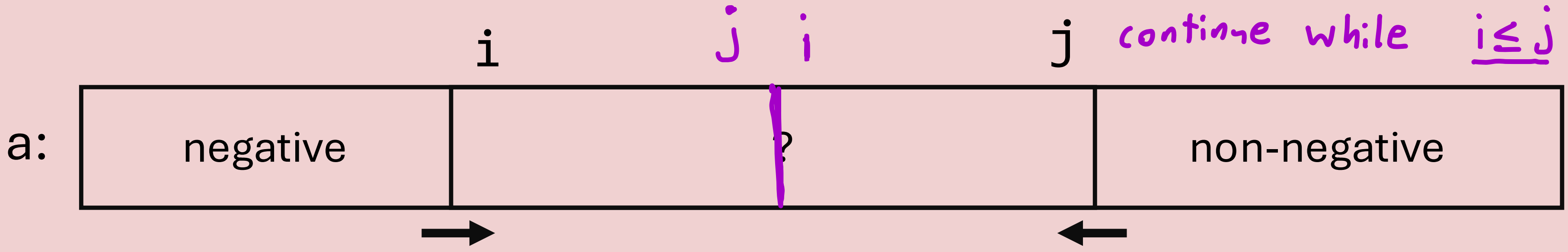
Poll Everywhere

PollEv.com/javabear   text **javabear** to **22333**

Suppose we are writing some "loopy" code with the following "Inv" diagram:

done when i>j

continue while i≤j



What should be the guard on our loop?

`a[i] >= 0` **(A)**	`i < j` **(C)**
`a[i] < a[j]` **(B)**	`i <= j` **(D)**

Lecture 5: Analyzing Complexity

CS 2110

February 3, 2026

Today's Learning Outcomes

25. Explain the benefits of using asymptotic analysis versus performance testing to evaluate the efficiency of a piece of code.

26. Determine the asymptotic time and space complexity of a piece of code involving one or more loops and/or method calls.

27. Determine whether a given (mathematical) function belongs to a given big-O complexity class.

28. Compare and contrast *linear search* and *binary search*.

Code Performance

Better code uses its resources more efficiently:

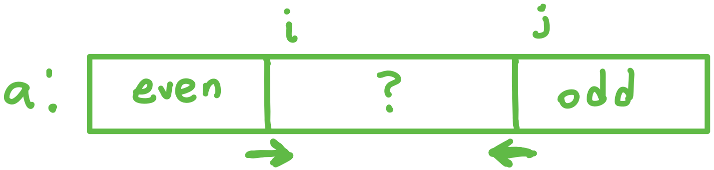
- Time — user experience (lag), use more data, get better answers, time-sensitive domains (flight software)
- Memory Space — embedded systems have space constraints
- Power
- CPU time, network bandwidth, cache space ....

Why does this matter?

To improve our code performance, we need a way to reliably measure time/space usage.

Recall: Our `paritySplit()` Method

```
static int paritySplit(int[] a) {
  int i = 0;      int j = a.length;
  /* Loop invariant: `a[..i)` is even, `a[j..]` is odd */
  while (i < j) {
    if (a[i] % 2 == 0) {
      i++;
    } else {
      swap(a,i,j-1);
      j--;
    }
  }
  return j;
}
```




```
static void swap(int[] a, int x, int y) {
  int temp = a[x];
  a[x] = a[y];
  a[y] = temp;
}
```

How should measure the runtime of this code?

Measuring Runtime

Option 1: Time its execution

```
int[] a = ... ; // initialize input array

long start = System.nanoTime();
int i = paritySplit(a);
long end = System.nanoTime();
System.out.println("paritySplit() ran in " + (end - start) / 1e6 + " ms.");
```

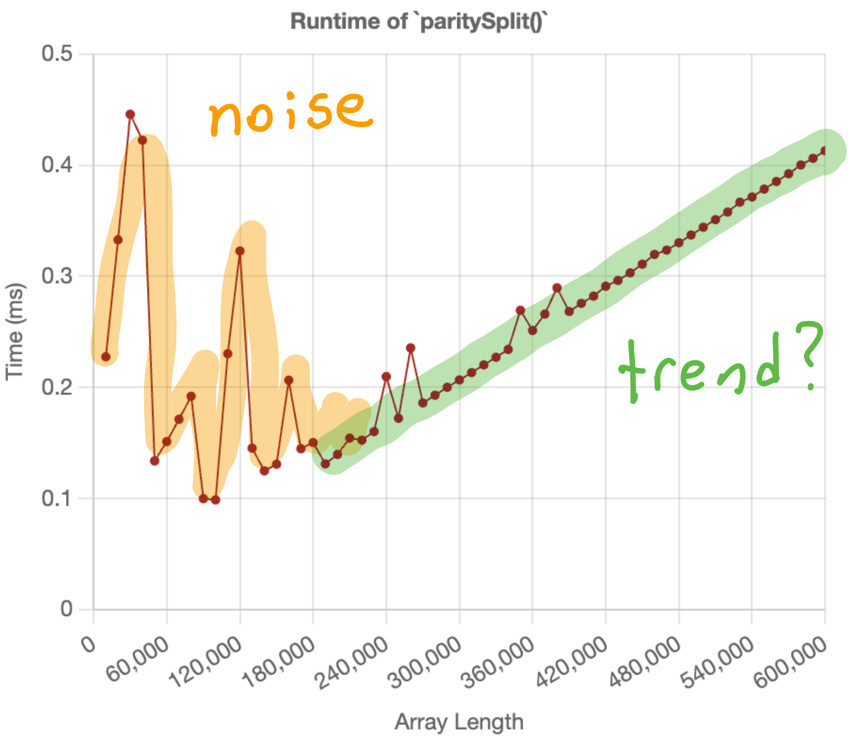
Output for 1000 elements:
- 0.285125 ms
- 0.222625 ms
- 0.169167 ms

Issues:
- What about other inputs?
  - Bigger inputs will take longer
  - Different inputs (of same size) might take more/less time
- Other computers may be faster/slower.

Measuring Runtime

Option 2: Time its execution across many input sizes

- Gives a better picture of how runtime depends on input size
- Picture is still a bit noisy
- Can we extract away noise and unneeded details to leave only trend summary?



Runtime vs. Time Complexity

The "wall-clock" runtime of a method is how long it takes to execute.

Its time complexity T is the number of basic operations performed during its execution, expressed as a function of its input parameter sizes.

Basic ops: - Read value of var, write value to var, math operation, etc. (soon, we'll see details aren't crucial)

Today: Methods involving arrays, $N =$ array length

runtimes will be functions of N, $T(N)$

Counting Basic Operations

```
static void swap(int[] a, int x, int y) {
  int temp = a[x];
  a[x] = a[y];
  a[y] = temp;
}
```

- read x
- read a[x]
- write to temp

3 ops

- read y
- read a[y]
- read x
- write to a[x]

4 ops

- read temp
- read y
- write to a[y]

3 ops

For swap() $T(N) = 10$

does not depend on input size or input values

Poll Everywhere

PollEv.com/javabear   text **javabear** to 22333

```java
static int paritySplit(int[] a) {
  int i = 0;    int j = a.length;
  /* Loop invariant: `a[..i)` is even, `a[j..]` is odd */
  while (i < j) {
    if (a[i] % 2 == 0) {
      i++;
    } else {
      swap(a,i,j-1);
      j--;
    }
  }
  return j;
}
```

even less work than odd

Which `paritySplit()` input will execute more basic operations?

- `{1, 2, 3, 5, 7}` (A)
- `{2, 4, 6, 7, 8}` (B)
- They are the same (C)

Best-Case vs. Worst-Case Inputs

Sometimes, # operations depends not only on input size, but also values (e.g., to determine if "if" body is executed)

Typically consider worst-case input, whose values force us down most complex execution path

vs. best-case input, whose values allow minimal # operations

CAUTION!!

Common misconception: worst-case ≠ bigger
best-case ≠ smaller

These are descriptors of inputs of some pre-determined size.

Counting Basic Operations (Worst-Case)

Code	# ops	# times executed	total # ops
`static int paritySplit(int[] a) {`			
`int i = 0;     int j = a.length;` ($\underbrace{\texttt{a.length}}_{N}$)	$1+2=3$	1	3
`while (i < j) {`	3	$N+1$	$3N+3$
`if (a[i] % 2 == 0) {`	4	N	$4N$
`i++;`	3	0	0
`} else {`			
`swap(a,i,j-1);`	$1+2+4+10$	N	$17N$
`j--; }}`	3	N	$3N$
`return j; }`	1	1	1

best-case will hit if branch

worst-case will hit else branch

$T(N) = 27N + 7$

Asymptotic (Big-O) Notation

$T(N) = 27N + 7$ is a bit too specific

- Variations in hardware can change 27 (drop leading coefficients)
- For large N, "low-order" term 7 is negligible (drop)

We write $\boxed{T(N) = O(N)}$ to summarize runtime complexity grows linearly with input size.

Def. We say runtime $T(N)$ is $O(g(N))$ for some function g if

$$\lim_{N \to \infty} \frac{T(N)}{g(N)} < \infty$$

"for large inputs, the growth of $T(N)$ does not exceed $g(N)$, up to a constant factor"

For our purposes, enough to place $T(N)$ appropriately in hierarchy:

The Runtime Hierarchy

Better Performance

Worse Performance

Complexity Class	Name
$O(1)$	Constant Time
$O(\log N)$	Logarithmic Time
$O(N)$	Linear Time
$O(N \log N)$	Linearithmic Time
$O(N^2)$	Quadratic Time
$O(N^3)$	Cubic Time
$O(2^N)$	Exponential Time

Really Good

swap()

binarySearch()

paritySplit()

linearSearch()

hasDuplicates()

Really Bad!

Another Method: `hasDuplicates()`

```
/** Returns whether `a` contains duplicate entries,
  *  distinct indices `i != j` with `a[i] == a[j]`.*/
static boolean hasDuplicates(int[] a) {
    for (int i = a.length - 1; i >= 0; i--) {
        for (int j = 0; j < i; j++) {
            if (a[i] == a[j]) {
                return true;
    }}}
    return false;
}
```

Poll Everywhere

PollEv.com/javabear text **javabear** to **22333**

Which of the following is a **best-case** input of length 6 to `hasDuplicates()`?

```
/** Returns whether `a` contains duplicate entries,
 * distinct indices `i != j` with `a[i] == a[j]`.*/
static boolean hasDuplicates(int[] a) {
  for (int i = a.length - 1; i >= 0; i--) {   ← i
    for (int j = 0; j < i; j++) {   j →
      if (a[i] == a[j]) {
        return true;
  }}}
  return false;
}
```

- { 1, 2, 3, 4, 5, 6 } **(A)**
- { 1, 1, 2, 3, 4, 5 } **(B)**
- { 1, 2, 3, 4, 5, 1 } **(C)** — first two compared
- { 1, 2, 3, 4, 5, 5 } **(D)**

Asymptotic Analysis (Worst-Case)

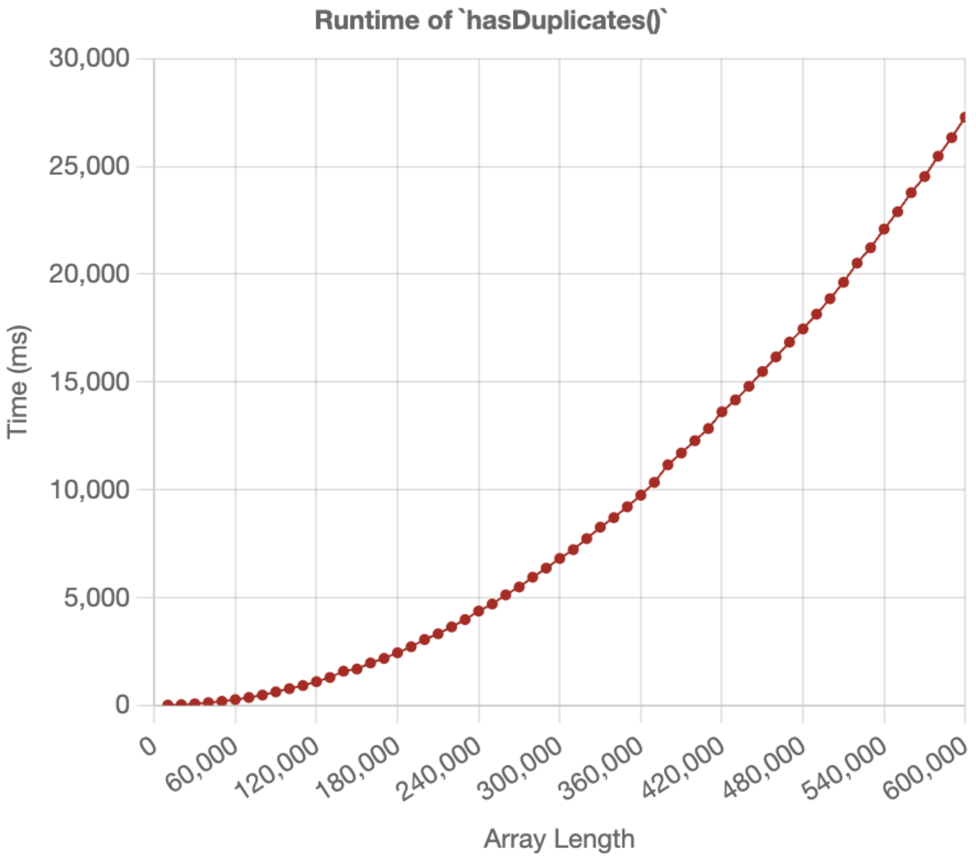
```
static boolean hasDuplicates(int[] a) {
  for (int i = a.length - 1; i >= 0; i--) {
    for (int j = 0; j < i; j++) {
      if (a[i] == a[j]) {
        return true;
}}}
  return false;
}
```

← not used by worst-case input

complexity	# executions	total complexity
$O(1)$	$O(N)$	$O(N)$
$O(1)$	$\sum_{i=0}^{N-1}(i+1) = O(N^2)$	$O(N^2)$
$O(1)$	$O(N^2)$	$O(N^2)$
$O(1)$	$O(1)$	$O(1)$

$$\boxed{O(N^2)}$$

Visualizing `hasDuplicates()` Runtime



Poll Everywhere

PollEv.com/javabear   text **javabear** to **22333**

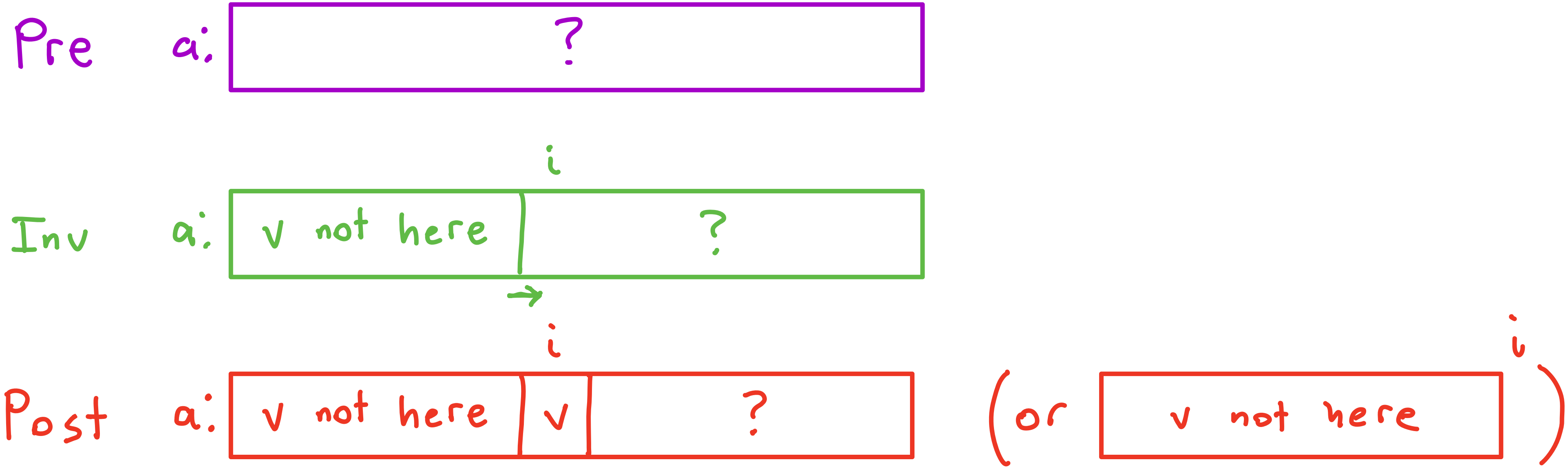
`hasDuplicates()` ran in 10 seconds for an array with 360,000 elements.

How long should it take to run on an array with 720,000 elements? $O(N^2)$ time complexity means

20 seconds	doubling input size $\times 2$	(A)
40 seconds	quadruples runtime $\times 2^2$	(B)
100 seconds		(C)

Linear Search

```java
/** Returns the smallest index `i` such that `a[i] == v` or returns
 * `a.length` if `a` does not contain `v`. */
static int linearSearch(int[] a, int v) { ... }
```



Coding Demo: Linear Search

Runtime Analysis

```java
static int linearSearch(int[] a, int v) {
  int i = 0;
  /* Loop inv: `a[..i)` does not contain `v`. */
  while (i < a.length) {
    if (a[i] == v) {
      return i;
    }
    i++;
  }
  return a.length;
}
```

– early return not worst-case

	complexity	# executions	total complexity
`int i = 0;`	O(1)	O(1)	O(1)
`while (i < a.length)`	O(1)	O(N)	O(N)
`if (a[i] == v)`	O(1)	O(N)	O(N)
`i++;`	O(1)	O(N)	O(N)
`return a.length;`	O(1)	O(1)	O(1)
			$\boxed{O(N)}$

Binary Search

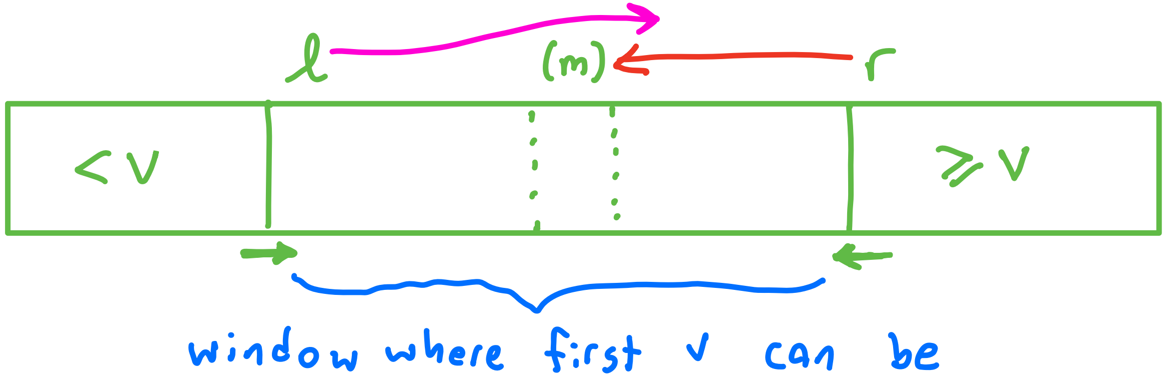
If array a is <u>sorted</u>, can we find the first index of v faster?

Idea: Inspect the middle element a[m]

- a[m] ≥ v: don't need to consider a(m..]
- a[m] < v: don't need to consider a[..m]

* like searching in a (paper) dictionary

Inv a:



Coding Demo: Binary Search