

# Building Java Programs

Chapter 13

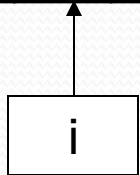
Lecture 13-1: binary search and complexity

**reading: 13.1-13.2**

# Sequential search

- **sequential search:** Locates a target value in an array / list by examining each element from start to finish. Used in `indexOf`.
  - How many elements will it need to examine?
  - Example: Searching the array below for the value **42**:

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
value	-4	2	7	10	15	20	22	25	30	36	42	50	56	68	85	92	103



- The array is sorted. Could we take advantage of this?

# Binary search (13.1)

- **binary search:** Locates a target value in a *sorted* array or list by successively eliminating half of the array from consideration.
  - How many elements will it need to examine?
  - Example: Searching the array below for the value **42**:

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
value	-4	2	7	10	15	20	22	25	30	36	42	50	56	68	85	92	103

Diagram illustrating a binary search on a sorted array. The array is shown with indices 0 to 16 and corresponding values. The value 42 is highlighted in yellow at index 10. Below the array, three boxes labeled 'min', 'mid', and 'max' are shown with arrows pointing to the corresponding indices: 'min' points to index 0, 'mid' points to index 8, and 'max' points to index 16.

# Arrays.binarySearch

```
// searches an entire sorted array for a given value
// returns its index if found; a negative number if not found
// Precondition: array is sorted
Arrays.binarySearch(array, value)
```

```
// searches given portion of a sorted array for a given value
// examines minIndex (inclusive) through maxIndex (exclusive)
// returns its index if found; a negative number if not found
// Precondition: array is sorted
Arrays.binarySearch(array, minIndex, maxIndex, value)
```

- The `binarySearch` method in the `Arrays` class searches an array very efficiently if the array is sorted.
  - You can search the entire array, or just a range of indexes (useful for "unfilled" arrays such as the one in `ArrayIntList`)

# Using `binarySearch`

```
// index    0  1  2  3  4  5  6  7  8  9  10  11  12  13  14  15
int[] a = {-4, 2, 7, 9, 15, 19, 25, 28, 30, 36, 42, 50, 56, 68, 85, 92};

int index = Arrays.binarySearch(a, 0, 16, 42); // index1 is 10
int index2 = Arrays.binarySearch(a, 0, 16, 21); // index2 is -7
```

- `binarySearch` returns the index where the value is found
- if the value is *not* found, `binarySearch` returns:
  - (`insertionPoint` + 1)
  - where `insertionPoint` is the index where the element *would* have been, if it had been in the array in sorted order.
  - To insert the value into the array, negate `insertionPoint + 1`

```
int indexToInsert21 = -(index2 + 1); // 6
```

# Runtime Efficiency (13.2)

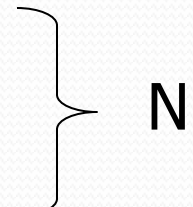
- How much better is binary search than sequential search?
- **efficiency**: measure of computing resources used by code.
  - can be relative to speed (time), memory (space), etc.
  - most commonly refers to run time
- Assume the following:
  - Any single Java statement takes same amount of time to run.
  - A method call's runtime is measured by the total of the statements inside the method's body.
  - A loop's runtime, if the loop repeats  $N$  times, is  $N$  times the runtime of the statements in its body.

# Efficiency examples

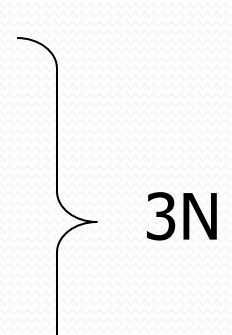
```
statement1;  
statement2;  
statement3;
```



```
for (int i = 1; i <= N; i++) {  
    statement4;  
}
```



```
for (int i = 1; i <= N; i++) {  
    statement5;  
    statement6;  
    statement7;  
}
```



$4N + 3$

# Efficiency examples 2

```
for (int i = 1; i <= N; i++) {  
    for (int j = 1; j <= N; j++) {  
        statement1;  
    }  
}
```

}  $N^2$

```
for (int i = 1; i <= N; i++) {  
    statement2;  
    statement3;  
    statement4;  
    statement5;  
}
```

}  $4N$

}  $N^2 + 4N$

- How many statements will execute if  $N = 10$ ? If  $N = 1000$ ?



# Algorithm growth rates (13.2)

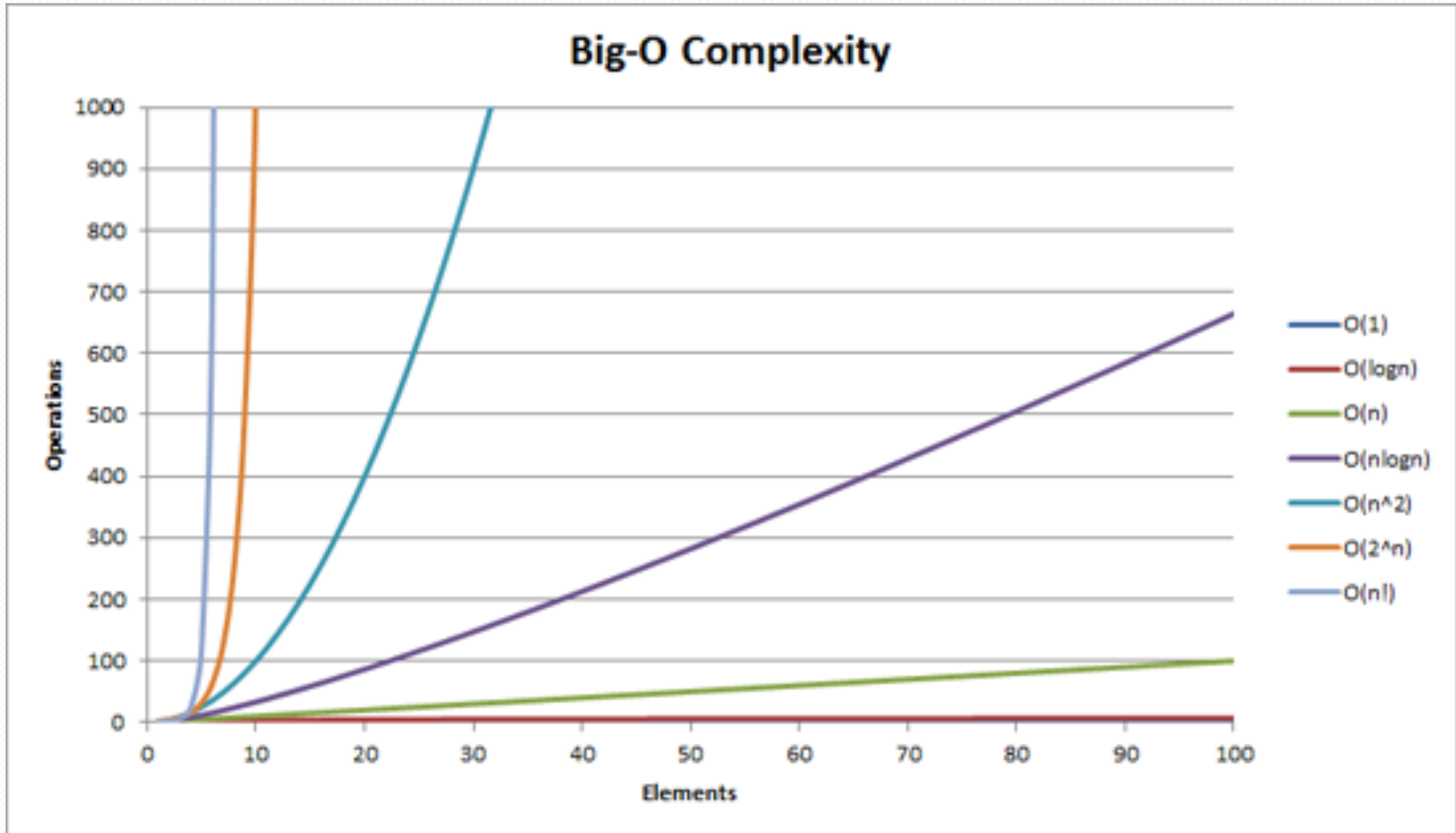
- We measure runtime in proportion to the input data size,  $N$ .
  - **growth rate**: Change in runtime as  $N$  changes.
- Say an algorithm runs  **$0.4N^3 + 25N^2 + 8N + 17$**  statements.
  - Consider the runtime when  $N$  is *extremely large* .
  - We ignore constants like 25 because they are tiny next to  $N$ .
  - The highest-order term ( $N^3$ ) dominates the overall runtime.
  - We say that this algorithm runs "on the order of"  $N^3$ .
  - or  **$O(N^3)$**  for short ("Big-Oh of  $N$  cubed")

# Complexity classes

- **complexity class:** A category of algorithm efficiency based on the algorithm's relationship to the input size  $N$ .

Class	Big-Oh	If you double $N$ , ...	Example
constant	$O(1)$	unchanged	10ms
logarithmic	$O(\log_2 N)$	increases slightly	175ms
linear	$O(N)$	doubles	3.2 sec
log-linear	$O(N \log_2 N)$	slightly more than doubles	6 sec
quadratic	$O(N^2)$	quadruples	1 min 42 sec
cubic	$O(N^3)$	multiplies by 8	55 min
...	...	...	...
exponential	$O(2^N)$	multiplies drastically	$5 * 10^{61}$ years

# Complexity classes



# Sequential search

- What is its complexity class?

```
public int indexOf(int value) {  
    for (int i = 0; i < size; i++) {  
        if (elementData[i] == value) {  
            return i;  
        }  
    }  
    return -1;    // not found  
}
```

} N

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
value	-4	2	7	10	15	20	22	25	30	36	42	50	56	68	85	92	103

- On average, "only"  $N/2$  elements are visited
  - $1/2$  is a constant that can be ignored

# Collection efficiency

- Efficiency of our `ArrayIntList` or Java's `ArrayList`:

<b>Method</b>	<b>ArrayList</b>
<code>add</code>	$O(1)$
<code>add(index, value)</code>	$O(N)$
<code>indexOf</code>	$O(N)$
<code>get</code>	$O(1)$
<code>remove</code>	$O(N)$
<code>set</code>	$O(1)$
<code>size</code>	$O(1)$

# Binary search

- **binary search** successively eliminates half of the elements.
  - *Algorithm:* Examine the middle element of the array.
    - If it is too big, eliminate the right half of the array and repeat.
    - If it is too small, eliminate the left half of the array and repeat.
    - Else it is the value we're searching for, so stop.
  - Which indexes does the algorithm examine to find value **42**?
  - What is the runtime complexity class of binary search?

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
value	-4	2	7	10	15	20	22	25	30	36	42	50	56	68	85	92	103

Diagram illustrating the binary search process on an array. The array contains 17 elements, indexed from 0 to 16. The values are: -4, 2, 7, 10, 15, 20, 22, 25, 30, 36, 42, 50, 56, 68, 85, 92, 103. The middle element (index 10, value 42) is highlighted in yellow. Three boxes labeled 'min', 'mid', and 'max' are positioned below the array, with arrows pointing to their respective indices (0, 10, and 16).

# Binary search runtime

- For an array of size  $N$ , it eliminates  $\frac{1}{2}$  until 1 element remains.

$N, N/2, N/4, N/8, \dots, 4, 2, 1$

- How many divisions does it take?
- Think of it from the other direction:
  - How many times do I have to multiply by 2 to reach  $N$ ?  
 $1, 2, 4, 8, \dots, N/4, N/2, N$
  - Call this number of multiplications " $x$ ".

$$2^x = N$$

$$\mathbf{x = \log_2 N}$$

- Binary search is in the **logarithmic** complexity class.

# Max subsequence sum

- Write a method `maxSum` to find the largest sum of any contiguous subsequence in an array of integers.
  - Easy for all positives: include the whole array.
  - What if there are negatives?

index	0	1	2	3	4	5	6	7	8
value	2	1	-4	10	15	-2	22	-8	5

Largest sum:  $10 + 15 + -2 + 22 = 45$

- (Let's define the max to be 0 if the array is entirely negative.)
- Ideas for algorithms?



# Algorithm 1 pseudocode

```
maxSum(a):  
    max = 0.  
    for each starting index i:  
        for each ending index j:  
            sum = add the elements from a[i] to a[j].  
            if sum > max,  
                max = sum.  
  
return max.
```

index	0	1	2	3	4	5	6	7	8
value	2	1	-4	10	15	-2	22	-8	5

# Algorithm 1 code

- What complexity class is this algorithm?
  - **$O(N^3)$** . Takes a few seconds to process 2000 elements.

```
public static int maxSum1(int[] a) {
    int max = 0;
    for (int i = 0; i < a.length; i++) {
        for (int j = i; j < a.length; j++) {
            // sum = add the elements from a[i] to a[j].
            int sum = 0;
            for (int k = i; k <= j; k++) {
                sum += a[k];
            }
            if (sum > max) {
                max = sum;
            }
        }
    }
    return max;
}
```

# Flaws in algorithm 1

- Observation: We are redundantly re-computing sums.
  - For example, we compute the sum between indexes 2 and 5:  
 $a[2] + a[3] + a[4] + a[5]$
  - Next we compute the sum between indexes 2 and 6:  
 $a[2] + a[3] + a[4] + a[5] + a[6]$
  - We already had computed the sum of 2-5, but we compute it again as part of the 2-6 computation.
  - Let's write an improved version that avoids this flaw.

index	0	1	2	3	4	5	6	7	8
value	2	1	-4	10	15	-2	22	-8	5

# Algorithm 2 code

- What complexity class is this algorithm?
  - **$O(N^2)$** . Can process tens of thousands of elements per second.

```
public static int maxSum2(int[] a) {  
    int max = 0;  
    for (int i = 0; i < a.length; i++) {  
        int sum = 0;  
        for (int j = i; j < a.length; j++) {  
            sum += a[j];  
            if (sum > max) {  
                max = sum;  
            }  
        }  
    }  
    return max;  
}
```

index	0	1	2	3	4	5	6	7	8
value	2	1	-4	10	15	-2	22	-8	5

# A clever solution

- *Claim 1* : A max range cannot start with a negative-sum range.

i	...	j	j+1	...	k
$< 0$			$\text{sum}(j+1, k)$		
$\text{sum}(i, k) < \text{sum}(j+1, k)$					

- *Claim 2* : If  $\text{sum}(i, j-1) \geq 0$  and  $\text{sum}(i, j) < 0$ , any max range that ends at  $j+1$  or higher cannot start at any of  $i$  through  $j$ .

i	...	j-1	j	j+1	...	k
$\geq 0$			$< 0$	$\text{sum}(j+1, k)$		
$< 0$				$\text{sum}(j+1, k)$		
$\text{sum}(?, k) < \text{sum}(j+1, k)$						

- Together, these observations lead to a very clever algorithm...

# Algorithm 3 code

- What complexity class is this algorithm?
  - **O(N)**. Handles many millions of elements per second!

```
public static int maxSum3(int[] a) {
    int max = 0;
    int sum = 0;
    int i = 0;
    for (int j = 0; j < a.length; j++) {
        if (sum < 0) {    // if sum becomes negative, max range
            i = j;      // cannot start with any of i - j-1
            sum = 0;    // (Claim 2)
        }
        sum += a[j];
        if (sum > max) {
            max = sum;
        }
    }
    return max;
}
```