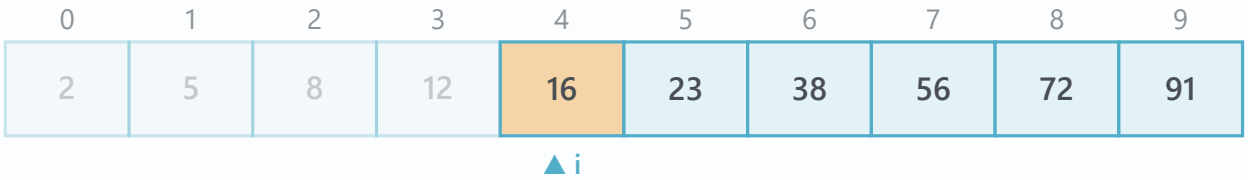
Step 5 / 7

```
1 int linearSearch(int[] a, int target) {
2     for (int i = 0; i < a.length; i++)
▶         if (a[i] == target) return i;
4     return -1;
5 }
```

VARIABLES

`target = 23` `i = 3` `a[i] = 12` `comparisons = 4`

Code Trace: `linearSearch(a, 23)`



a[4] = 16 ≠ 23 → i moves on.

Step 6 / 7

```
1 int linearSearch(int[] a, int target) {
2     for (int i = 0; i < a.length; i++)
▶         if (a[i] == target) return i;
4     return -1;
5 }
```

VARIABLES

target = 23 | i = 4 | a[i] = 16 | comparisons = 5

Code Trace: `linearSearch(a, 23)`



a[5] = 23 == 23 → **return 5**. 6 comparisons; with the target last or absent, n = 10 → **O(n)**.

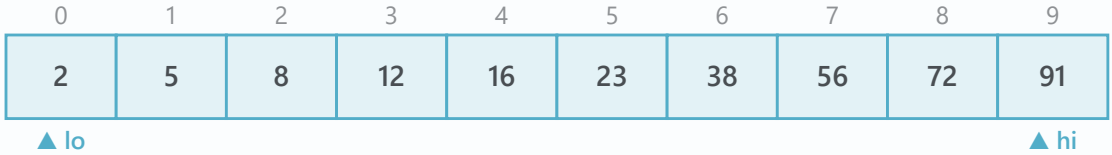
Step 7 / 7

```
1 int linearSearch(int[] a, int target) {
2     for (int i = 0; i < a.length; i++)
▶         if (a[i] == target) return i;
4     return -1;
5 }
```

VARIABLES

`target = 23` `i = 5` `a[i] = 23` `comparisons = 6`

Code Trace: `binarySearch(a, 23)`



The search range is the whole array: **lo = 0**, **hi = 9**.

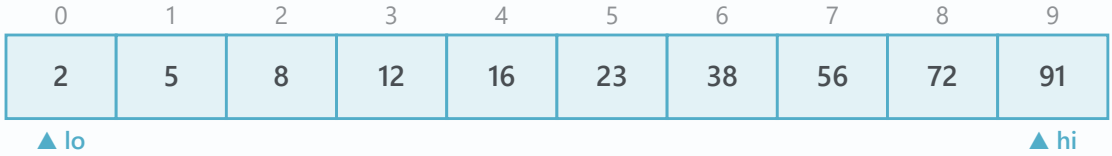
Step 1 / 10

```
1  int binarySearch(int[] a, int target) {
▶      int lo = 0, hi = a.length - 1;
3      while (lo <= hi) {
4          int mid = (lo + hi) / 2;
5          if (a[mid] == target) return mid;
6          if (a[mid] < target) lo = mid + 1;
7          else hi = mid - 1;
8      }
9      return -1;
10 }
```

VARIABLES

`target = 23` `lo = 0` `hi = 9`

Code Trace: `binarySearch(a, 23)`



0 ≤ 9 → range nonempty, round 1 begins.

Step 2 / 10

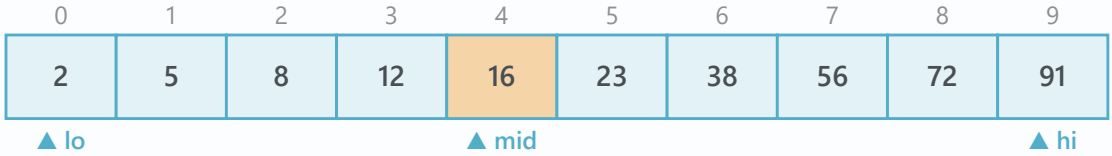
```
1  int binarySearch(int[] a, int target) {
2      int lo = 0, hi = a.length - 1;
▶      while (lo <= hi) {
4          int mid = (lo + hi) / 2;
5          if (a[mid] == target) return mid;
6          if (a[mid] < target) lo = mid + 1;
7          else hi = mid - 1;
8      }
9      return -1;
10 }
```

VARIABLES

`target = 23` `lo = 0` `hi = 9`

`round = 1`

Code Trace: `binarySearch(a, 23)`



mid = (0 + 9) / 2 = 4, a[4] = 16.

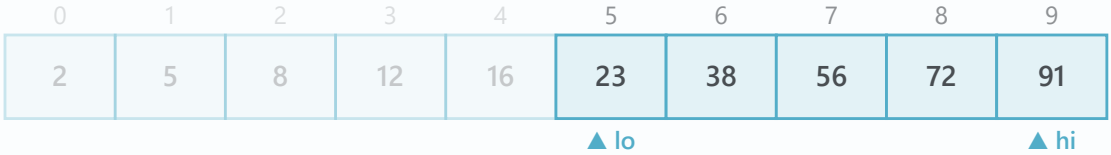
Step 3 / 10

```
1  int binarySearch(int[] a, int target) {
2      int lo = 0, hi = a.length - 1;
3      while (lo <= hi) {
▶          int mid = (lo + hi) / 2;
5          if (a[mid] == target) return mid;
6          if (a[mid] < target) lo = mid + 1;
7          else hi = mid - 1;
8      }
9      return -1;
10 }
```

VARIABLES

`target = 23` `lo = 0` `hi = 9` `mid = 4` `a[mid] = 16` `round = 1`

Code Trace: `binarySearch(a, 23)`



16 < 23 → the target is in the right half: **lo = mid + 1 = 5**. Left half discarded.

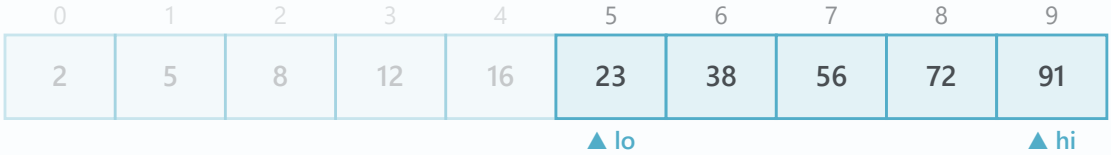
Step 4 / 10

```
1  int binarySearch(int[] a, int target) {
2      int lo = 0, hi = a.length - 1;
3      while (lo <= hi) {
4          int mid = (lo + hi) / 2;
5          if (a[mid] == target) return mid;
▶          if (a[mid] < target) lo = mid + 1;
7          else hi = mid - 1;
8      }
9      return -1;
10 }
```

VARIABLES

`target = 23` `lo = 5` `hi = 9` `mid = 4` `a[mid] = 16` `round = 1`

Code Trace: `binarySearch(a, 23)`



5 ≤ 9 → range nonempty, round 2 begins.

Step 5 / 10

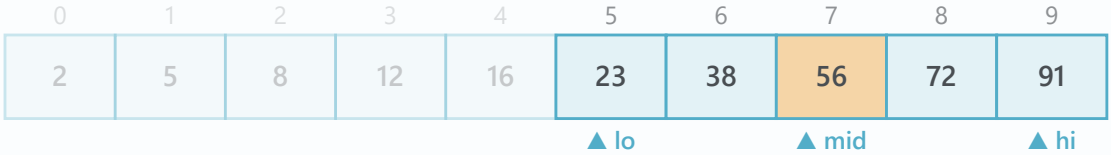
```
1  int binarySearch(int[] a, int target) {
2      int lo = 0, hi = a.length - 1;
▶      while (lo <= hi) {
4          int mid = (lo + hi) / 2;
5          if (a[mid] == target) return mid;
6          if (a[mid] < target) lo = mid + 1;
7          else hi = mid - 1;
8      }
9      return -1;
10 }
```

VARIABLES

`target = 23` `lo = 5` `hi = 9`

`round = 2`

Code Trace: `binarySearch(a, 23)`



mid = (5 + 9) / 2 = 7, a[7] = 56.

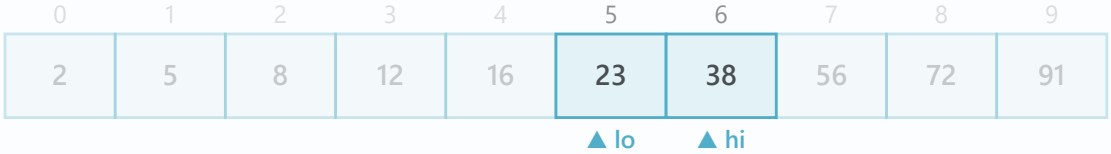
Step 6 / 10

```
1  int binarySearch(int[] a, int target) {
2      int lo = 0, hi = a.length - 1;
3      while (lo <= hi) {
▶          int mid = (lo + hi) / 2;
5          if (a[mid] == target) return mid;
6          if (a[mid] < target) lo = mid + 1;
7          else hi = mid - 1;
8      }
9      return -1;
10 }
```

VARIABLES

`target = 23` `lo = 5` `hi = 9` `mid = 7`
`a[mid] = 56` `round = 2`

Code Trace: `binarySearch(a, 23)`



56 > 23 → the target is in the left half: **hi = mid − 1 = 6**. Right half discarded.

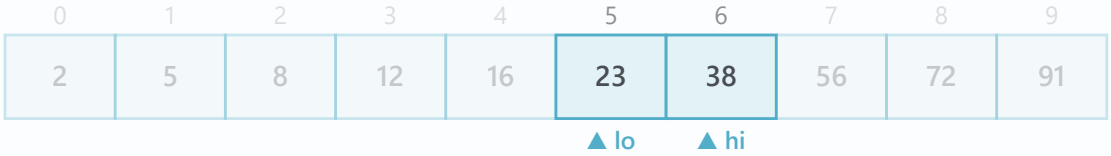
Step 7 / 10

```
1  int binarySearch(int[] a, int target) {
2      int lo = 0, hi = a.length - 1;
3      while (lo <= hi) {
4          int mid = (lo + hi) / 2;
5          if (a[mid] == target) return mid;
6          if (a[mid] < target) lo = mid + 1;
▶          else hi = mid - 1;
8      }
9      return -1;
10 }
```

VARIABLES

`target = 23` `lo = 5` `hi = 6` `mid = 7` `a[mid] = 56` `round = 2`

Code Trace: `binarySearch(a, 23)`



5 ≤ 6 → range nonempty, round 3 begins.

Step 8 / 10

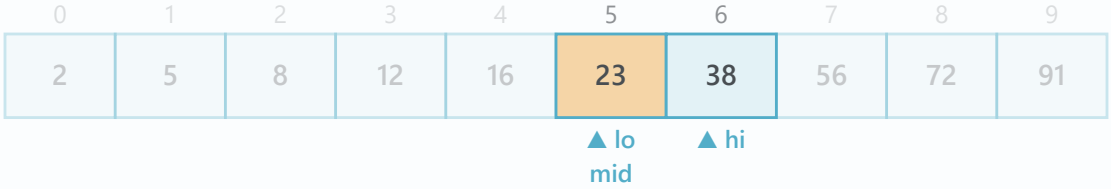
```
1  int binarySearch(int[] a, int target) {
2      int lo = 0, hi = a.length - 1;
▶      while (lo <= hi) {
4          int mid = (lo + hi) / 2;
5          if (a[mid] == target) return mid;
6          if (a[mid] < target) lo = mid + 1;
7          else hi = mid - 1;
8      }
9      return -1;
10 }
```

VARIABLES

`target = 23` `lo = 5` `hi = 6`

`round = 3`

Code Trace: `binarySearch(a, 23)`



mid = (5 + 6) / 2 = 5, a[5] = 23.

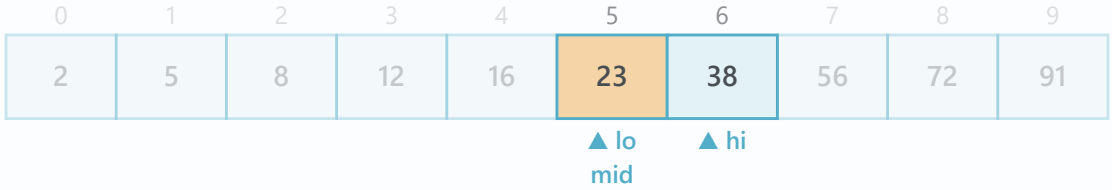
Step 9 / 10

```
1  int binarySearch(int[] a, int target) {
2      int lo = 0, hi = a.length - 1;
3      while (lo <= hi) {
▶          int mid = (lo + hi) / 2;
5          if (a[mid] == target) return mid;
6          if (a[mid] < target) lo = mid + 1;
7          else hi = mid - 1;
8      }
9      return -1;
10 }
```

VARIABLES

`target = 23` `lo = 5` `hi = 6` `mid = 5`
`a[mid] = 23` `round = 3`

Code Trace: `binarySearch(a, 23)`



23 == 23 → found, **return 5**. Only **3 rounds**; linear search needed 6 comparisons. Each round halves the range → **O(log n)**.

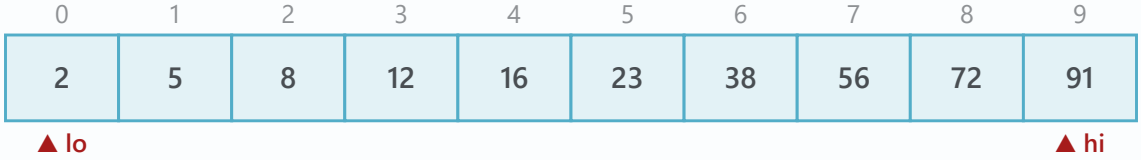
Step 10 / 10

```
1  int binarySearch(int[] a, int target) {
2      int lo = 0, hi = a.length - 1;
3      while (lo <= hi) {
4          int mid = (lo + hi) / 2;
▶          if (a[mid] == target) return mid;
6          if (a[mid] < target) lo = mid + 1;
7          else hi = mid - 1;
8      }
9      return -1;
10 }
```

VARIABLES

`target = 23` `lo = 5` `hi = 6` `mid = 5`
`a[mid] = 23` `round = 3`

Code Trace: `binarySearch(a, 40)` — target absent



This time **40** is **absent**: the worst case. The range is the whole array.

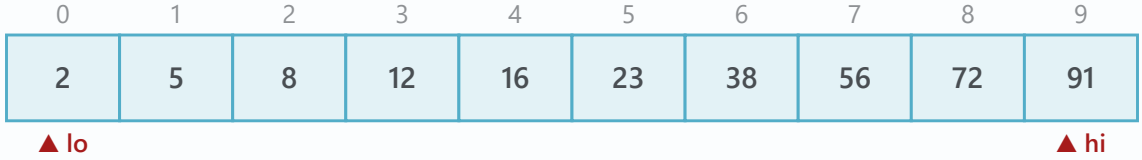
Step 1 / 15

```
1  int binarySearch(int[] a, int target) {
▶      int lo = 0, hi = a.length - 1;
3      while (lo <= hi) {
4          int mid = (lo + hi) / 2;
5          if (a[mid] == target) return mid;
6          if (a[mid] < target) lo = mid + 1;
7          else hi = mid - 1;
8      }
9      return -1;
10 }
```

VARIABLES

`target = 40` `lo = 0` `hi = 9`

Code Trace: `binarySearch(a, 40)` — target absent



0 ≤ 9 → range nonempty, round 1.

Step 2 / 15

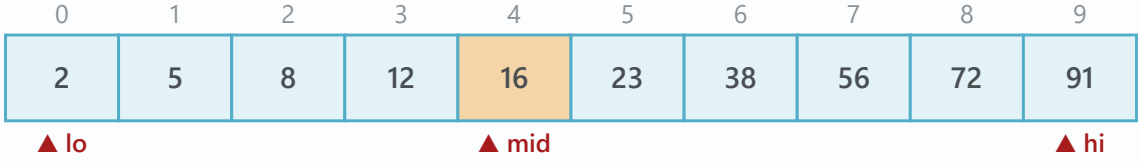
```
1  int binarySearch(int[] a, int target) {
2      int lo = 0, hi = a.length - 1;
▶      while (lo <= hi) {
4          int mid = (lo + hi) / 2;
5          if (a[mid] == target) return mid;
6          if (a[mid] < target) lo = mid + 1;
7          else hi = mid - 1;
8      }
9      return -1;
10 }
```

VARIABLES

`target = 40` `lo = 0` `hi = 9`

`round = 1`

Code Trace: `binarySearch(a, 40)` — target absent



Round 1: **mid = (0 + 9) / 2 = 4**, a[4] = 16.

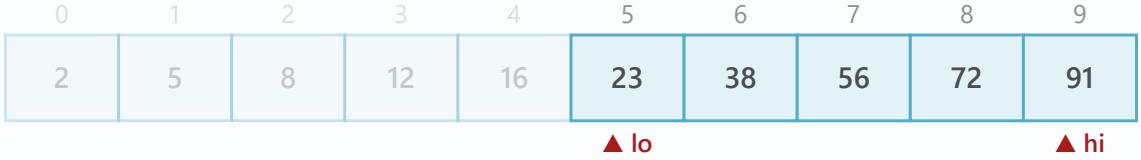
Step 3 / 15

```
1  int binarySearch(int[] a, int target) {
2      int lo = 0, hi = a.length - 1;
3      while (lo <= hi) {
▶          int mid = (lo + hi) / 2;
5          if (a[mid] == target) return mid;
6          if (a[mid] < target) lo = mid + 1;
7          else hi = mid - 1;
8      }
9      return -1;
10 }
```

VARIABLES

`target = 40` `lo = 0` `hi = 9` `mid = 4`
`a[mid] = 16` `round = 1`

Code Trace: `binarySearch(a, 40)` — target absent



16 < 40 → **lo = 5**; left side discarded.

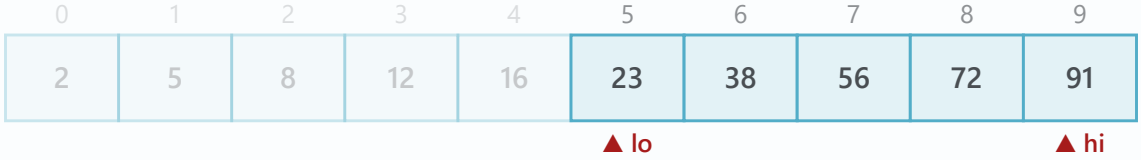
Step 4 / 15

```
1  int binarySearch(int[] a, int target) {
2      int lo = 0, hi = a.length - 1;
3      while (lo <= hi) {
4          int mid = (lo + hi) / 2;
5          if (a[mid] == target) return mid;
▶          if (a[mid] < target) lo = mid + 1;
7          else hi = mid - 1;
8      }
9      return -1;
10 }
```

VARIABLES

`target = 40` `lo = 5` `hi = 9` `mid = 4` `a[mid] = 16` `round = 1`

Code Trace: `binarySearch(a, 40)` — target absent



5 ≤ 9 → range nonempty, round 2.

Step 5 / 15

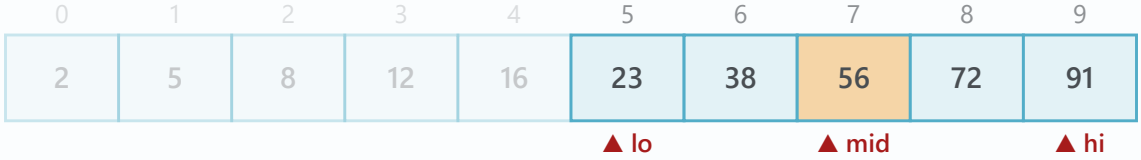
```
1  int binarySearch(int[] a, int target) {
2      int lo = 0, hi = a.length - 1;
▶      while (lo <= hi) {
4          int mid = (lo + hi) / 2;
5          if (a[mid] == target) return mid;
6          if (a[mid] < target) lo = mid + 1;
7          else hi = mid - 1;
8      }
9      return -1;
10 }
```

VARIABLES

`target = 40` `lo = 5` `hi = 9`

`round = 2`

Code Trace: `binarySearch(a, 40)` — target absent



Round 2: **mid = (5 + 9) / 2 = 7**, a[7] = 56.

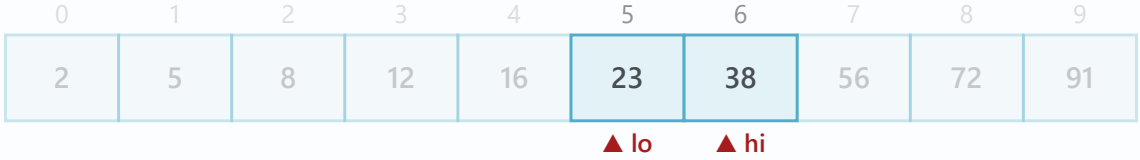
Step 6 / 15

```
1  int binarySearch(int[] a, int target) {
2      int lo = 0, hi = a.length - 1;
3      while (lo <= hi) {
▶          int mid = (lo + hi) / 2;
5          if (a[mid] == target) return mid;
6          if (a[mid] < target) lo = mid + 1;
7          else hi = mid - 1;
8      }
9      return -1;
10 }
```

VARIABLES

`target = 40` `lo = 5` `hi = 9` `mid = 7`
`a[mid] = 56` `round = 2`

Code Trace: `binarySearch(a, 40)` — target absent



56 > 40 → **hi = 6**; right side discarded.

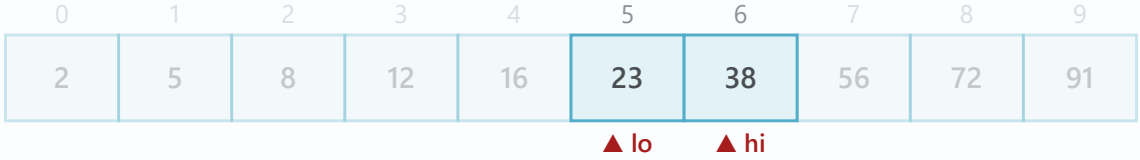
Step 7 / 15

```
1  int binarySearch(int[] a, int target) {
2      int lo = 0, hi = a.length - 1;
3      while (lo <= hi) {
4          int mid = (lo + hi) / 2;
5          if (a[mid] == target) return mid;
6          if (a[mid] < target) lo = mid + 1;
▶          else hi = mid - 1;
8      }
9      return -1;
10 }
```

VARIABLES

`target = 40` `lo = 5` `hi = 6` `mid = 7` `a[mid] = 56` `round = 2`

Code Trace: `binarySearch(a, 40)` — target absent



5 ≤ 6 → range nonempty, round 3.

Step 8 / 15

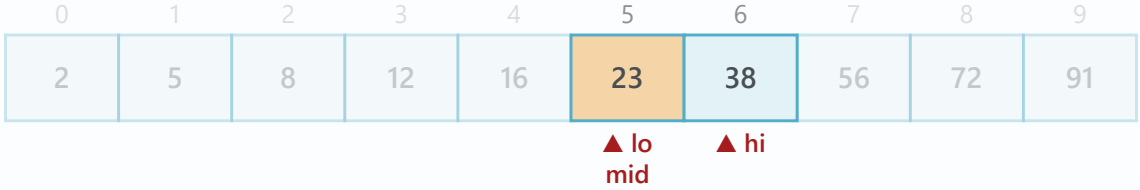
```
1  int binarySearch(int[] a, int target) {
2      int lo = 0, hi = a.length - 1;
▶      while (lo <= hi) {
4          int mid = (lo + hi) / 2;
5          if (a[mid] == target) return mid;
6          if (a[mid] < target) lo = mid + 1;
7          else hi = mid - 1;
8      }
9      return -1;
10 }
```

VARIABLES

`target = 40` `lo = 5` `hi = 6`

`round = 3`

Code Trace: `binarySearch(a, 40)` — target absent



Round 3: **mid = (5 + 6) / 2 = 5**, a[5] = 23.

Step 9 / 15

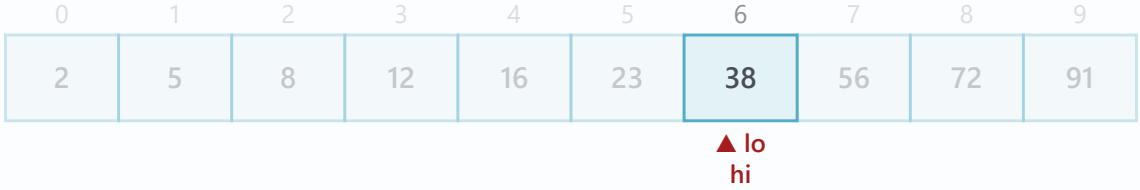
```
1  int binarySearch(int[] a, int target) {
2      int lo = 0, hi = a.length - 1;
3      while (lo <= hi) {
▶          int mid = (lo + hi) / 2;
5          if (a[mid] == target) return mid;
6          if (a[mid] < target) lo = mid + 1;
7          else hi = mid - 1;
8      }
9      return -1;
10 }
```

VARIABLES

`target = 40` `lo = 5` `hi = 6` `mid = 5`

`a[mid] = 23` `round = 3`

Code Trace: `binarySearch(a, 40)` — target absent



23 < 40 → **lo = 6**; left side discarded.

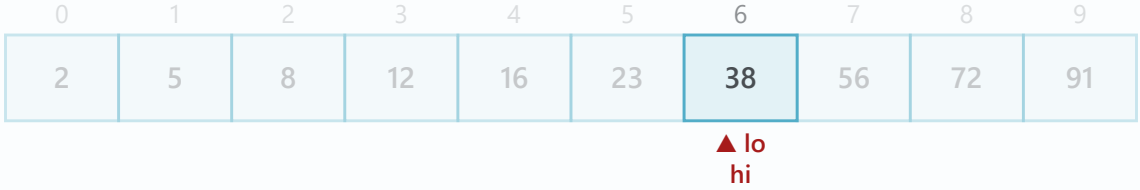
Step 10 / 15

```
1  int binarySearch(int[] a, int target) {
2      int lo = 0, hi = a.length - 1;
3      while (lo <= hi) {
4          int mid = (lo + hi) / 2;
5          if (a[mid] == target) return mid;
▶          if (a[mid] < target) lo = mid + 1;
7          else hi = mid - 1;
8      }
9      return -1;
10 }
```

VARIABLES

target = 40 | lo = 6 | hi = 6 | mid = 5 | a[mid] = 23 | round = 3

Code Trace: `binarySearch(a, 40)` — target absent



6 ≤ 6 → range nonempty, round 4.

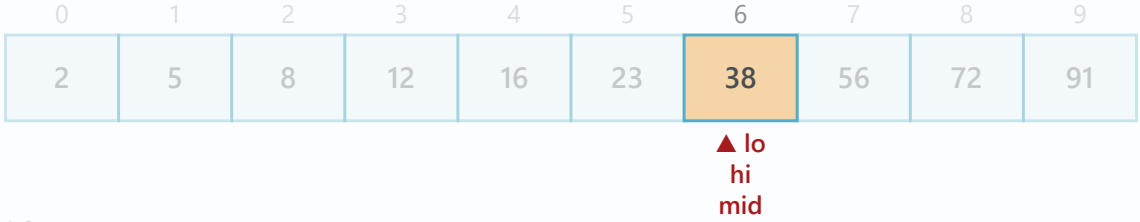
Step 11 / 15

```
1  int binarySearch(int[] a, int target) {
2      int lo = 0, hi = a.length - 1;
▶      while (lo <= hi) {
4          int mid = (lo + hi) / 2;
5          if (a[mid] == target) return mid;
6          if (a[mid] < target) lo = mid + 1;
7          else hi = mid - 1;
8      }
9      return -1;
10 }
```

VARIABLES

`target = 40` `lo = 6` `hi = 6` `round = 4`

Code Trace: `binarySearch(a, 40)` — target absent



Round 4: **mid = (6 + 6) / 2 = 6**, a[6] = 38.

Step 12 / 15

```
1  int binarySearch(int[] a, int target) {
2      int lo = 0, hi = a.length - 1;
3      while (lo <= hi) {
▶          int mid = (lo + hi) / 2;
5          if (a[mid] == target) return mid;
6          if (a[mid] < target) lo = mid + 1;
7          else hi = mid - 1;
8      }
9      return -1;
10 }
```

VARIABLES

`target = 40` `lo = 6` `hi = 6` `mid = 6`

`a[mid] = 38` `round = 4`

Code Trace: `binarySearch(a, 40)` — target absent



38 < 40 → **lo = 7**; left side discarded.

Step 13 / 15

```
1  int binarySearch(int[] a, int target) {
2      int lo = 0, hi = a.length - 1;
3      while (lo <= hi) {
4          int mid = (lo + hi) / 2;
5          if (a[mid] == target) return mid;
▶          if (a[mid] < target) lo = mid + 1;
7          else hi = mid - 1;
8      }
9      return -1;
10 }
```

VARIABLES

`target = 40` `lo = 7` `hi = 6` `mid = 6` `a[mid] = 38` `round = 4`

Code Trace: `binarySearch(a, 40)` — target absent



7 ≤ 6 is false → range empty, loop ends.

Step 14 / 15

```
1  int binarySearch(int[] a, int target) {
2      int lo = 0, hi = a.length - 1;
▶      while (lo <= hi) {
4          int mid = (lo + hi) / 2;
5          if (a[mid] == target) return mid;
6          if (a[mid] < target) lo = mid + 1;
7          else hi = mid - 1;
8      }
9      return -1;
10 }
```

VARIABLES

`target = 40` `lo = 7` `hi = 6` `round = 4`

Code Trace: `binarySearch(a, 40)` — target absent



return -1. Even in the worst case, only **4 rounds** = $\lfloor \log_2 10 \rfloor + 1$; linear search would need 10 comparisons.

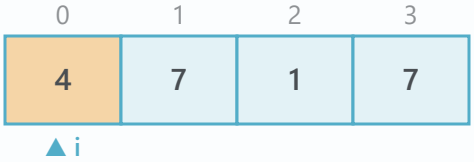
Step 15 / 15

```
1  int binarySearch(int[] a, int target) {
2      int lo = 0, hi = a.length - 1;
3      while (lo <= hi) {
4          int mid = (lo + hi) / 2;
5          if (a[mid] == target) return mid;
6          if (a[mid] < target) lo = mid + 1;
7          else hi = mid - 1;
8      }
▶      return -1;
10 }
```

VARIABLES

`target = 40` `lo = 7` `hi = 6` `round = 4`

Code Trace: `hasDuplicates()`



Outer loop: **i = 0** (a[i] = 4). j starts at i + 1.

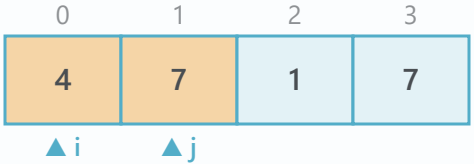
Step 1 / 7

```
1 boolean hasDuplicates(int[] a) {
▶     for (int i = 0; i < a.length; i++)
3         for (int j = i + 1; j < a.length; j++)
4             if (a[i] == a[j])
5                 return true;
6     return false;
7 }
```

VARIABLES

`i = 0` `comparisons = 0`

Code Trace: `hasDuplicates()`



a[0] = 4 vs. a[1] = 7 → different, j moves on.

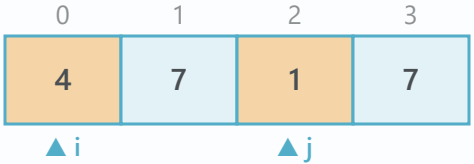
Step 2 / 7

```
1 boolean hasDuplicates(int[] a) {
2     for (int i = 0; i < a.length; i++)
3         for (int j = i + 1; j < a.length; j++)
▶               if (a[i] == a[j])
5                   return true;
6     return false;
7 }
```

VARIABLES

`i = 0` `j = 1` `a[i] = 4` `a[j] = 7` `comparisons = 1`

Code Trace: `hasDuplicates()`



a[0] = 4 vs. a[2] = 1 → different, j moves on.

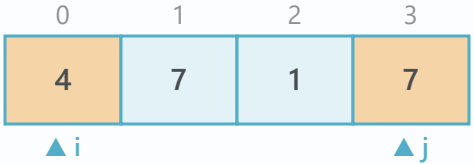
Step 3 / 7

```
1 boolean hasDuplicates(int[] a) {
2     for (int i = 0; i < a.length; i++)
3         for (int j = i + 1; j < a.length; j++)
▶               if (a[i] == a[j])
5                   return true;
6     return false;
7 }
```

VARIABLES

`i = 0` `j = 2` `a[i] = 4` `a[j] = 1` `comparisons = 2`

Code Trace: `hasDuplicates()`



a[0] = 4 vs. a[3] = 7 → different, j moves on.

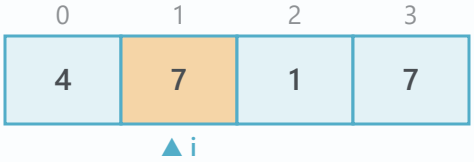
Step 4 / 7

```
1 boolean hasDuplicates(int[] a) {
2     for (int i = 0; i < a.length; i++)
3         for (int j = i + 1; j < a.length; j++)
▶               if (a[i] == a[j])
5                   return true;
6     return false;
7 }
```

VARIABLES

`i = 0` `j = 3` `a[i] = 4` `a[j] = 7` `comparisons = 3`

Code Trace: `hasDuplicates()`



Outer loop: **i = 1** (a[i] = 7). j starts at i + 1.

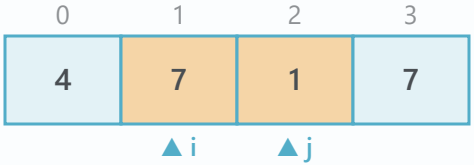
Step 5 / 7

```
1 boolean hasDuplicates(int[] a) {
    for (int i = 0; i < a.length; i++)
3       for (int j = i + 1; j < a.length; j++)
4           if (a[i] == a[j])
5               return true;
6   return false;
7 }
```

VARIABLES

`i = 1` `comparisons = 3`

Code Trace: `hasDuplicates()`



a[1] = 7 vs. a[2] = 1 → different, j moves on.

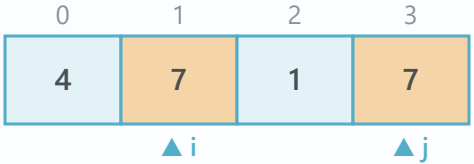
Step 6 / 7

```
1 boolean hasDuplicates(int[] a) {
2     for (int i = 0; i < a.length; i++)
3         for (int j = i + 1; j < a.length; j++)
▶               if (a[i] == a[j])
5                   return true;
6     return false;
7 }
```

VARIABLES

`i = 1` `j = 2` `a[i] = 7` `a[j] = 1` `comparisons = 4`

Code Trace: `hasDuplicates()`



a[1] = 7 and a[3] = 7 are **equal** → **return true**. The worst case (no duplicates) takes n(n−1)/2 = 6 comparisons → **O(n²)**.

Step 7 / 7

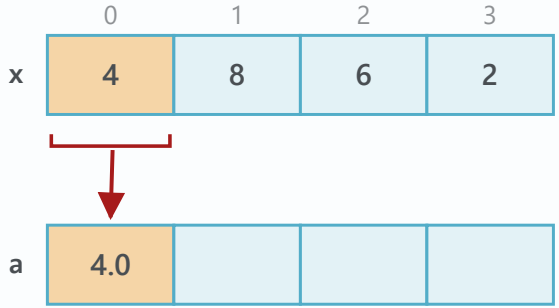
```
1 boolean hasDuplicates(int[] a) {
2     for (int i = 0; i < a.length; i++)
3         for (int j = i + 1; j < a.length; j++)
▶             if (a[i] == a[j])
▶                 return true;
6     return false;
7 }
```

VARIABLES

`i = 1` `j = 3` `a[i] = 7` `a[j] = 7` `comparisons = 5`

Example Problem: Prefix Averages

Input x[0..n−1]; output **a[j] = (x[0] + … + x[j]) / (j + 1)**. Used for moving averages and financial trend analysis (Goodrich §4.3).



a[0] = (4) / 1 = 4.0

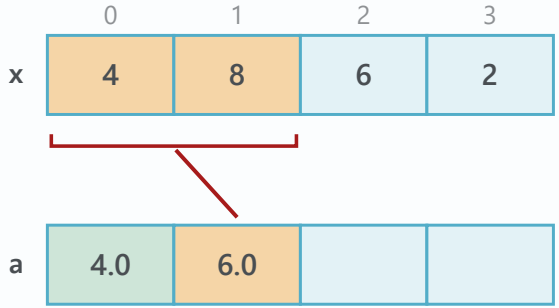
a[0] = average of x[0] … x[0] = **4.0**.

Step 1 / 4

Two solutions: re-sum from scratch for each j ($O(n^2)$) · carry a running sum ($O(n)$).

Example Problem: Prefix Averages

Input x[0..n−1]; output **a[j] = (x[0] + … + x[j]) / (j + 1)**. Used for moving averages and financial trend analysis (Goodrich §4.3).



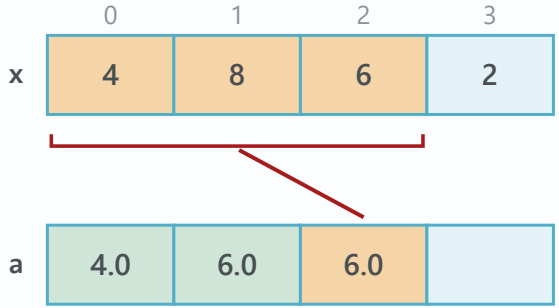
a[1] = (4 + 8) / 2 = 6.0

a[1] = average of x[0] … x[1] = **6.0**.

Two solutions: re-sum from scratch for each j ($O(n^2)$) · carry a running sum ($O(n)$).

Example Problem: Prefix Averages

Input x[0..n−1]; output **a[j] = (x[0] + … + x[j]) / (j + 1)**. Used for moving averages and financial trend analysis (Goodrich §4.3).



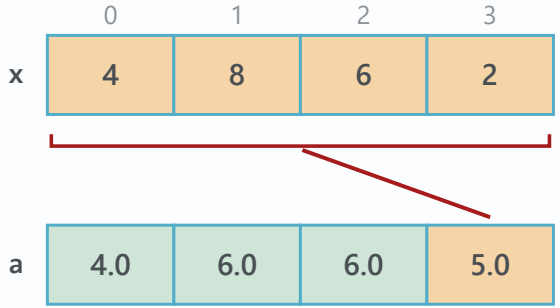
a[2] = (4 + 8 + 6) / 3 = 6.0

a[2] = average of x[0] … x[2] = **6.0**.

Two solutions: re-sum from scratch for each j ($O(n^2)$) · carry a running sum ($O(n)$).

Example Problem: Prefix Averages

Input x[0..n−1]; output **a[j] = (x[0] + … + x[j]) / (j + 1)**. Used for moving averages and financial trend analysis (Goodrich §4.3).



a[3] = (4 + 8 + 6 + 2) / 4 = 5.0

a[3] = average of x[0] … x[3] = **5.0**.

Step 4 / 4

Two solutions: re-sum from scratch for each j ($O(n^2)$) · carry a running sum ($O(n)$).

Code Trace: `prefixAverage1` — re-sum from scratch for each j



n = 4; the result array **a** is created, filled with 0.0.

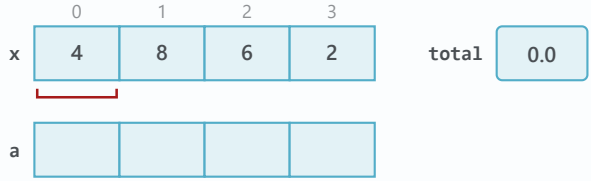
Step 1 / 20

```
1  double[] prefixAverage1(double[] x) {
▶    int n = x.length;
▶    double[] a = new double[n];
4    for (int j = 0; j < n; j++) {
5      double total = 0;
6      for (int i = 0; i <= j; i++)
7        total += x[i];
8      a[j] = total / (j + 1);
9    }
10   return a;
11 }
```

VARIABLES

`n = 4` `a = [0.0, 0.0, 0.0, 0.0]`

Code Trace: `prefixAverage1` — re-sum from scratch for each j



j = 0: total = 0 — each j re-sums **from scratch**.

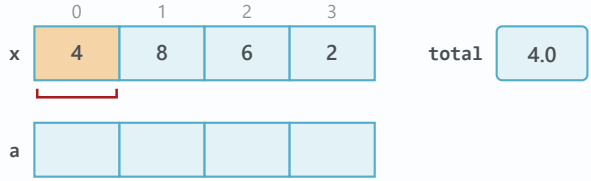
Step 2 / 20

```
1  double[] prefixAverage1(double[] x) {
2    int n = x.length;
3    double[] a = new double[n];
4    for (int j = 0; j < n; j++) {
▶      double total = 0;
6      for (int i = 0; i <= j; i++)
7        total += x[i];
8      a[j] = total / (j + 1);
9    }
10   return a;
11 }
```

VARIABLES

`j = 0` `total = 0.0` `additions = 0`

Code Trace: `prefixAverage1` — re-sum from scratch for each j



total += x[0] = 4 → total = 4.0.

Step 3 / 20

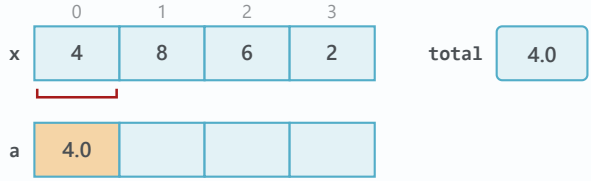
```
1  double[] prefixAverage1(double[] x) {
2    int n = x.length;
3    double[] a = new double[n];
4    for (int j = 0; j < n; j++) {
5      double total = 0;
6      for (int i = 0; i <= j; i++)
▶        total += x[i];
8      a[j] = total / (j + 1);
9    }
10   return a;
11 }
```

VARIABLES

`j = 0` `i = 0` `total = 4.0`

`additions = 1`

Code Trace: `prefixAverage1` — re-sum from scratch for each j



a[0] = 4.0 / 1 = **4.0**.

Step 4 / 20

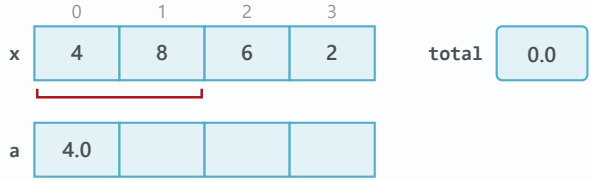
```
1  double[] prefixAverage1(double[] x) {
2    int n = x.length;
3    double[] a = new double[n];
4    for (int j = 0; j < n; j++) {
5      double total = 0;
6      for (int i = 0; i <= j; i++)
7        total += x[i];
▶      a[j] = total / (j + 1);
9    }
10   return a;
11 }
```

VARIABLES

`j = 0` `total = 4.0`

`a = [4.0, 0.0, 0.0, 0.0]` `additions = 1`

Code Trace: `prefixAverage1` — re-sum from scratch for each j



j = 1: total = 0 — each j re-sums **from scratch**.

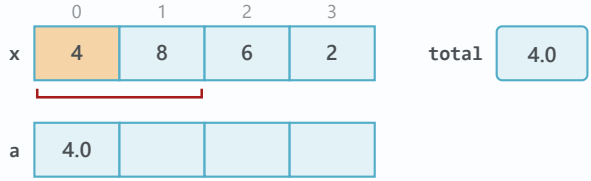
Step 5 / 20

```
1  double[] prefixAverage1(double[] x) {
2    int n = x.length;
3    double[] a = new double[n];
4    for (int j = 0; j < n; j++) {
▶      double total = 0;
6      for (int i = 0; i <= j; i++)
7        total += x[i];
8      a[j] = total / (j + 1);
9    }
10   return a;
11 }
```

VARIABLES

`j = 1` `total = 0.0` `additions = 1`

Code Trace: `prefixAverage1` — re-sum from scratch for each j



total += x[0] = 4 → total = 4.0.

Step 6 / 20

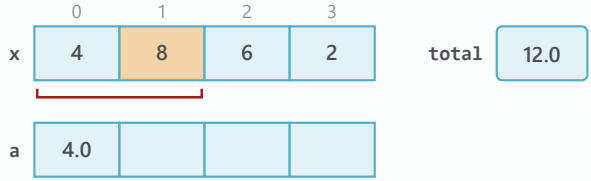
```
1  double[] prefixAverage1(double[] x) {
2    int n = x.length;
3    double[] a = new double[n];
4    for (int j = 0; j < n; j++) {
5      double total = 0;
6      for (int i = 0; i <= j; i++)
▶        total += x[i];
8      a[j] = total / (j + 1);
9    }
10   return a;
11 }
```

VARIABLES

`j = 1` `i = 0` `total = 4.0`

`additions = 2`

Code Trace: `prefixAverage1` — re-sum from scratch for each j



total += x[1] = 8 → total = 12.0.

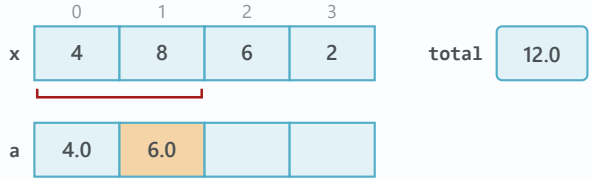
Step 7 / 20

```
1  double[] prefixAverage1(double[] x) {
2    int n = x.length;
3    double[] a = new double[n];
4    for (int j = 0; j < n; j++) {
5      double total = 0;
6      for (int i = 0; i <= j; i++)
▶        total += x[i];
8      a[j] = total / (j + 1);
9    }
10   return a;
11 }
```

VARIABLES

`j = 1` `i = 1` `total = 12.0` `additions = 3`

Code Trace: `prefixAverage1` — re-sum from scratch for each j



a[1] = 12.0 / 2 = **6.0**.

Step 8 / 20

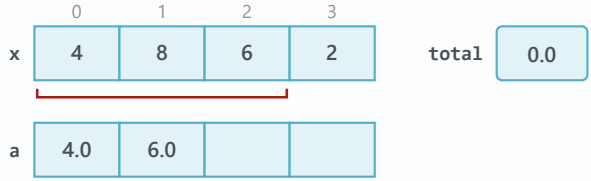
```
1  double[] prefixAverage1(double[] x) {
2    int n = x.length;
3    double[] a = new double[n];
4    for (int j = 0; j < n; j++) {
5      double total = 0;
6      for (int i = 0; i <= j; i++)
7        total += x[i];
▶      a[j] = total / (j + 1);
9    }
10   return a;
11 }
```

VARIABLES

`j = 1` `total = 12.0`

`a = [4.0, 6.0, 0.0, 0.0]` `additions = 3`

Code Trace: `prefixAverage1` — re-sum from scratch for each j



j = 2: total = 0 — each j re-sums **from scratch**.

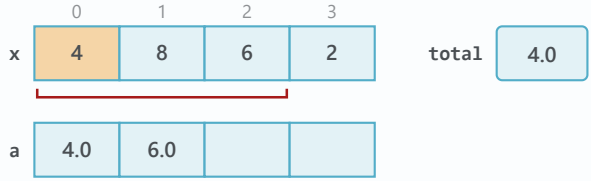
Step 9 / 20

```
1  double[] prefixAverage1(double[] x) {
2    int n = x.length;
3    double[] a = new double[n];
4    for (int j = 0; j < n; j++) {
▶      double total = 0;
6      for (int i = 0; i <= j; i++)
7        total += x[i];
8      a[j] = total / (j + 1);
9    }
10   return a;
11 }
```

VARIABLES

`j = 2` `total = 0.0` `additions = 3`

Code Trace: `prefixAverage1` — re-sum from scratch for each j



total += x[0] = 4 → total = 4.0.

Step 10 / 20

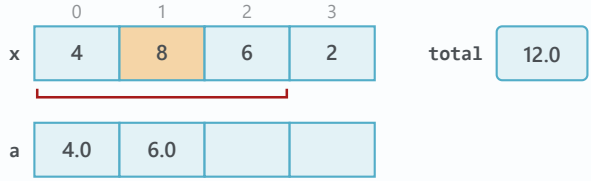
```
1  double[] prefixAverage1(double[] x) {
2    int n = x.length;
3    double[] a = new double[n];
4    for (int j = 0; j < n; j++) {
5      double total = 0;
6      for (int i = 0; i <= j; i++)
▶        total += x[i];
8      a[j] = total / (j + 1);
9    }
10   return a;
11 }
```

VARIABLES

`j = 2` `i = 0` `total = 4.0`

`additions = 4`

Code Trace: `prefixAverage1` — re-sum from scratch for each j



total += x[1] = 8 → total = 12.0.

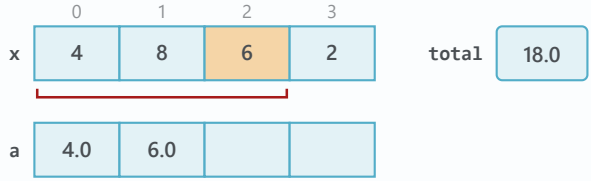
Step 11 / 20

```
1  double[] prefixAverage1(double[] x) {
2    int n = x.length;
3    double[] a = new double[n];
4    for (int j = 0; j < n; j++) {
5      double total = 0;
6      for (int i = 0; i <= j; i++)
▶        total += x[i];
8      a[j] = total / (j + 1);
9    }
10   return a;
11 }
```

VARIABLES

`j = 2` `i = 1` `total = 12.0` `additions = 5`

Code Trace: `prefixAverage1` — re-sum from scratch for each j



total += x[2] = 6 → total = 18.0.

Step 12 / 20

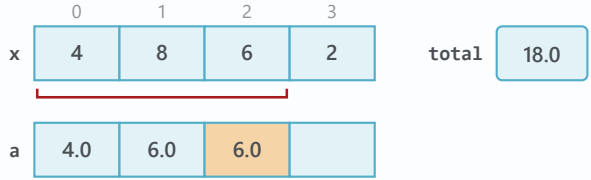
```
1  double[] prefixAverage1(double[] x) {
2    int n = x.length;
3    double[] a = new double[n];
4    for (int j = 0; j < n; j++) {
5      double total = 0;
6      for (int i = 0; i <= j; i++)
▶        total += x[i];
8      a[j] = total / (j + 1);
9    }
10   return a;
11 }
```

VARIABLES

`j = 2` `i = 2` `total = 18.0`

`additions = 6`

Code Trace: `prefixAverage1` — re-sum from scratch for each j



a[2] = 18.0 / 3 = **6.0**.

Step 13 / 20

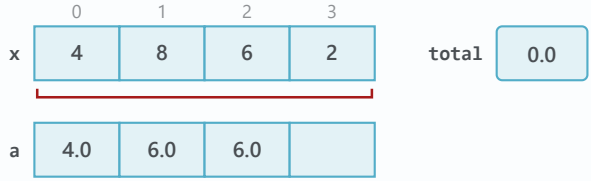
```
1  double[] prefixAverage1(double[] x) {
2    int n = x.length;
3    double[] a = new double[n];
4    for (int j = 0; j < n; j++) {
5      double total = 0;
6      for (int i = 0; i <= j; i++)
7        total += x[i];
▶      a[j] = total / (j + 1);
9    }
10   return a;
11 }
```

VARIABLES

`j = 2` `total = 18.0`

`a = [4.0, 6.0, 6.0, 0.0]` `additions = 6`

Code Trace: `prefixAverage1` — re-sum from scratch for each j



j = 3: total = 0 — each j re-sums **from scratch**.

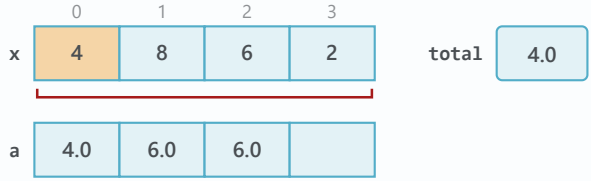
Step 14 / 20

```
1  double[] prefixAverage1(double[] x) {
2    int n = x.length;
3    double[] a = new double[n];
4    for (int j = 0; j < n; j++) {
▶      double total = 0;
6      for (int i = 0; i <= j; i++)
7        total += x[i];
8      a[j] = total / (j + 1);
9    }
10   return a;
11 }
```

VARIABLES

`j = 3` `total = 0.0` `additions = 6`

Code Trace: `prefixAverage1` — re-sum from scratch for each j



total += x[0] = 4 → total = 4.0.

Step 15 / 20

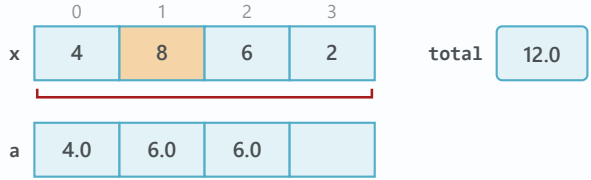
```
1  double[] prefixAverage1(double[] x) {
2    int n = x.length;
3    double[] a = new double[n];
4    for (int j = 0; j < n; j++) {
5      double total = 0;
6      for (int i = 0; i <= j; i++)
▶        total += x[i];
8      a[j] = total / (j + 1);
9    }
10   return a;
11 }
```

VARIABLES

`j = 3` `i = 0` `total = 4.0`

`additions = 7`

Code Trace: `prefixAverage1` — re-sum from scratch for each j



total += x[1] = 8 → total = 12.0.

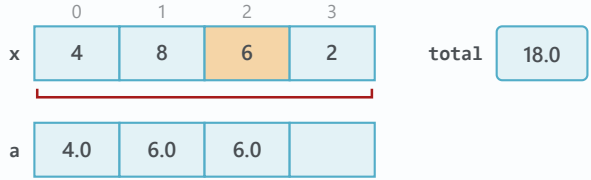
Step 16 / 20

```
1  double[] prefixAverage1(double[] x) {
2    int n = x.length;
3    double[] a = new double[n];
4    for (int j = 0; j < n; j++) {
5      double total = 0;
6      for (int i = 0; i <= j; i++)
▶        total += x[i];
8      a[j] = total / (j + 1);
9    }
10   return a;
11 }
```

VARIABLES

`j = 3` `i = 1` `total = 12.0` `additions = 8`

Code Trace: `prefixAverage1` — re-sum from scratch for each j



total += x[2] = 6 → total = 18.0.

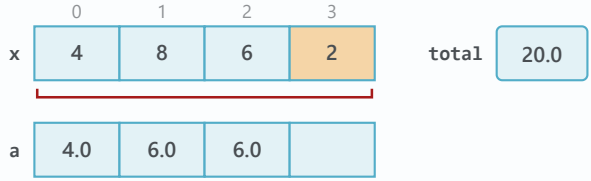
Step 17 / 20

```
1  double[] prefixAverage1(double[] x) {
2    int n = x.length;
3    double[] a = new double[n];
4    for (int j = 0; j < n; j++) {
5      double total = 0;
6      for (int i = 0; i <= j; i++)
▶        total += x[i];
8      a[j] = total / (j + 1);
9    }
10   return a;
11 }
```

VARIABLES

`j = 3` `i = 2` `total = 18.0` `additions = 9`

Code Trace: `prefixAverage1` — re-sum from scratch for each j



total += x[3] = 2 → total = 20.0.

Step 18 / 20

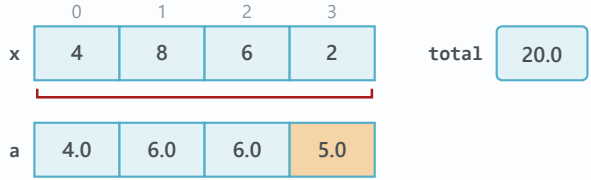
```
1  double[] prefixAverage1(double[] x) {
2    int n = x.length;
3    double[] a = new double[n];
4    for (int j = 0; j < n; j++) {
5      double total = 0;
6      for (int i = 0; i <= j; i++)
▶        total += x[i];
8      a[j] = total / (j + 1);
9    }
10   return a;
11 }
```

VARIABLES

`j = 3` `i = 3` `total = 20.0`

`additions = 10`

Code Trace: `prefixAverage1` — re-sum from scratch for each j



a[3] = 20.0 / 4 = **5.0**.

Step 19 / 20

```
1  double[] prefixAverage1(double[] x) {
2    int n = x.length;
3    double[] a = new double[n];
4    for (int j = 0; j < n; j++) {
5      double total = 0;
6      for (int i = 0; i <= j; i++)
7        total += x[i];
▶      a[j] = total / (j + 1);
9    }
10   return a;
11 }
```

VARIABLES

`j = 3` `total = 20.0`

`a = [4.0, 6.0, 6.0, 5.0]` `additions = 10`

Code Trace: `prefixAverage1` — re-sum from scratch for each j



return a. Additions: 1 + 2 + 3 + 4 = **10** = n(n+1)/2 → **O(n²)**.

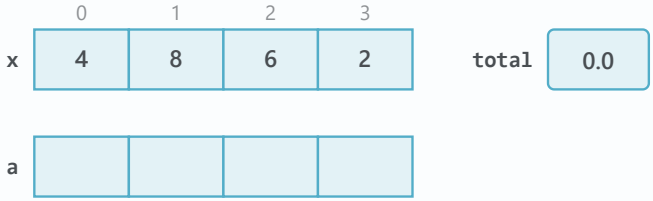
Step 20 / 20

```
1  double[] prefixAverage1(double[] x) {
2    int n = x.length;
3    double[] a = new double[n];
4    for (int j = 0; j < n; j++) {
5      double total = 0;
6      for (int i = 0; i <= j; i++)
7        total += x[i];
8      a[j] = total / (j + 1);
9    }
▶    return a;
11 }
```

VARIABLES

`a = [4.0, 6.0, 6.0, 5.0]` `additions = 10`

Code Trace: `prefixAverage2` — running sum



total is set to 0 once, **before** the loop, and carried across rounds.

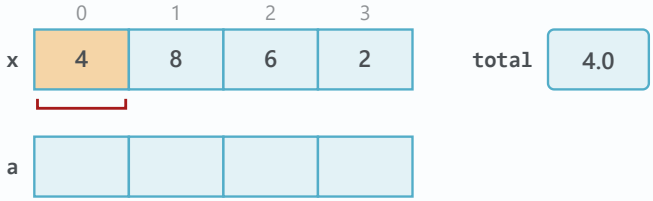
Step 1 / 10

```
1  double[] prefixAverage2(double[] x) {
▶    int n = x.length;
▶    double[] a = new double[n];
▶    double total = 0;
5    for (int j = 0; j < n; j++) {
6      total += x[j];
7      a[j] = total / (j + 1);
8    }
9    return a;
10 }
```

VARIABLES

`n = 4` `a = [0.0, 0.0, 0.0, 0.0]` `total = 0.0`

Code Trace: `prefixAverage2` — running sum



j = 0: total += x[0] = 4 → total = 4.0 (previous sum reused).

Step 2 / 10

```
1  double[] prefixAverage2(double[] x) {
2    int n = x.length;
3    double[] a = new double[n];
4    double total = 0;
5    for (int j = 0; j < n; j++) {
▶      total += x[j];
7      a[j] = total / (j + 1);
8    }
9    return a;
10 }
```

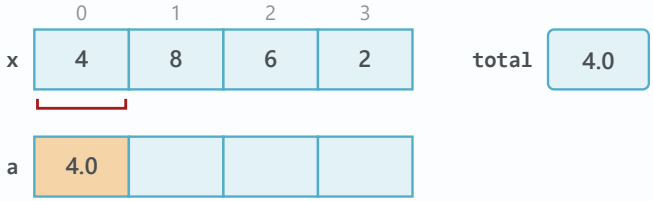
VARIABLES

`j = 0` `total = 4.0`

`a = [0.0, 0.0, 0.0, 0.0]`

`additions = 1`

Code Trace: `prefixAverage2` — running sum



a[0] = 4.0 / 1 = **4.0**.

Step 3 / 10

```
1  double[] prefixAverage2(double[] x) {
2    int n = x.length;
3    double[] a = new double[n];
4    double total = 0;
5    for (int j = 0; j < n; j++) {
6      total += x[j];
▶      a[j] = total / (j + 1);
8    }
9    return a;
10 }
```

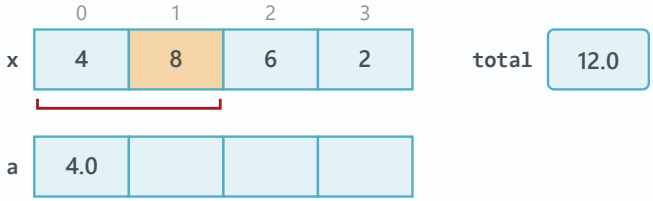
VARIABLES

`j = 0` `total = 4.0`

`a = [4.0, 0.0, 0.0, 0.0]`

`additions = 1`

Code Trace: `prefixAverage2` — running sum



j = 1: total += x[1] = 8 → total = 12.0 (previous sum reused).

Step 4 / 10

```
1  double[] prefixAverage2(double[] x) {
2    int n = x.length;
3    double[] a = new double[n];
4    double total = 0;
5    for (int j = 0; j < n; j++) {
▶      total += x[j];
7      a[j] = total / (j + 1);
8    }
9    return a;
10 }
```

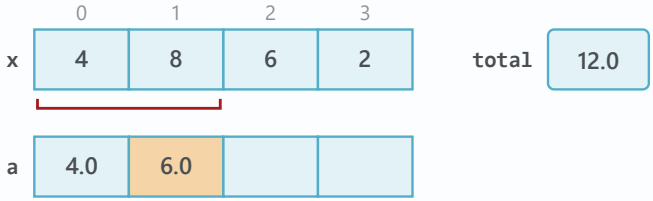
VARIABLES

`j = 1` `total = 12.0`

`a = [4.0, 0.0, 0.0, 0.0]`

`additions = 2`

Code Trace: `prefixAverage2` — running sum



a[1] = 12.0 / 2 = **6.0**.

Step 5 / 10

```
1  double[] prefixAverage2(double[] x) {
2    int n = x.length;
3    double[] a = new double[n];
4    double total = 0;
5    for (int j = 0; j < n; j++) {
6      total += x[j];
▶      a[j] = total / (j + 1);
8    }
9    return a;
10 }
```

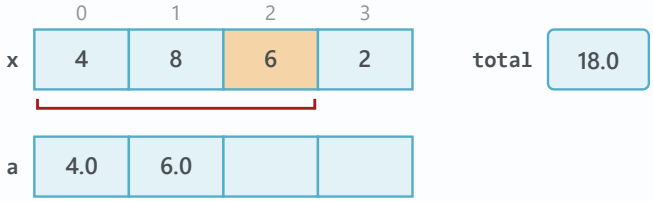
VARIABLES

`j = 1` `total = 12.0`

`a = [4.0, 6.0, 0.0, 0.0]`

`additions = 2`

Code Trace: `prefixAverage2` — running sum



j = 2: total += x[2] = 6 → total = 18.0 (previous sum reused).

Step 6 / 10

```
1  double[] prefixAverage2(double[] x) {
2    int n = x.length;
3    double[] a = new double[n];
4    double total = 0;
5    for (int j = 0; j < n; j++) {
▶      total += x[j];
7      a[j] = total / (j + 1);
8    }
9    return a;
10 }
```

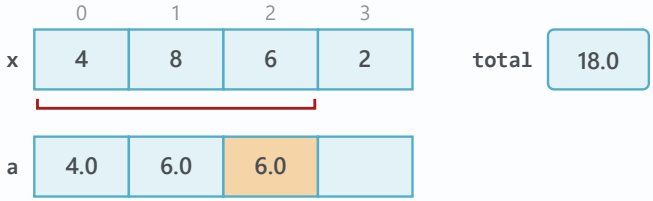
VARIABLES

`j = 2` `total = 18.0`

`a = [4.0, 6.0, 0.0, 0.0]`

`additions = 3`

Code Trace: `prefixAverage2` — running sum



a[2] = 18.0 / 3 = **6.0**.

Step 7 / 10

```
1  double[] prefixAverage2(double[] x) {
2    int n = x.length;
3    double[] a = new double[n];
4    double total = 0;
5    for (int j = 0; j < n; j++) {
6      total += x[j];
▶      a[j] = total / (j + 1);
8    }
9    return a;
10 }
```

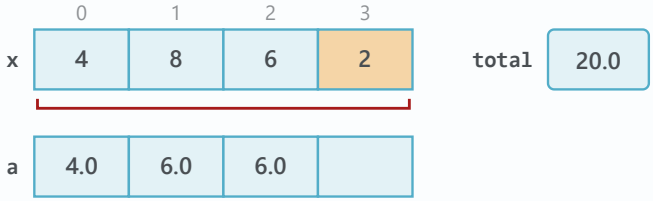
VARIABLES

`j = 2` `total = 18.0`

`a = [4.0, 6.0, 6.0, 0.0]`

`additions = 3`

Code Trace: `prefixAverage2` — running sum



j = 3: total += x[3] = 2 → total = 20.0 (previous sum reused).

Step 8 / 10

```
1  double[] prefixAverage2(double[] x) {
2    int n = x.length;
3    double[] a = new double[n];
4    double total = 0;
5    for (int j = 0; j < n; j++) {
▶      total += x[j];
7      a[j] = total / (j + 1);
8    }
9    return a;
10 }
```

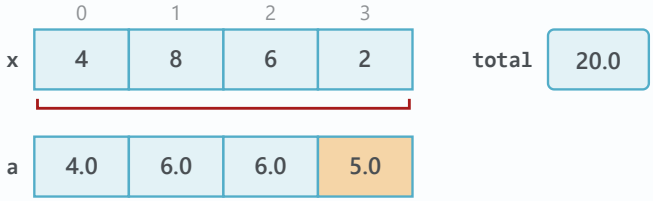
VARIABLES

`j = 3` `total = 20.0`

`a = [4.0, 6.0, 6.0, 0.0]`

`additions = 4`

Code Trace: `prefixAverage2` — running sum



a[3] = 20.0 / 4 = **5.0**.

Step 9 / 10

```
1  double[] prefixAverage2(double[] x) {
2    int n = x.length;
3    double[] a = new double[n];
4    double total = 0;
5    for (int j = 0; j < n; j++) {
6      total += x[j];
▶      a[j] = total / (j + 1);
8    }
9    return a;
10 }
```

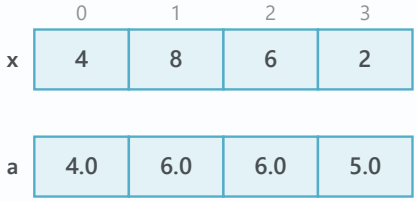
VARIABLES

`j = 3` `total = 20.0`

`a = [4.0, 6.0, 6.0, 5.0]`

`additions = 4`

Code Trace: `prefixAverage2` — running sum



Same result with only **4** additions (prefixAverage1: 10). One loop → **O(n)**.

Step 10 / 10

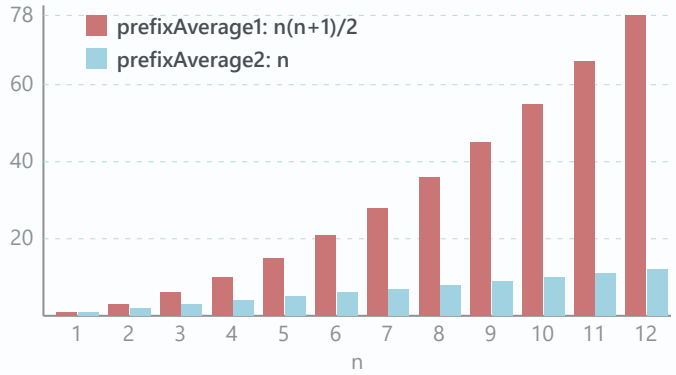
```
1  double[] prefixAverage2(double[] x) {
2    int n = x.length;
3    double[] a = new double[n];
4    double total = 0;
5    for (int j = 0; j < n; j++) {
6      total += x[j];
7      a[j] = total / (j + 1);
8    }
▶    return a;
10 }
```

VARIABLES

`a = [4.0, 6.0, 6.0, 5.0]`

`additions = 4`

Comparing the Two Solutions



n	prefixAverage1	prefixAverage2
10	55	10
100	5,050	100
1,000	500,500	1,000
1,000,000	500,000,500,000	1,000,000

The gain comes from removing **repeated work**: prefixAverage2 remembers the previous sum. For $n = 10^6$, that is 10^6 additions instead of $\sim 5 \cdot 10^{11}$.

Space Complexity

- Measures how much memory an algorithm uses, usually in Big O notation.
- Describes how memory use grows with the input size (usually not counting the input itself: extra space).
- Memory efficiency matters on systems with limited memory.

O(1) · Constant

Example: creating one variable. Memory use does not change with the input size.



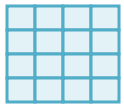
O(n) · Linear

Example: storing every item of an array, like the array a in prefixAverage.



O(n^2) · Quadratic

Example: storing an n×n matrix. Memory grows with the square of n.



Correctness

- A correct algorithm produces the expected output for every valid input in finite time.
- A fast algorithm that gives wrong answers is worthless.

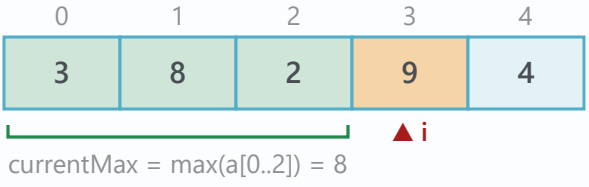
How do we show correctness?

Test cases
Compare outputs with known correct results.

Edge cases
Empty input, a single item, the largest/smallest values.

Loop invariant
A proof via a property that holds at every step.

Counterexample
A single wrong result proves the algorithm incorrect.



Invariant:
before each round, currentMax is the largest in a[0..i−1].

At exit i = n → correct result.

SPACE AND CORRECTNESS

Robustness

- An algorithm's ability to cope with unexpected situations.
- A robust algorithm handles even out-of-range inputs without crashing or giving wrong results.

How do we test robustness?

Invalid input
Coping with inappropriate or unexpected input.
`null, negative size`

Bad data
Coping with corrupt or missing data.
`missing rows, wrong format`

Large data sets
Handling very large inputs.
`10⁸ records`

Overload
How the algorithm behaves under heavy load.
`thousands of simultaneous requests`

SUMMARY

Common Mistakes

“O means worst case”

O denotes an **upper bound**; best and worst case are a separate idea. Any case can be described with O, Ω, or Θ.

“O(1) is always fast”

The constant can be large: an O(1) operation taking 1000 steps per call is slower than an n-step O(n) one while n < 1000.

“Nested loops mean n^2”

If the inner loop advances by `j *= 2`, it is n log n; if it runs a fixed number of times, O(n).

“The log base matters”

$\log_2 n$ and $\log_{10} n$ differ only by a constant factor: both are O(log n).

`(lo + hi) / 2` overflow

For very large arrays the `int` sum overflows; `lo + (hi - lo) / 2` is safe.

Comparing arrays with `a == b`

`==` compares references; use `Arrays.equals(a, b)` for the contents.

LAB · JAVA CODE, STEP BY STEP

Lab: Binary Search and Prefix Averages

Binary search: looking for 23

Input (sorted numbers | target) `2, 5, 8, 12, 16, 23, 38, 56, 72, 91 | 23`

CODE · **step 1 / 13**

```
 1 int binarySearch(int[] a, int target) {
 2     int lo = 0, hi = a.length - 1;
 3     while (lo <= hi) {
 4         int mid = (lo + hi) / 2;
 5         if (a[mid] == target) return mi…
 6         if (a[mid] < target) lo = mid +…
 7         else hi = mid - 1;
 8     }
 9     return -1;
10 }
```

a | 2 | 5 | 8 | 12 | 16 | 23 | 38 | 56 | 72 | 91 |

target = **23** round = **0**

RETURN VALUE –

binarySearch is called with a sorted array a of 10 items and target = 23.

Summary

Topic	Key points
Experimental analysis	time depends on hardware and input; doubling experiment: $T(2n)/T(n) \approx 2^k \rightarrow T(n) \approx c \cdot n^k$
Counting operations	assignment, comparison, arithmetic, array access; hardware-independent, valid for all inputs
O, Ω, Θ	upper, lower, and tight bounds; constant factors and lower-order terms are dropped
Cases	best, average, worst; the worst case for a guarantee, amortized for long sequences of operations
Loop patterns	single loop $O(n)$ · nested $O(n^2)$ · halving $O(\log n)$ · outer n, inner log n → $O(n \log n)$
Choosing an algorithm	prefix averages $O(n^2) \rightarrow O(n)$; binary search in a sorted array $O(\log n)$
Space, correctness	extra memory is measured in Big O too; a fast algorithm that is not correct and robust is worthless

SUMMARY

Operation Costs in Basic Data Structures

Data structure	Access item i	Search	Insert	Delete
Array	O(1)	O(n)	O(n)	O(n)
Linked list	O(n)	O(n)	O(1)†	O(1)†
Stack	top only	O(n)	O(1) push	O(1) pop
Queue	front only	O(n)	O(1) enqueue	O(1) dequeue
Map / set (hash)	—	O(1)*	O(1)*	O(1)*

† If the position (the previous node) is known. * Average case; O(n) in the worst case.

SUMMARY

Sample Problems

1. What is the time complexity of an algorithm that compares every pair of items?
2. Which Θ class does $f(n) = 5n^3 + 2n^2 + 100$ belong to?
3. How many rounds does binary search take in the worst case on a sorted array of $n = 1024$ items?
4. How many times does the loop `for (int i = 1; i < n; i *= 2)` run?
5. What are the best and worst cases of linear search?
6. Algorithm A takes $100n$ steps, algorithm B takes n^2. For which values of n is A faster?

▶ **Show answers**

SUMMARY

Exercises

1. What is the Big O of this code?

```
for (int i = n; i > 1; i /= 2)
  for (int j = 0; j < n; j++) s++;
```

2. Show that $f(n) = 3n^2 + 5n + 2 = O(n^2)$ by giving **c** and $\mathbf{n_0}$.
3. In a doubling experiment, doubling n multiplies the time by ≈ 8. What class is T(n) in?
4. How many additions do prefixAverage1 and prefixAverage2 each make for n = 5?
5. After `for (i = 0; i < n; i++) for (j = i; j < n; j++) s++;`, what is s?
6. An $O(n^2)$ algorithm takes 2 seconds for n = 1000. About how long will it take for n = 4000?

▶ **Answers / hints**

SUMMARY

Further Reading

- Goodrich, Tamassia, and Goldwasser, *Data Structures and Algorithms in Java*, 6th ed., 2014.
- Cormen, Leiserson, Rivest, and Stein, *Introduction to Algorithms*, 4th ed., 2022.
- Sedgewick and Wayne, *Algorithms*, 4th ed., 2011.

The End

Chapter 2: Algorithm Analysis

Next deck: Chapter 3: Arrays · Study questions · Glossary