

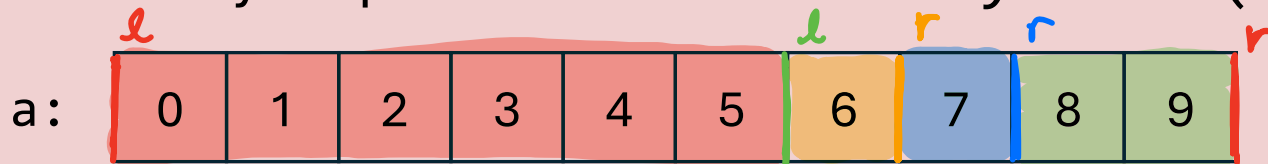
Poll Everywhere

PollEv.com/2110fa25

text 2110fa25 to 22333



How many loop iterations does `binarySearch(a, 7)` run?



```
static int binarySearch(int[] a, int v) {  
    int l = 0; int r = a.length;  
    /* Loop invariant: `a[..l) < v`, `a[r..] >= v` */  
    while (l < r) {  
        int m = l + (r - l) / 2;  
        if (a[m] < v) { l = m + 1;  
        } else { r = m;  
        }  
    }  
    return r;  
}
```

l	r	m
0	10	5
6	10	8
6	8	7
6	7	6
7	7	

3

(A)

4

(B)

5

(C)

6

(D)

Runtime Analysis

```
static int binarySearch(int[] a, int v) {  
    int l = 0; int r = a.length;    }  $O(1)$   
    /* Loop invariant: `a[..l) < v`, `a[r..] >= v` */  
    while (l < r) {  
        int m = l + (r - l) / 2;  
        if (a[m] < v) { l = m + 1; }  
        else { r = m; }  
    }  
    return r;    }  $O(1)$   
}
```

$O(1)$ work

- In every loop iteration,
the "window" $[l, r)$ shrinks
to at most half its old size.

- Stop when $r - l < 1$

$$N \cdot \left(\frac{1}{2}\right)^{\text{\# iterations}} \geq 1$$

$$\text{so \# iterations} \leq \log_2(N) \\ = O(\log N)$$

$O(\log N)$ runtime

Space Complexity

Measures the total amount of memory allocated at one time during the execution of a method (beyond the space for its parameters).

- "scratch space for computation" these are the caller's responsibility
- can reuse space later (unlike time)
- all methods we saw last lecture had $O(1)$ space complexity
- gets more subtle when we analyze recursive methods



Lecture 6: Recursion

CS 2110

September 11, 2025

Today's Learning Outcomes

- 10. Develop recursive methods in Java given their specifications.
- 29. 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.

- Another way, in addition to loops, to achieve conditional repetition of code

Loops

Loop Vars
Loop Body
Loop Guard
Loop Invariant

Recursion

Parameters
Method Body
Base case(s)
Method Spec

"simplest" inputs whose result is computed directly

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

Computing Factorials

Def: The factorial of an int $n \geq 0$ is the product of all positive ints $\leq n$.

$$\text{Ex. } 4! = 1 \cdot 2 \cdot 3 \cdot 4 = 24$$

$$6! = 1 \cdot 2 \cdot 3 \cdot 4 \cdot 5 \cdot 6 = 720$$

$$1! = 1$$

$$0! = 1 \quad (\text{mult. identity})$$

```
/** Returns `n!`. Requires `0 <= n <= 12` */  
static int factorial(int n) {  
    int product = 1;  
    /* loop invariant: product = (i-1)! */  
    for (int i = 1; i <= n; i++) {  
        product *= i;  
    }  
    return product;  
}
```

A Recursive Implementation


Base Case: $0! = 1$ (also $1! = 1$) so
factorial(n) can return 1 if $n \leq 1$

Recursive Case:

ASK "How can calling factorial() on a smaller input help compute it for a larger input?"

$$5! = \underbrace{1 \cdot 2 \cdot 3 \cdot 4}_{4!} \cdot 5$$

More generally, $n! = (n-1)! \cdot n$

 recursive call



Coding Demo: Recursive factorial()



Visualizing Recursion

Suppose we call `factorial(4)`

`factorial(1)`

`n: int` 1

`f: int`

`factorial(2)`

`n: int` 2

`f: int` 1

`factorial(3)`

`n: int` 3

`f: int` 2

`factorial(4)`

`n: int` 4

`f: int` 6

* Ordinarily, we don't draw call frames after they return, but this doesn't translate on a slide. See animations in notes.

returns
24

```
/** Returns `n!`
```

```
 * Requires `0 <= n <= 12` */
```

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

Time Complexity of Recursive Code

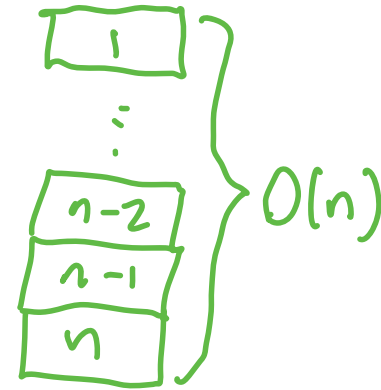
We must account for the operations performed across all of the recursive calls.

Determine 2 things:

- "Non-Recursive" work done in each call (is a function of the parameters)

For factorial() this was $O(1)$

- Recursive call structure (to know how many of the above to add up)



Space Complexity of Recursive Code

Again, we account for two things:

1. Additional space of objects constructed by any calls (none for `factorial()`)
2. Space taken up by call frames

Typically $O(1) * \text{maximum \# of call frames present at any point of the execution}$

"recursive depth"

For `factorial()`: $0 + O(1) * n = O(n)$

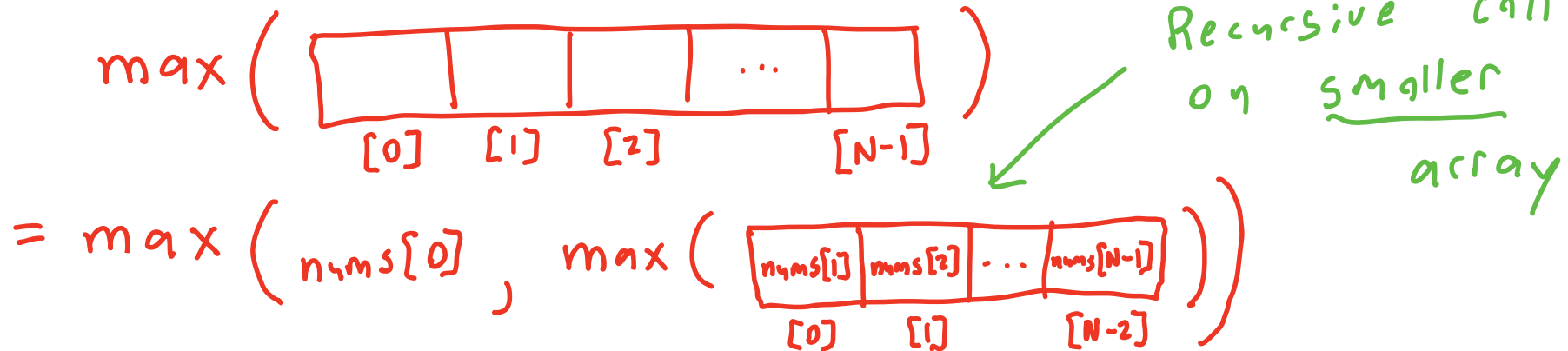
← additional space + time to make all stack frames makes recursion less efficient than iteration

Recursion on Arrays

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

Base Case: If `nums.length` is 1, then the only value is the max value

Recursive Case:





Coding Demo: Recursive maxValue()



Poll Everywhere

PollEv.com/2110fa25

text 2110fa25 to 22333



What are the time and space complexities of our recursive `maxValue()` implementation?

Each of $O(N)$ recursive calls used $O(N)$ space and did $O(N)$ work

Time Complexity: $O(N)$ Space Complexity: $O(N)$ **(A)**

Time Complexity: $O(N)$ Space Complexity: $O(N^2)$ **(B)**

Time Complexity: $O(N^2)$ Space Complexity: $O(N)$ **(C)**

Time Complexity: $O(N^2)$ Space Complexity: $O(N^2)$ **(D)**

Array Views

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

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

Solution: Use additional parameters to define array view

```
maxValueRecursive(double[] nums, int begin) { }
```




Coding Demo: `maxValue()`, Take 2



Poll Everywhere

PollEv.com/2110fa25

text 2110fa25 to 22333



What are the time and space complexities of our new `maxValue()` implementation?

Each of $O(N)$ recursive calls used $O(1)$ space and did $O(1)$ work

Time Complexity: $O(N)$ Space Complexity: $O(N)$

(A)

Time Complexity: $O(N)$ Space Complexity: $O(N^2)$

(B)

Time Complexity: $O(N^2)$ Space Complexity: $O(N)$

(C)

Time Complexity: $O(N^2)$ Space Complexity: $O(N^2)$

(D)

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. $\text{canMakeChange}(16, [1, 1, 5, 10, 25]) = \text{true}$

$\text{canMakeChange}(8, [1, 1, 5, 10, 25]) = \text{false}$

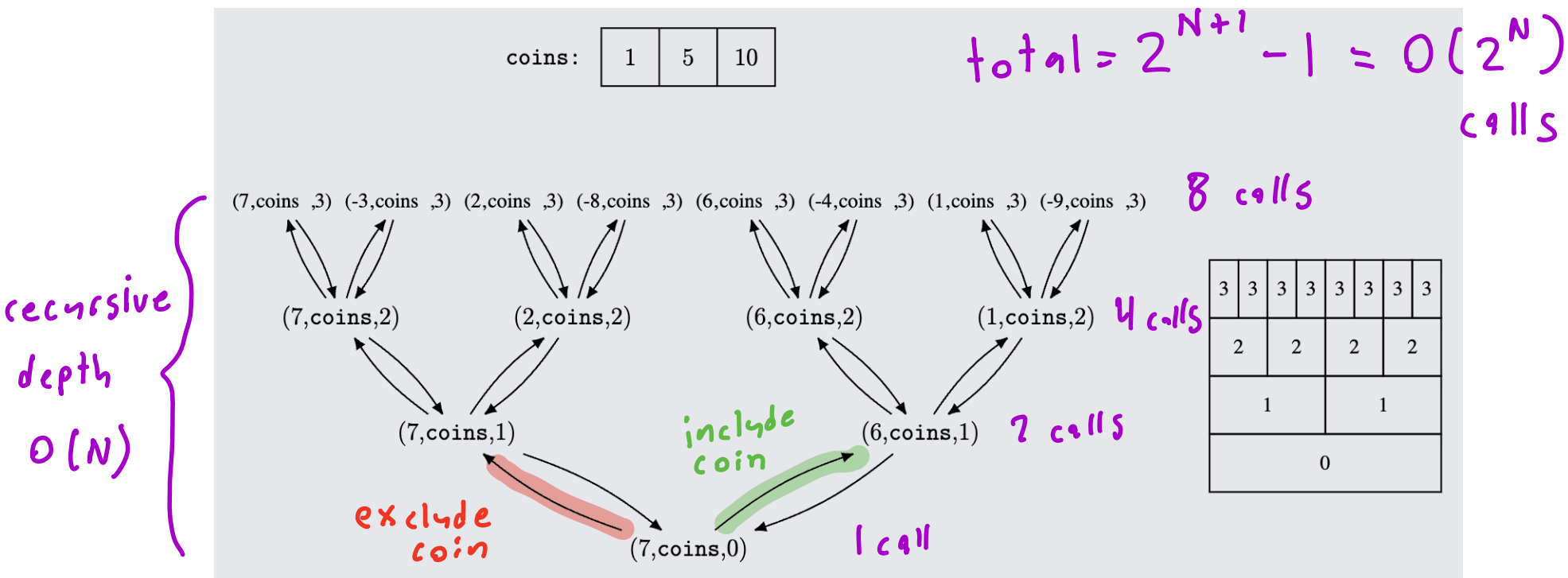
* Think about base cases + recursive calls *



Coding Demo: canMakeChange()



Visualizing the Call Structure



Time and Space Complexity

Each execution of `canMakeChange Recursive()` does $O(1)$ non-recursive work and allocates $O(1)$ memory

time complexity = $O(1) * \# \text{ calls} = O(2^N)$ exponential time
⋮

space complexity = $O(1) * \text{recursive depth} = O(N)$