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:**

- **(Memory) Space:**

- 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

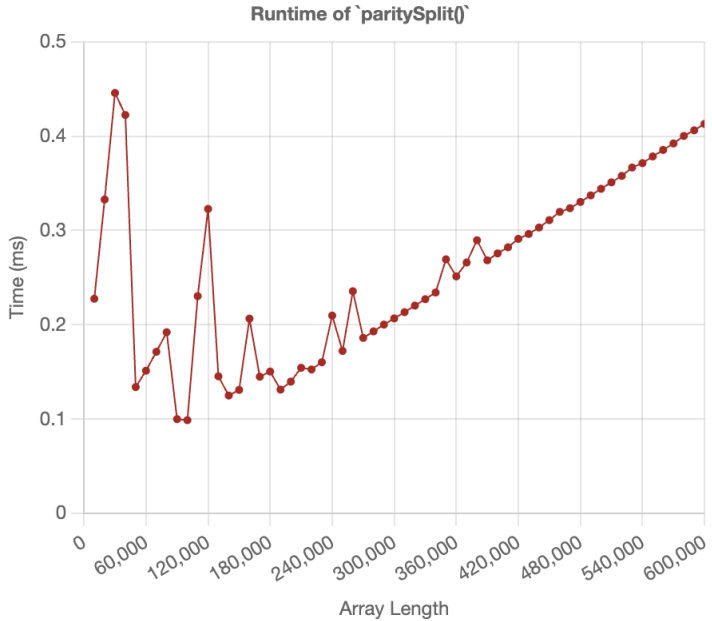
```
1  int[] a = ... ; // initialize input array
2
3  long start = System.nanoTime();
4  int i = paritySplit(a);
5  long end = System.nanoTime();
6  System.out.println("paritySplit() ran in " + (end - start) / 1e6 + " ms.");
```

Issues:

Measuring Runtime

Option 2: Time its execution across many input sizes

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



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*.

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  }
```

Line 2:

Line 3:

Line 4:

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

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

Line #	# Basic Ops	# Times Run	Total # Ops
2			
3			
4			
5			
7			
8			
11			

Asymptotic (Big-O) Notation

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

Really Bad!

Another Method: hasDuplicates()

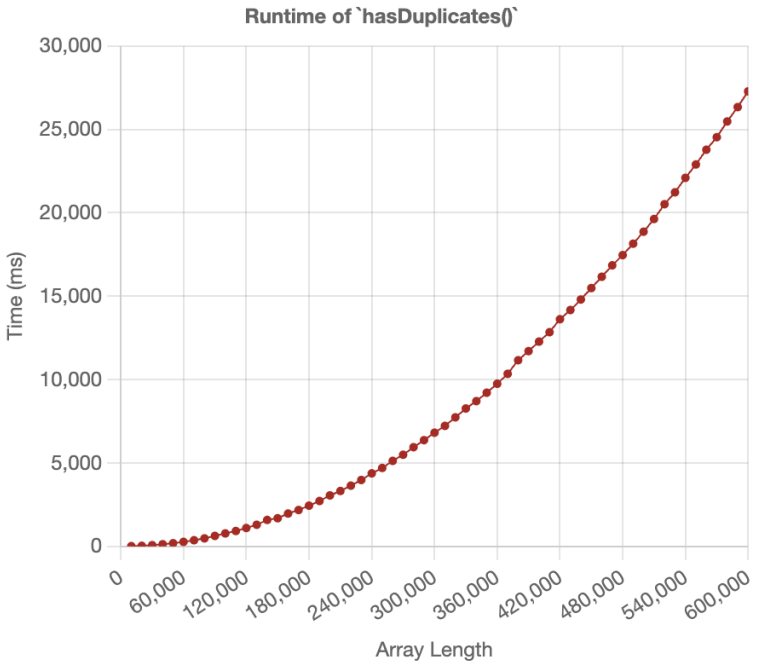
```java
/** Returns whether `a` contains duplicate entries,
 *  distinct indices `i != j` with `a[i] == a[j]`. */
static boolean hasDuplicates(int[] a) {
  for (int i = a.length - 1; i >= 0; i--) {
    for (int j = 0; j < i; j++) {
      if (a[i] == a[j]) {
        return true;
  }}}
  return false;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

(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  }
```

Line #	Per-run Complexity	# Times Run	Total Complexity
2 (guard)			
3 (guard)			
4			
5			

Visualizing `hasDuplicates()` Runtime



Linear Search

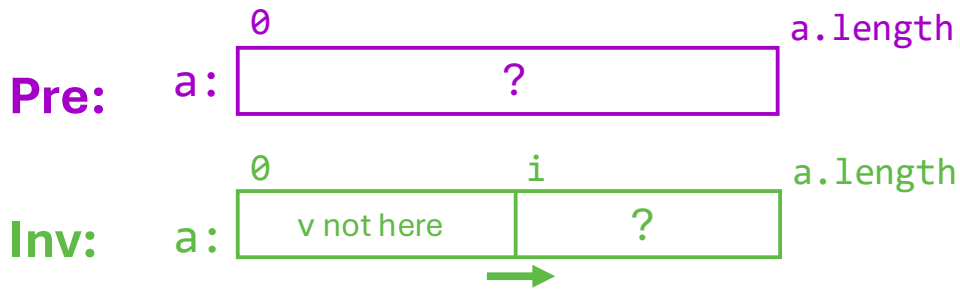
```
/** Returns the smallest index `i` such that `a[i] == v`.
 *  Returns `a.length` if `a` does not contain `v`. */
static int linearSearch(int[] a, int v) { ... }
```

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) { ... }
```

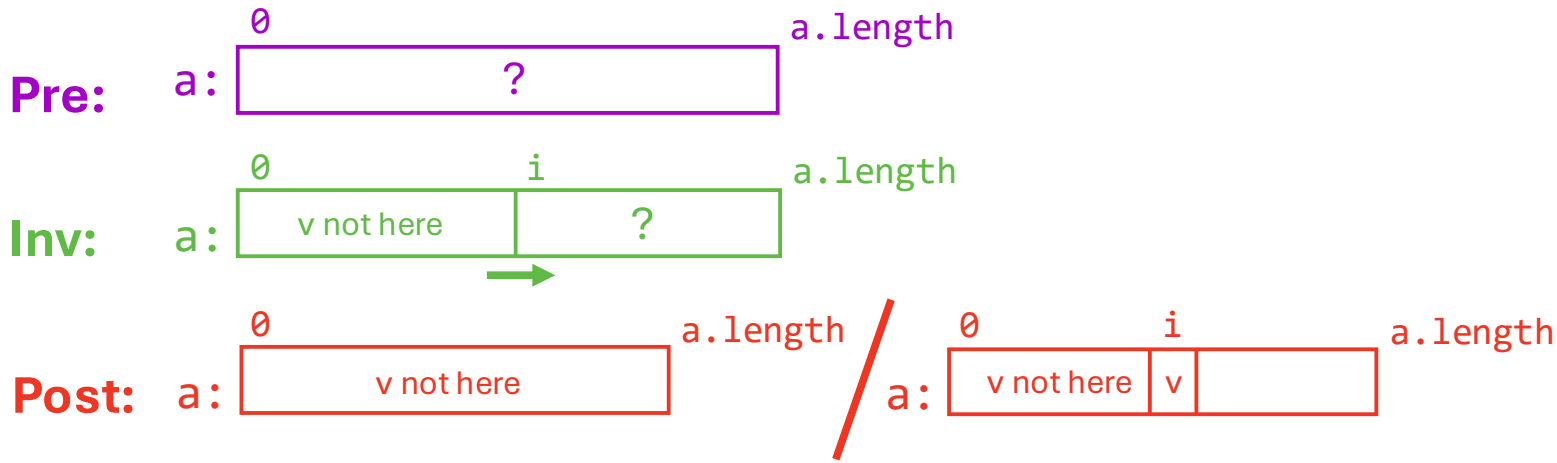
Linear Search

```
/** 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

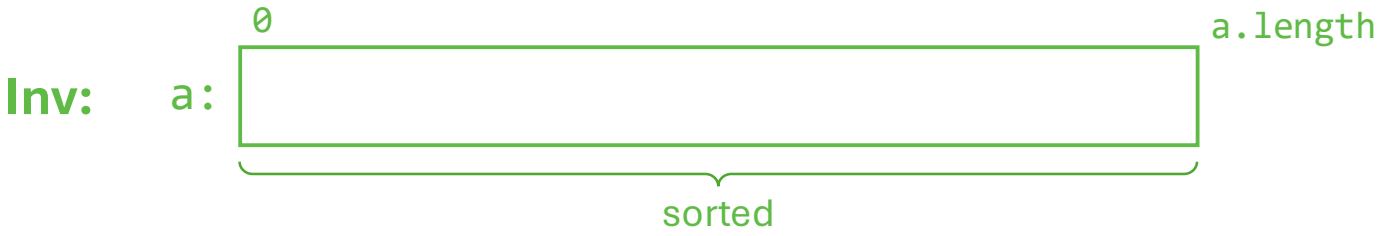
Linear Search Runtime Analysis

```
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 }
```

Line #	Complexity	# Times Run	Total Complexity
2			
4			
5			
6			
8			
10			

Binary Search

If an array a is **sorted**, how can we find the first occurrence of v faster?



Coding Demo: Binary Search

Binary Search Runtime Analysis

```
1  static int binarySearch(int[] a, int v) {
2    int l = 0;   int r = a.length;
3    /* Loop inv: `a[..l) < v`, `a[r..] >= v` */
4    while (l < r) {
5      int m = l + (r - l) / 2;
6      if (a[m] < v) {  l = m + 1;
7      } else {  r = m; }
8    }
9    return r;
10 }
```