Computational Complexity

Computational Complexity

- In order to make most effective use of our computational resources, it’s important that we have the skill set to analyze the complexity of algorithms, so we know what resources those algorithms require.

- Being able to analyze an algorithm allows us to have an idea of how well it scales as we throw larger and larger data sets at it.

Computational Complexity

- When we talk about the complexity of an algorithm, we generally refer to the **worst-case scenario**.
  - We refer to this as O.
- We sometimes also care about the **best-case scenario** (also known as Ω)
- In CS50, we'll leave the rigorous analysis aside and focus more on reasoning complexity with common sense.

Computational Complexity

- What is a data set?
    - Whatever makes the most sense in context.

- We can measure an algorithm based on how it handles these inputs. Let’s call this measure *f(n)*.

- We don’t actually care about what *f(n)* is precisely. Rather, we care only about its **tendency**, which is dictated by its highest-order term.

Computational Complexity

n	$f(n) = n^3$	$f(n) = n^3 + n^2$	$f(n) = n^3 - 8n^2 + 20n$
1	1	2	13
10	1,000	1,100	400
1,000	1,000,000,000	1,001,000,000	992,020,000
1,000,000	1.0×10^{18}	1.000001×10^{18}	9.99992×10^{17}

Computational Complexity

n	$f(n) = n^3$	$f(n) = n^3 + n^2$	$f(n) = n^3 - 8n^2 + 20n$
1	1	2	13
10	1,000	1,100	400
1,000	1,000,000,000	1,001,000,000	992,020,000
1,000,000	1.0×10^{18}	1.000001×10^{18}	9.99992×10^{17}

Computational Complexity

n	$f(n) = n^3$	$f(n) = n^3 + n^2$	$f(n) = n^3 – 8n^2 + 20n$
1	1	2	13
10	1,000	1,100	400
1,000	1,000,000,000	1,001,000,000	992,020,000
1,000,000	1.0×10^{18}	1.000001×10^{18}	9.99992×10^{17}

Computational Complexity

n	$f(n) = n^3$	$f(n) = n^3 + n^2$	$f(n) = n^3 - 8n^2 + 20n$
1	1	2	13
10	1,000	1,100	400
1,000	1,000,000,000	1,001,000,000	992,020,000
1,000,000	1.0×10^{18}	1.000001×10^{18}	9.99992×10^{17}

Computational Complexity

n	$f(n) = n^3$	$f(n) = n^3 + n^2$	$f(n) = n^3 - 8n^2 + 20n$
1	1	2	13
10	1,000	1,100	400
1,000	1,000,000,000	1,001,000,000	992,020,000
1,000,000	1.0×10^{18}	1.000001×10^{18}	9.99992×10^{17}

Computational Complexity

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^c)	polynomial time
O(c^n)	exponential time
O($n!$)	factorial time
O(∞)	infinite time

Computational Complexity

- O(*1*)
  - Always takes a single operation in the worst case.

```
int four_for_you(int array[1000])
{
   return 4;
}

int add_two_nums(int a, int b)
{
   return a + b;
}
```

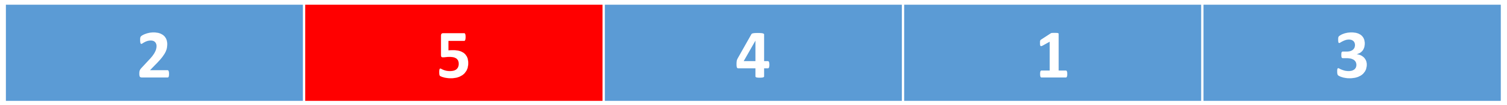
Computational Complexity

- O(n)
  - Always takes n operations in the worst case.

2	5	4	1	3

Computational Complexity

- O(*n*)
  - Always takes *n* operations in the worst case.



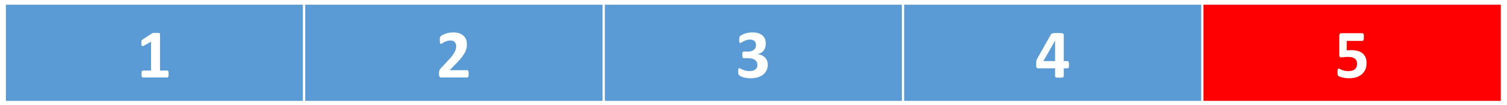
Computational Complexity

- O(n)
    - Always takes n operations in the worst case.

1	2	3	4	5

Computational Complexity

- O(*n*)
  - Always takes *n* operations in the worst case.



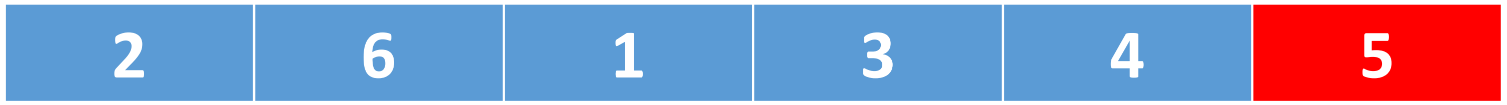
Computational Complexity

- O(*n*)
  - Always takes *n* operations in the worst case.

2	**6**	**1**	**3**	**4**	**5**

Computational Complexity

- O(n)
  - Always takes n operations in the worst case.



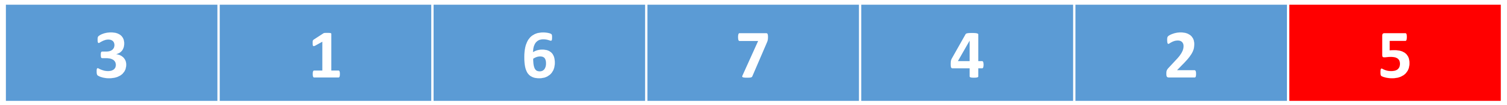
Computational Complexity

- O(*n*)
  - Always takes *n* operations in the worst case.

3	1	6	7	4	2	5

Computational Complexity

- O(*n*)
  - Always takes *n* operations in the worst case.



Computational Complexity

- What’s the runtime?

```
for (int j = 0; j < m; j++)
{
    // loop body that runs in O(1)
}
```

Computational Complexity

- What's the runtime?

```
for (int j = 0; j < m; j++)
{
    // loop body that runs in O(1)
}
```

O(m)

Computational Complexity

- What’s the runtime?

```
for (int j = 0; j < p; j++)
{
    for (int k = 0; k < p; k++)
    {
        // loop body that runs in O(1)
    }
}
```

Computational Complexity

- What’s the runtime?

```
for (int j = 0; j < p; j++)
{
    for (int k = 0; k < p; k++)
    {
        // loop body that runs in O(1)
    }
}
```

O(p^2)