Runtime Complexity with the UFPT

UF WiCSE Code-a-thon 2020

First, a problem

“Missing”

You’re given two arrays of numbers, creatively named *array1* and *array2*. The two arrays are extremely similar: *array2* has all the elements that *array1* has, except for exactly one. Write a program to find the missing element.

`[ 1, 2, 3, 4, 5 ]`
`[ 1, 2,    4, 5 ]`

Take some time and try it out!
Try to make your code run as fast as possible.

Template:
jdoodle.com/ia/4rV
Submit:
(2:35pm) https://forms.gle/bwPn51H5Ne5KGw2p8
(3:10pm) https://forms.gle/1QGrBNznbe7ji6486

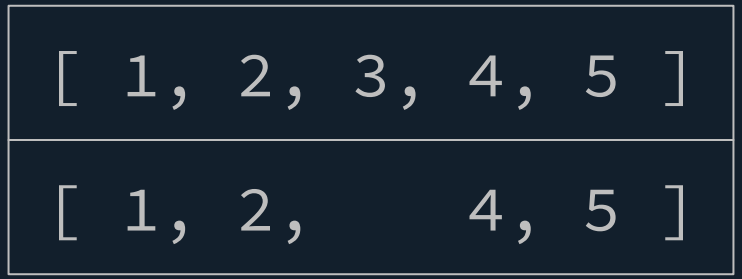
One possible solution

```
int n = array1.length;
int ans = 0;

for(int i = 0; i < n; i++) {
    int num = array1[i];
    boolean foundNum = false;

    for(int j = 0; j < n-1; j++) {
        int check = array2[j];
        if(check == num) {
            foundNum = true;
        }
    }
    if(!foundNum) {
        ans = num;
    }
}

return ans;
```

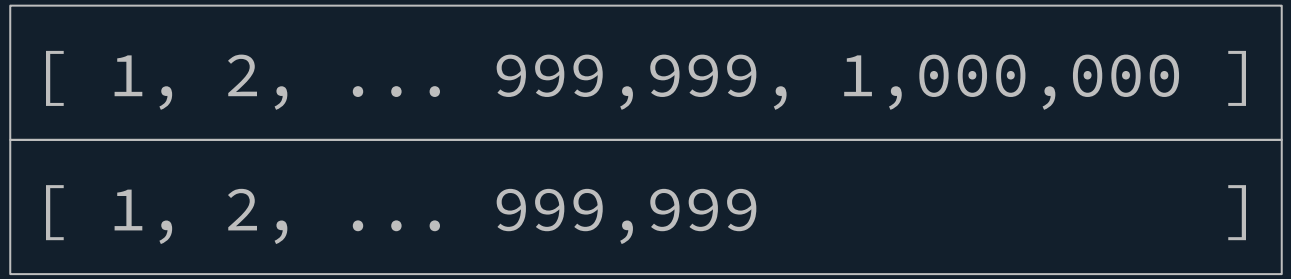
One possible solution - optimized

```
int n = array1.length;


for(int i = 0; i < n; i++) {
    int num = array1[i];
    boolean foundNum = false;

    for(int j = 0; j < n-1; j++) {
        int check = array2[j];
        if(check == num) {
            foundNum = true;
            break; // Stop trying as soon as we find the number
        }
    }
    if(!foundNum) {
        return num; // Return the answer as soon as we get it
    }
}

return 0; // We'll never get here, but the compiler needs this line
```

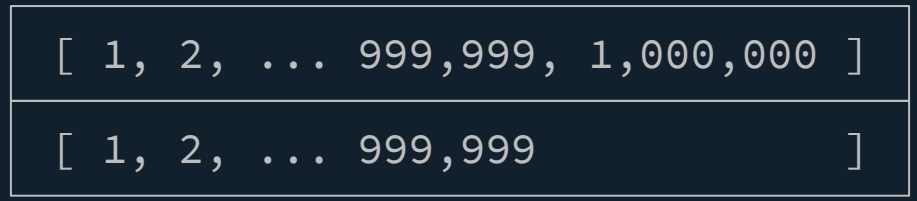
How long does the code take?



How long does the code take?



How long does the code take?



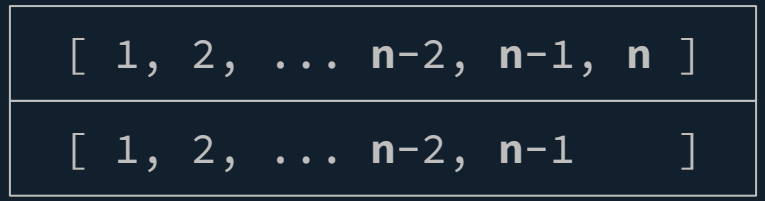
Unoptimized:

1,000,000 + 1,000,000 + ... + 1,000,000 + 1,000,000 = ~$1x10^{12}$ checks

Optimized (break early, return early):

1 + 2 + ... + 999,998 + 999,999 + 1 = ~$0.5x10^{12}$ checks

How long does the code take (in general)?



Unoptimized:

$\mathbf{n} + \mathbf{n} + \ldots + \mathbf{n} + \mathbf{n} = \sim\mathbf{n}^2$ checks

Optimized (break early, return early):

$1 + 2 + \ldots + \mathbf{n}\text{-}2 + \mathbf{n}\text{-}1 + 1 = \sim\mathbf{n}^2/2$ checks

Worst-case analysis

Worst-case scenario: the input that causes our code to run as long as possible

- Ask how long it will take to run
- When we do this in general, it's called *worst-case analysis*
- Language to describe how fast our code is

Big-O Notation

We want to know roughly *how much work the code does* given the size of the input, *in the worst possible scenario*

- Denote this is O(*something*)
- Something that takes a constant amount of time runs in O(1)

Since we just need a general feel, we have the following rules:

- Only take the fastest-growing term of the function
- Drop addition/multiplication of constants
- *Both the optimized and unoptimized solutions run in O($\boldsymbol{n}^2$)*

Really, really analyzing it

```
int n = array1.length;


for(int i = 0; i < n; i++) {
    int num = array1[i];
    boolean foundNum = false;

    for(int j = 0; j < n-1; j++) {
        int check = array2[j];
        if(check == num) {
            foundNum = true;
            break;
        }
    }
    if(!foundNum) {
        return num;
    }
}

return 0;
```

A much faster solution

```
int n = array1.length;

int sum1 = 0;
for(int i = 0; i < n; i++) {
    sum1 += array1[i];
}

int sum2 = 0;
for(int i = 0; i < n-1; i++) {
    sum2 += array2[i];
}

// Where the magic happens
int ans = sum1 - sum2;
return ans
```

Another fast solution

```
int n = array1.length;

// Java function that sort the arrays in ascending (increasing) order
Arrays.sort(array1);
Arrays.sort(array2);

for(int i = 0; i < n-1; i++) {
    int num = array1[i];
    int check = array2[i];

    if (num != check) {
        return num;
    }
}

// If we never find a mismatch, it must be the last element
return array1[n-1];
```

Another fast solution

```
int n = array1.length;

// Java function that sort the arrays in ascending (increasing) order
Arrays.sort(array1);
Arrays.sort(array2);

for(int i = 0; i < n-1; i++) {
    int num = array1[i];
    int check = array2[i];

    if (num != check) {
        return num;
    }
}

// If we never find a mismatch, it must be the last element
return array1[n-1];
```

Runtime complexity isn’t everything

- Without a well-made algorithm, your code will run slowly for large amounts of input

Runtime complexity isn’t everything

- Without a well-made algorithm, your code will run slowly for large amounts of input

But it’s not everything

- There are other forms of analysis, such as *average-case* and *best-case*
- Obviously, we’re not always going to get the worst possible scenario
  - (The automated tests we ran were mostly random)
- Sometimes we don’t even get big input!
  - If we had a max array size of **n** = 4, optimizing the solution would change nothing

(This is the part of the meeting where I check the time and either thank you for coming or realize there’s a lot of time left)

What's the big-O?

```
// Reverse a list
int n = list.length;
int i, j;
for(i = 0, j = n-1; i > j; i++, j--) {
    // Simple swap
    int temp = list[i];
    list[i] = list[j];
    list[j] = temp;
}
```

What's the big-O?

```
// Add elements at power-of-10 positions: 1, 10, 100, etc.
int n = list.length;
int sum = 0;
for(int i = 1; i < n; i++) {
    // Assume this function takes O(1) time
    if(isPowerOf10(i)) {
        sum += list[i];
    }
}

return sum;
```

What's the big-O?

```
// Sum of elements at power-of-2 positions
int n = list.length;
int sum = 0;
for(int i = 1; i < n; i = i*10) {
    sum += list[i];
}

return sum;
```

What's the big-O?

```
// Finds the maximum product between two elements in the lists
int n = list1.length;
int m = list2.length;

int ans = 0;
for(int i = 0; i < n; i++) {
    for(int j = 0; j < m; j++) {
        int product = list1[i] * list2[j];
        if(product > ans) {
            ans = product;
        }
    }
}
return ans;
```

What’s the big-O?

```
// Bogosort implementation
while(!isSorted(list)) {
    randomlyShuffle(list);
}
```