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 }
```

Space Complexity

= the maximum amount of memory needed *at any point in time* during a method's execution (beyond the memory used to store its parameters).

Lecture 6: Recursion

CS 2110

September 10, 2026

Today's Learning Outcomes

10. Develop recursive methods in Java from their specifications.

27. Determine the number of recursive calls and the maximum depth of the call stack of a recursive method and use these to compute its time and space complexities.

Recursive Methods

A method is **recursive** if it can be invoked from within its own definition.

Recursion offers another way, in addition to loops, to achieve conditional repetition of code.

Loops

Loop Vars

Loop Body

Loop Invariant

Loop Guard

Recursion

When we write a recursive method, we're both its implementer and its client!

Computing Factorials

The **factorial** of an integer n ≥ 0 is the product of all positive integers ≤ n.

```
1  /** Returns `n!`. Requires `0 <= n <= 12` */
2  static int factorial(int n) {
3    int product = 1;
4    /* loop inv:                                */
5    for (int i = 1; i <= n; i++) {
6      product *= i;
7    }
8    return product;
9  }
```

A Recursive Implementation

Base Case:

Recursive Case:

Coding Demo: Recursive factorial( )

Visualizing Recursion: `factorial(4)`

Runtime Stack

```
1  static int factorial(int n) {
2    assert 0 <= n && n <= 12;
3    if (n <= 1) { return 1; }
4    int f = factorial(n - 1);
5    return n * f;
6  }
```

Time Complexity of Recursive Code

We must account for the operations performed across **all** the recursive calls (not just the outermost one).

To do this, we'll determine 2 things:

1.

2.

Space Complexity of Recursive Code

Again, there are two things to consider:

1.

2.

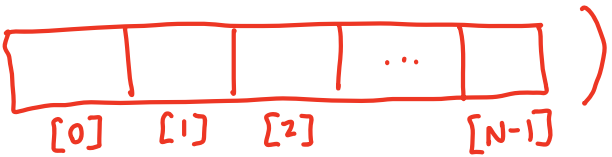
The additional space from all stack frames often makes recursion less efficient than iteration.

Recursion on Arrays

```
/** Returns the maximum value in `nums`. Requires that `nums` is non-empty. */
static double maxValue(double[] nums) { ... }
```

Base Case:

Recursive Case:

max(nums 

)

Coding Demo: Recursive `maxValue()`

Array Views

Constructing a new, smaller array to pass into a recursive call is expensive!

Instead, we'd like to use only one array (passing an alias reference) and "tell" the recursive call to "only look at certain entries".

Coding Demo: `maxValue()`, Take 2

Another Recursive Method on Arrays

```
/** Returns true if there is a subset of entries from `coins` whose sum
 *  is equal to `total`, otherwise returns `false`. */
static boolean canMakeChange(int total, int[] coins) { ... }
```

Ex. canMakeChange(30, 

) =

canMakeChange(28, 

) =

Designing the Recursion

```
/** Returns true if there is a subset of entries from `coins` whose sum
 *  is equal to `total`, otherwise returns `false`. */
static boolean canMakeChange(int total, int[] coins) { ... }
```

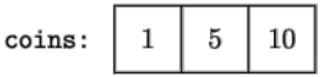
Base Case:

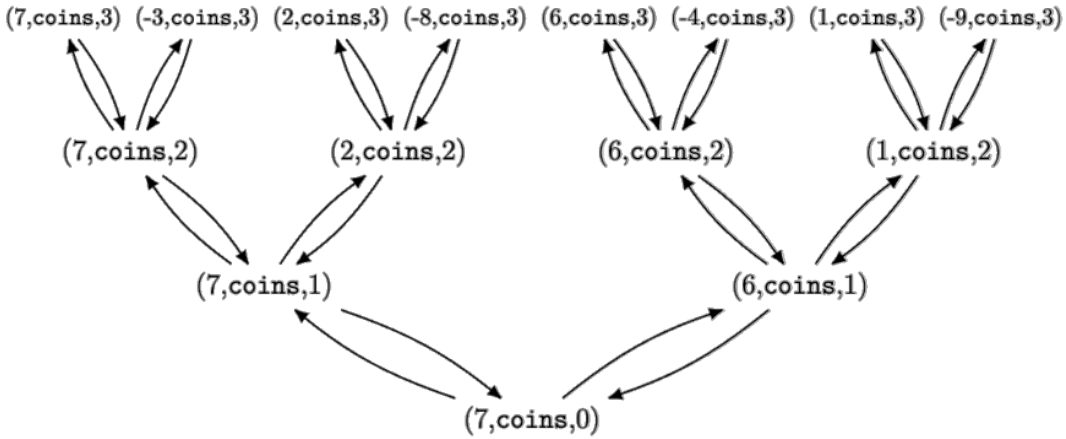
Recursive Cases:

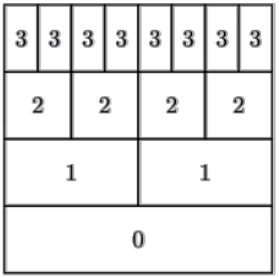
canMakeChange(30, | 1 | 5 | 5 | 10 | 25 |)

Coding Demo: `canMakeChange()`

Visualizing the Call Structure







Time and Space Complexity