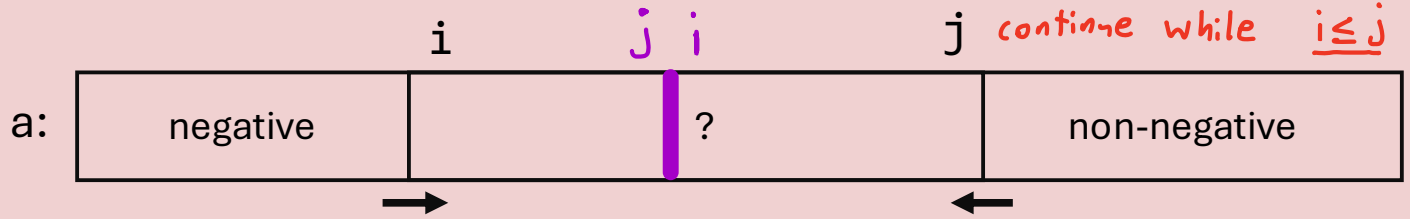
Poll Everywhere

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

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

done when i>j, negate

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

September 8, 2026

Today's Learning Outcomes

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

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

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

26. Compare and contrast *linear search* and *binary search*, discussing aspects such as time/space complexity and pre-conditions.

Code Performance

In lecture 1, we claimed that "better" code uses its resources more efficiently. Why does this matter?

- **Time:** user experience (lag), use more data, get better answers, time-sensitive domains (flight software)
- **(Memory) Space:** embedded systems have space constraints
- Many other factors (power, CPU time, network bandwidth, cache space, ...) beyond our scope

To improve our code performance, we need a way to reliably "measure" and compare time/space usage of different approaches.

Recall: Our `paritySplit()` Method

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

static int paritySplit(int[] a) {
  int i = 0; int j = a.length;
  /* Loop Inv: `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;
}
```

How should we measure the runtime of this code?

Measuring Runtime

Option 1: Time its execution

```
1  int[] a = ... ; // initialize input array
2
3  long start = System.nanoTime();
4  int i = paritySplit(a);
5  long end = System.nanoTime();
6  System.out.println("paritySplit() ran in " + (end - start) / 1e6 + " ms.");
```

Output for 10,000 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.

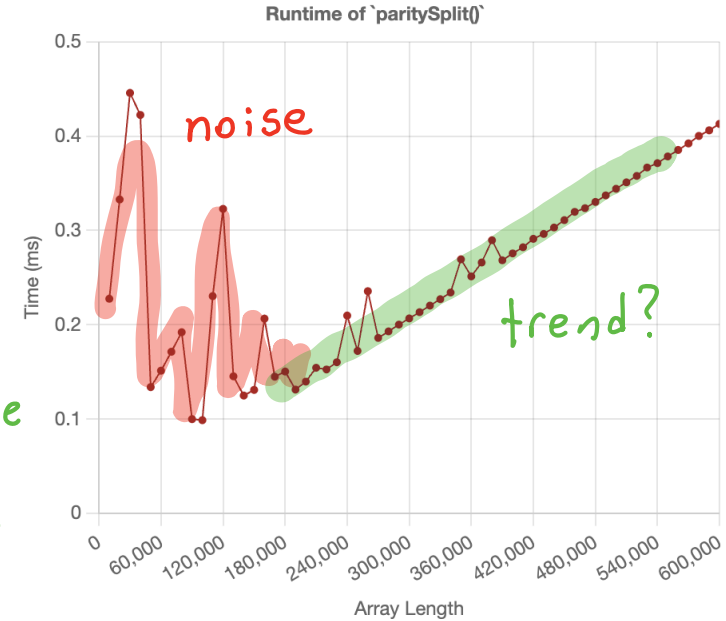
Measuring Runtime

Option 2: Time its execution across many input sizes

This offers a better picture of how the runtime scales with parameter sizes

- 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** is the number of *basic operations* performed during its execution, expressed as a *function of its 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

```
1  static void swap(int[] a, int x, int y) {
2    int temp = a[x];
3    a[x] = a[y];
4    a[y] = temp;
5  }
```

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

does not depend on input size (or input values)

Line 2:

1. read x
2. read a[x]
3. write to temp

Line 3:

1. read y
2. read a[y]
3. read x
4. write to a[x]

Line 4:

1. read temp
2. read y
3. write to a[y]

Poll Everywhere

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

```
1  static int paritySplit(int[] a) {
2    int i = 0; int j = a.length;
3    while (i < j) {
4      if (a[i] % 2 == 0) {
5        i++;
6      } else {
7        swap(a,i,j-1);
8        j--;
9      }
10   }
11   return j;
12 }
```

even case does less work than odd case

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, the number of operations depends not only on the parameter *sizes*, but also their *values* (e.g., to determine if `if` or `else` block 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)

```
1  static int paritySplit(int[] a) {
2    int i = 0; int j = a.length;
3    while (i < j) {
4      if (a[i] % 2 == 0) {
5        i++;
6      } else {
7        swap(a,i,j-1);
8        j--;
9      }
10   }
11   return j;
12 }
```

N

best-case uses if branch

worst-case uses else branch

Line #	# Basic Ops	# Times Run	Total # Ops
2	1+3 = 4	1	4
3	3	N+1	3N+3
4	4	N	4N
5	3	0	0
7	4 (eval args) +3 (copy params) +10 (swap())	N	17N
8	3	N	3N
11	1	1	1

T(N) = 27N + 8

Asymptotic (Big-O) Notation

$T(N) = 27N + 8$ is a bit too specific

- Variations in hardware can change 27 (drop leading coefficients)
- For large N, "low-order" term 8 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

swap()

binarySearch()

Really Good

partitionSplit()

linearSearch()

hasDuplicates()

Really Bad!

Another Method: hasDuplicates()

```java
/** 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()`?

```
1  static boolean hasDuplicates(int[] a) {
2    for (int i = a.length - 1; i >= 0; i--) {
3      for (int j = 0; j < i; j++) {
4        if (a[i] == a[j]) {
5          return true;
6    }}}
7    return false;
8  }
```

i ←

j →

{ 1, 2, 3, 4, 5, 6 } **(A)**

{ 1, 1, 2, 3, 4, 5 } **(B)**

first two compared

{ 1, 2, 3, 4, 5, 1 } **(C)**

{ 1, 2, 3, 4, 5, 5 } **(D)**

(Worst-Case) Asymptotic Analysis

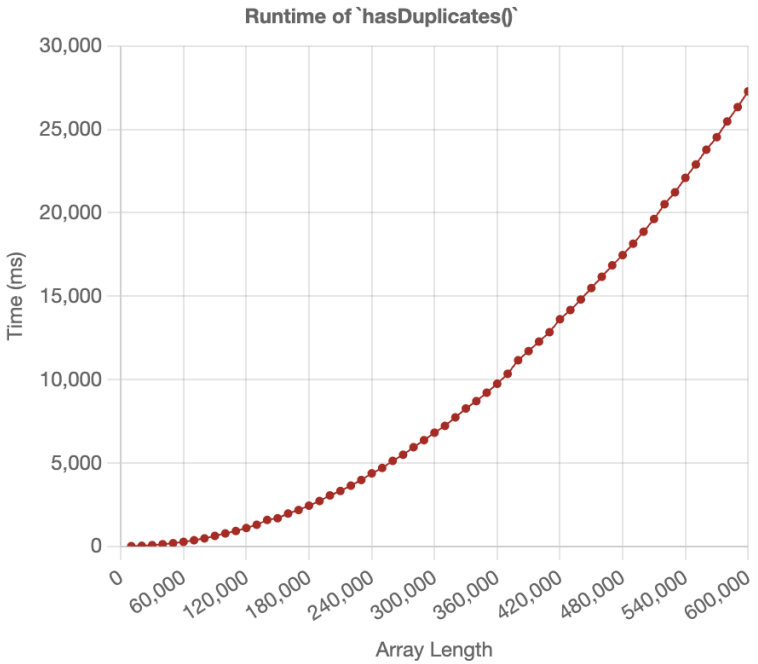
```
1  static boolean hasDuplicates(int[] a) {
2    for (int i = a.length - 1; i >= 0; i--) {
3      for (int j = 0; j < i; j++) {
4        if (a[i] == a[j]) {
5          return true;
6    }}}
7    return false;
8  }
```

never used in worst case

Line #	Per-run Complexity	# Times Run	Total Complexity
2 (guard)	$O(1)$	$O(N)$	$O(N)$
3 (guard)	$O(1)$	$\sum_{i=0}^{n-1} i = O(N^2)$	$O(N^2)$
4	$O(1)$	$O(N^2)$	$O(N^2)$
5	$O(1)$	$O(1)$	$O(1)$

$O(N^2)$

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 doubling input size $\times 2$ quadruples runtime $\times 2^2$

20 seconds	(A)
40 seconds	(B)
100 seconds	(C)

Linear Search

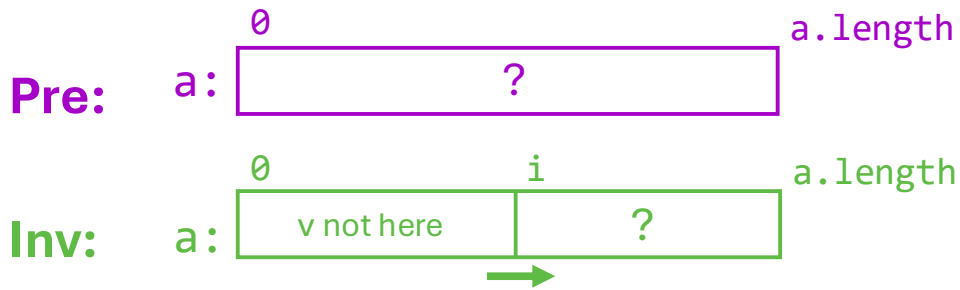
```
/** Returns the smallest index `i` such that `a[i] == v`.
 *  Returns `a.length` if `a` does not contain `v`. */
static int linearSearch(int[] a, int v) { ... }
```

Linear Search

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

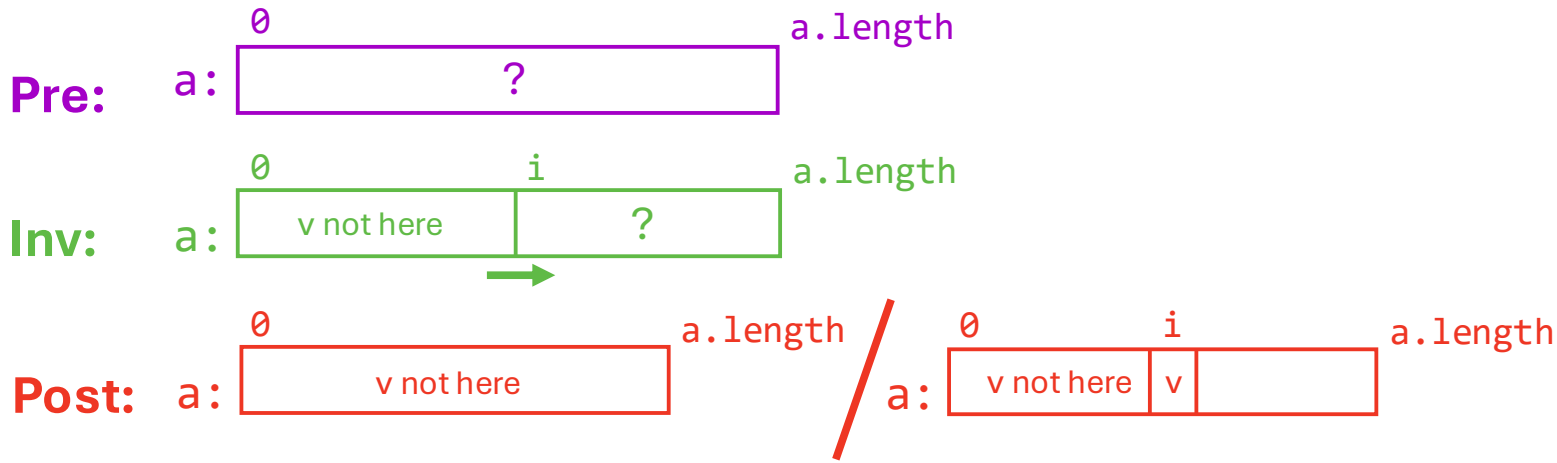
Linear Search

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



Linear Search

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



Coding Demo: Linear Search

Linear Search Runtime Analysis

```
1  static int linearSearch(int[] a, int v) {
2    int i = 0;
3    /* Loop inv: `v` not in `a[..i)`. */
4    while (i < a.length) {
5      if (a[i] == v) {
6        return i;
7      }
8      i++;
9    }
10   return a.length;
11 }
```

worst-case: v not in a

Line #	Complexity	# Times Run	Total Complexity
2	O(1)	O(1)	O(1)
4	O(1)	O(N)	O(N)
5	O(1)	O(N)	O(N)
6	O(1)	O	O
8	O(1)	O(N)	O(N)
10	O(1)	O(1)	O(1)
			O(N)

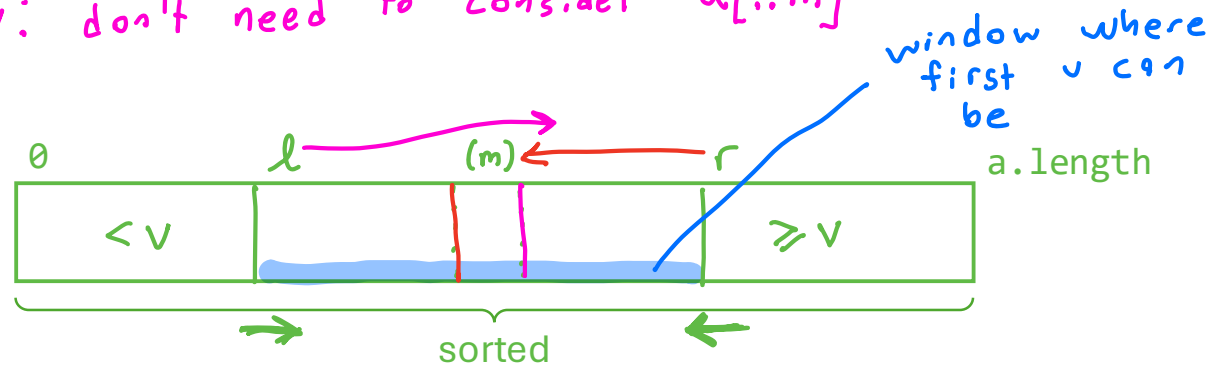
Binary Search

If an array a is **sorted**, how can we find the first occurrence 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



Coding Demo: Binary Search