Big-O Analysis

Elyse Cornwall

July 6th, 2023

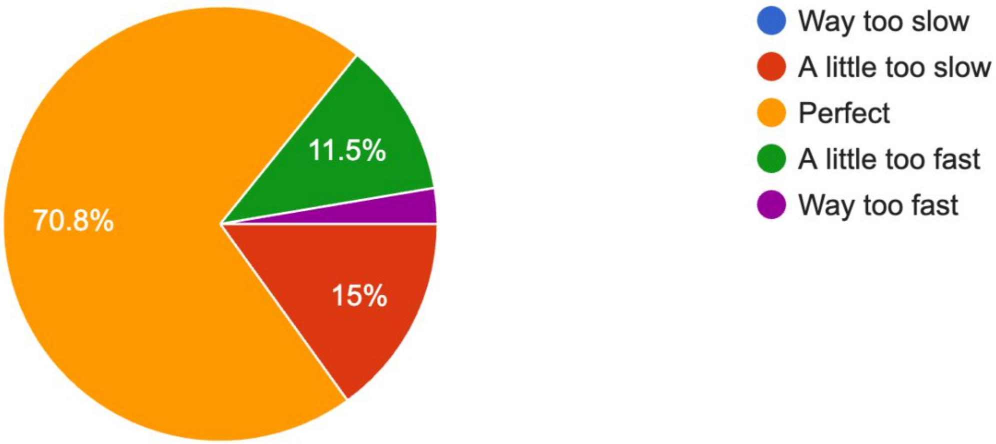
Contributions made from previous CS106B Instructors

Stanford University

Week 1 Feedback

Rate the pace of lecture

113 responses



Week 1 Feedback

Things you liked:

“I like the fact that you guys give time to **answer specific questions** that the students might have.”

“I enjoy the **modeling of things on the whiteboard**, it makes it easy to follow concepts.”

“Lots of **worked examples**!! Big fan of that.”

Week 1 Feedback

Places we can improve:

“some of the more complex questions and answers are more confusing than helpful”

“I think the participation tickets are a little hard”

“Would like a stronger emphasis on a recap at the end of each session”

Week 1 Feedback

We hear you…

“It would also be really helpful if you could release the lecture slides a day or two in advance”

“It's complicated having resources on so many different sites like ed, cppreference.com, cs106b.stanford.edu, etc.”

“Plz use VScode for future students :(”

Announcements

- Week 2 feedback survey is out (✨ bonus participation points ✨)
- Tomorrow (Friday 7/7) is course add/drop deadline
- Assignment 1 is due tomorrow at 11:59pm
    - Help resources drop off over the weekend, go to LaIR tonight!
- Assignment 2 will be released tomorrow afternoon
    - Assignment 2 YEAH Hours on Friday from 3-4pm at this Zoom link

ADT Highlight Reel

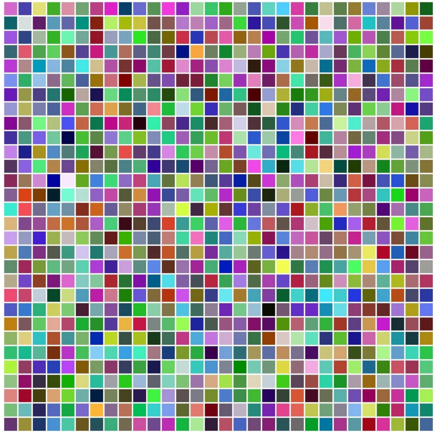
Recap of ADTs

Ordered ADTs

4	7	-3	6
0	1	2	3

Elements with indices

- Vectors (1D)
- Grids (2D)



Recap of ADTs

Ordered ADTs

Elements with indices

- Vectors (1D)
- Grids (2D)

Elements without indices

- Stacks (LIFO)
- Queues (FIFO)

Stanford University

Recap of ADTs

Ordered ADTs

Elements with indices

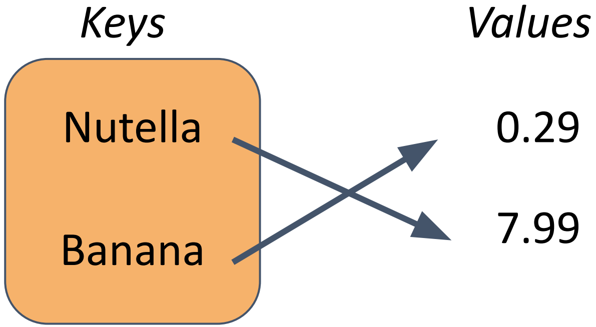
- Vectors (1D)
- Grids (2D)

Elements without indices

- Stacks (LIFO)
- Queues (FIFO)

Unordered ADTs

- Sets (unique elements)
- Maps (key, value pairs)



Nested ADTs

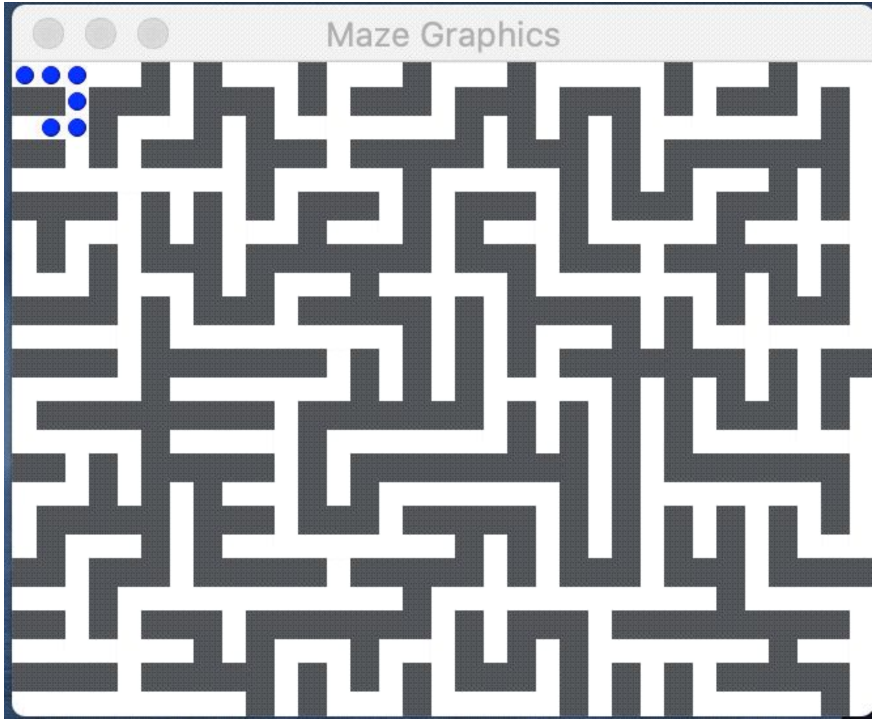
- We can “nest” ADTs (e.g. `Map<string, Set<string>>`)
- This allows us to represent more complex data
- Nested ADTs can be tricky to work with, especially because of reference and copies

Assignment 2: Fun with Collections!

`Grid<bool>`

Each location is either:

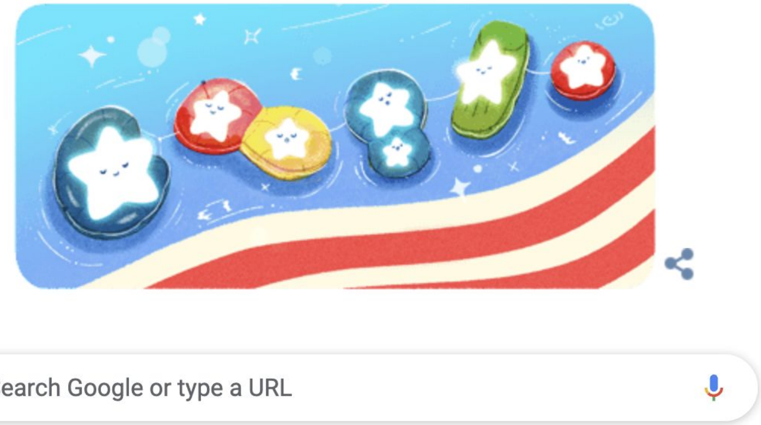
- Corridor (`true`)
- Wall (`false`)



Assignment 2: Fun with Collections!

```
Map<string, Set<string>>
```

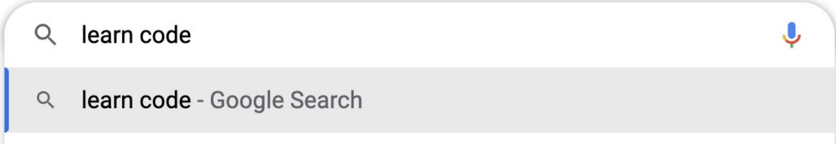
Keyword, URLs



Assignment 2: Fun with Collections!

```
{“learn”: {“desmos.com”, “stanford.edu”},
 “code” : {“stanford.edu”, “cpp.com”}, ... }
```

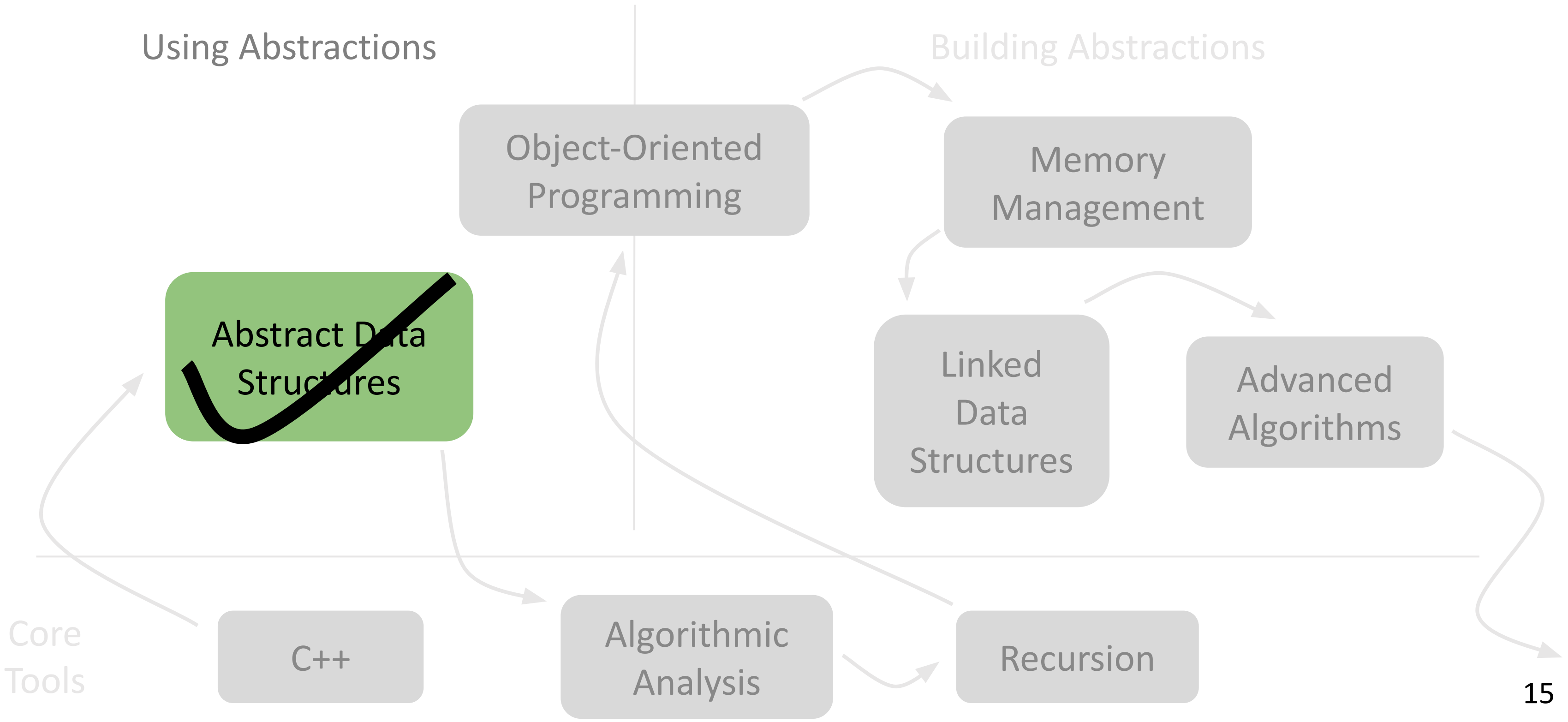
Keyword, URLs



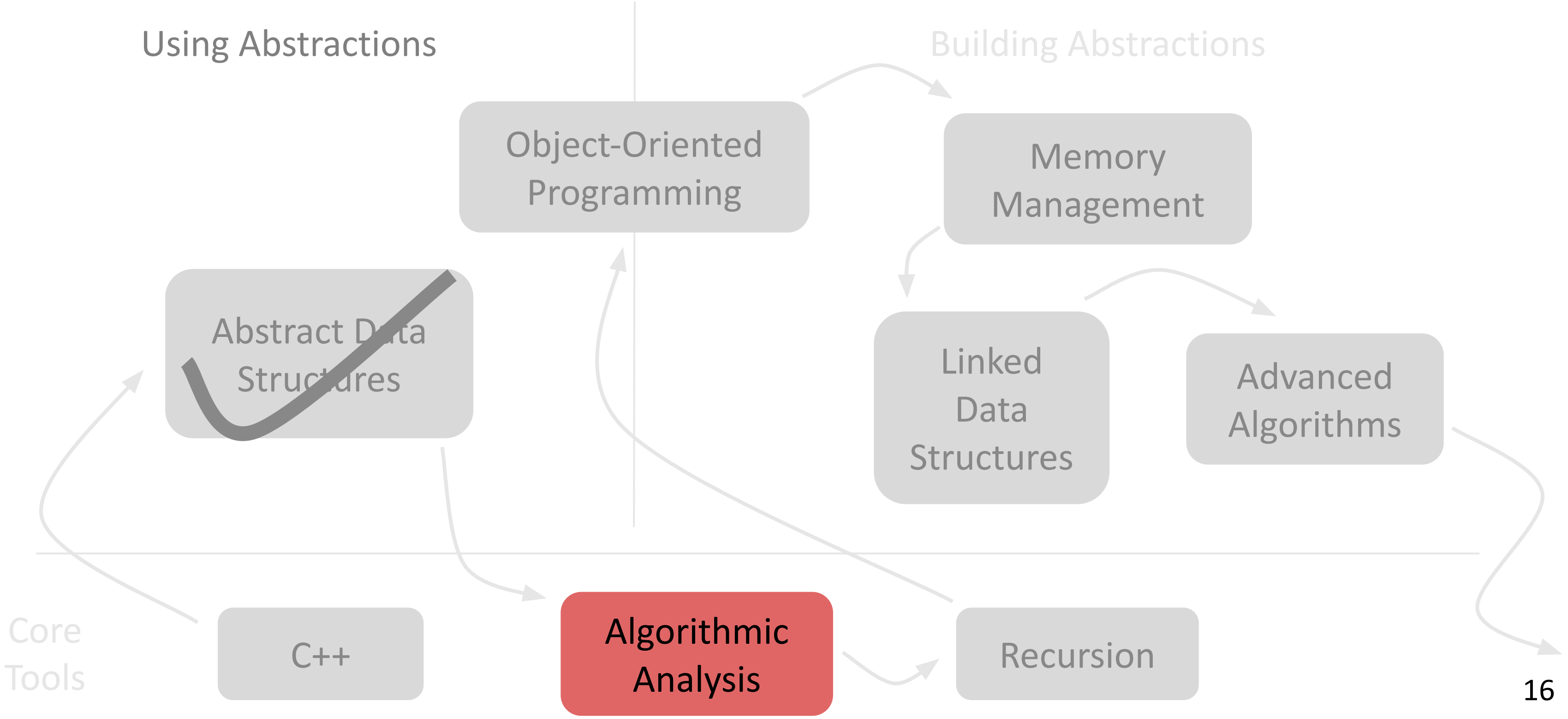
learn code

learn code - Google Search

Roadmap


Using Abstractions
Building Abstractions
Object-Oriented Programming
Memory Management
Abstract Data Structures
Linked Data Structures
Advanced Algorithms
Core Tools
C++
Algorithmic Analysis
Recursion

Roadmap


Using Abstractions
Building Abstractions
Object-Oriented Programming
Memory Management
Abstract Data Structures
Linked Data Structures
Advanced Algorithms
Core Tools
C++
Algorithmic Analysis
Recursion

👥 Discuss with a Neighbor

What does it mean for a program to be

- “Faster”
- “More efficient”
- “Better”

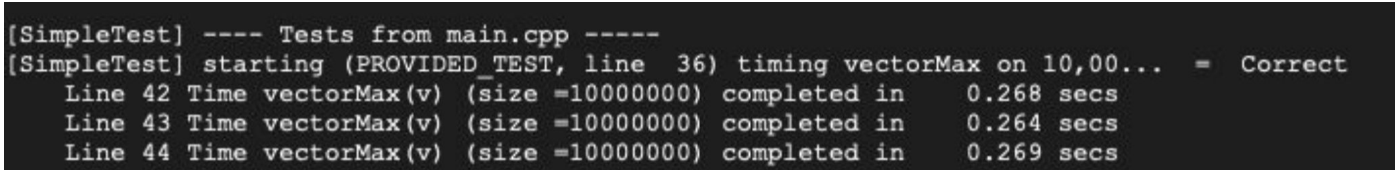
than another program?

Is it Fast?

Measuring the speed of our programs

One Idea: Runtime

- Measure how long a program takes to complete
- Example: timing the `vectorMax` function

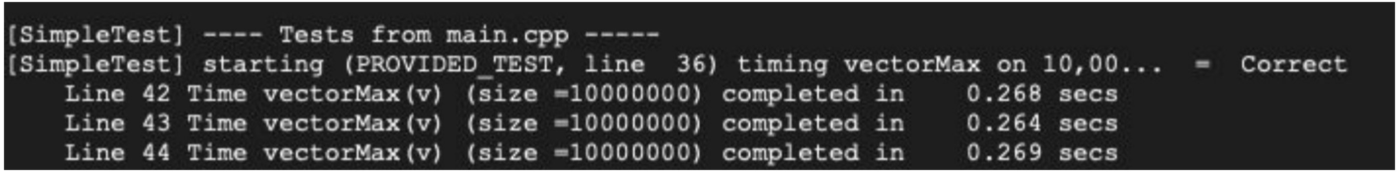


```
[SimpleTest] ---- Tests from main.cpp -----
[SimpleTest] starting (PROVIDED_TEST, line  36) timing vectorMax on 10,00...  =  Correct
   Line 42 Time vectorMax(v) (size =10000000) completed in    0.268 secs
   Line 43 Time vectorMax(v) (size =10000000) completed in    0.264 secs
   Line 44 Time vectorMax(v) (size =10000000) completed in    0.269 secs
```

Running on a
2012 MacBook

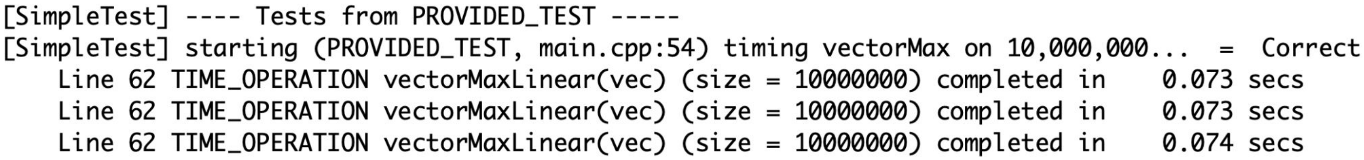
One Idea: Runtime

- Measure how long a program takes to complete
- Example: timing the `vectorMax` function



```
[SimpleTest] ---- Tests from main.cpp -----
[SimpleTest] starting (PROVIDED_TEST, line  36) timing vectorMax on 10,00...  =  Correct
    Line 42 Time vectorMax(v) (size =10000000) completed in    0.268 secs
    Line 43 Time vectorMax(v) (size =10000000) completed in    0.264 secs
    Line 44 Time vectorMax(v) (size =10000000) completed in    0.269 secs
```

Running on a 2012 MacBook



```
[SimpleTest] ---- Tests from PROVIDED_TEST -----
[SimpleTest] starting (PROVIDED_TEST, main.cpp:54) timing vectorMax on 10,000,000...  =  Correct
    Line 62 TIME_OPERATION vectorMaxLinear(vec) (size = 10000000) completed in    0.073 secs
    Line 62 TIME_OPERATION vectorMaxLinear(vec) (size = 10000000) completed in    0.073 secs
    Line 62 TIME_OPERATION vectorMaxLinear(vec) (size = 10000000) completed in    0.074 secs
```

Running on a 2020 MacBook

Why Runtime Isn’t Enough

Runtime depends on:

- The computer you’re using
- Other applications running on your computer
- Whether your computer is trying to conserve power
- And more!

Another Idea: Number of Operations

- We could count the number of operations, or steps it takes for a program to complete
- This doesn’t change across computers, as long as the input to our program is the same

Analyzing vectorMax

```
int vectorMax(Vector<int> &v) {
    int currentMax = v[0];
    int n = v.size();
    for (int i = 1; i < n; i++) {
        if (currentMax < v[i]) {
            currentMax = v[i];
        }
    }
    return currentMax;
}
```

Analyzing vectorMax

```
int vectorMax(Vector<int> &v) {
    int currentMax = v[0];
    int n = v.size();
    for (int i = 1; i < n; i++) {
        if (currentMax < v[i]) {
            currentMax = v[i];
        }
    }
    return currentMax;
}
```

Take in a Vector of ints and return the maximum value

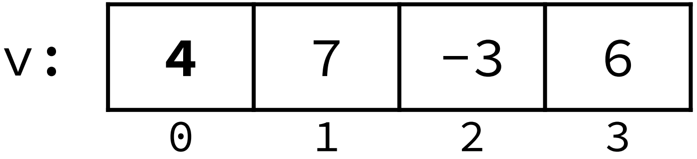
Analyzing vectorMax

v:	4	7	-3	6
	0	1	2	3

```
int vectorMax(Vector<int> &v) {
    int currentMax = v[0];
    int n = v.size();
    for (int i = 1; i < n; i++) {
        if (currentMax < v[i]) {
            currentMax = v[i];
        }
    }
    return currentMax;
}
```

Analyzing vectorMax

```
int vectorMax(Vector<int> &v) {
    int currentMax = v[0];
    int n = v.size();
    for (int i = 1; i < n; i++) {
        if (currentMax < v[i]) {
            currentMax = v[i];
        }
    }
    return currentMax;
}
```



v:

4	7	-3	6
0	1	2	3

currentMax: 4

Analyzing vectorMax

```
int vectorMax(Vector<int> &v) {
    int currentMax = v[0];
    int n = v.size();
    for (int i = 1; i < n; i++) {
        if (currentMax < v[i]) {
            currentMax = v[i];
        }
    }
    return currentMax;
}
```

v:	4	7	-3	6
	0	1	2	3

currentMax: 4

n: 4

Analyzing vectorMax

```
int vectorMax(Vector<int> &v) {
    int currentMax = v[0];
    int n = v.size();
    for (int i = 1; i < n; i++) {
        if (currentMax < v[i]) {
            currentMax = v[i];
        }
    }
    return currentMax;
}
```

v:

4	7	-3	6
0	1	2	3

currentMax: 4

n: 4

i: **1**

Analyzing vectorMax

```
int vectorMax(Vector<int> &v) {
    int currentMax = v[0];
    int n = v.size();
    for (int i = 1; i < n; i++) {
        if (currentMax < v[i]) {
            currentMax = v[i];
        }
    }
    return currentMax;
}
```

v:

4	7	-3	6
0	1	2	3

currentMax: 4

n: **4**

i: **1**

Analyzing vectorMax

```
int vectorMax(Vector<int> &v) {
    int currentMax = v[0];
    int n = v.size();
    for (int i = 1; i < n; i++) {
        if (currentMax < v[i]) {
            currentMax = v[i];
        }
    }
    return currentMax;
}
```

v:

v:	4	**7**	-3	6
	0	1	2	3

currentMax: **4**

n: 4

i: 1

Analyzing vectorMax

```
int vectorMax(Vector<int> &v) {
    int currentMax = v[0];
    int n = v.size();
    for (int i = 1; i < n; i++) {
        if (currentMax < v[i]) {
            currentMax = v[i];
        }
    }
    return currentMax;
}
```

v:

4	**7**	-3	6
0	1	2	3

currentMax: **7**

n: 4

i: 1

Analyzing vectorMax

```
int vectorMax(Vector<int> &v) {
    int currentMax = v[0];
    int n = v.size();
    for (int i = 1; i < n; i++) {
        if (currentMax < v[i]) {
            currentMax = v[i];
        }
    }
    return currentMax;
}
```

v:

4	7	-3	6
0	1	2	3

currentMax: 7

n: 4

i: **2**

Analyzing vectorMax

```
int vectorMax(Vector<int> &v) {
    int currentMax = v[0];
    int n = v.size();
    for (int i = 1; i < n; i++) {
        if (currentMax < v[i]) {
            currentMax = v[i];
        }
    }
    return currentMax;
}
```

v:

4	7	-3	6
0	1	2	3

currentMax: 7

n: **4**

i: **2**

Analyzing vectorMax

```
int vectorMax(Vector<int> &v) {
    int currentMax = v[0];
    int n = v.size();
    for (int i = 1; i < n; i++) {
        if (currentMax < v[i]) {
            currentMax = v[i];
        }
    }
    return currentMax;
}
```

v:	4	7	**-3**	6
	0	1	2	3

currentMax: **7**

n: 4

i: 2

Analyzing vectorMax

```
int vectorMax(Vector<int> &v) {
    int currentMax = v[0];
    int n = v.size();
    for (int i = 1; i < n; i++) {
        if (currentMax < v[i]) {
            currentMax = v[i];
        }
    }
    return currentMax;
}
```

v:

4	7	-3	6
0	1	2	3

currentMax: 7

n: 4

i: **3**

Analyzing vectorMax

```
int vectorMax(Vector<int> &v) {
    int currentMax = v[0];
    int n = v.size();
    for (int i = 1; i < n; i++) {
        if (currentMax < v[i]) {
            currentMax = v[i];
        }
    }
    return currentMax;
}
```

v:

4	7	-3	6
0	1	2	3

currentMax: 7

n: **4**

i: **3**

Analyzing vectorMax

```
int vectorMax(Vector<int> &v) {
    int currentMax = v[0];
    int n = v.size();
    for (int i = 1; i < n; i++) {
        if (currentMax < v[i]) {
            currentMax = v[i];
        }
    }
    return currentMax;
}
```

v:

4	7	-3	**6**
0	1	2	3

currentMax: **7**

n: 4

i: 3

Analyzing vectorMax

```
int vectorMax(Vector<int> &v) {
    int currentMax = v[0];
    int n = v.size();
    for (int i = 1; i < n; i++) {
        if (currentMax < v[i]) {
            currentMax = v[i];
        }
    }
    return currentMax;
}
```

v:

4	7	-3	6
0	1	2	3

```
currentMax: 7
```

n: **4**

i: **4**

Analyzing vectorMax

```
int vectorMax(Vector<int> &v) {
    int currentMax = v[0];
    int n = v.size();
    for (int i = 1; i < n; i++) {
        if (currentMax < v[i]) {
            currentMax = v[i];
        }
    }
    return currentMax;
}
```

v:	4	7	-3	6
	0	1	2	3

currentMax: 7

n: **4**

i: **4**

Analyzing vectorMax

```
int vectorMax(Vector<int> &v) {
    int currentMax = v[0];
    int n = v.size();
    for (int i = 1; i < n; i++) {
        if (currentMax < v[i]) {
            currentMax = v[i];
        }
    }
    return currentMax;
}
```

v:	4	7	-3	6
	0	1	2	3

currentMax: **7**

n: 4

i: 3

Analyzing `vectorMax`

```
int vectorMax(Vector<int> &v) {
    int currentMax = v[0];
    int n = v.size();
    for (int i = 1; i < n; i++) {
        if (currentMax < v[i]) {
            currentMax = v[i];
        }
    }
    return currentMax;
}
```

What are the “operations” in this function?

Analyzing vectorMax

```
int vectorMax(Vector<int> &v) {
    int currentMax = v[0];
    int n = v.size();
    for (int i = 1; i < n; i++) {
        if (currentMax < v[i]) {
            currentMax = v[i];
        }
    }
    return currentMax;
}
```

Initialize

Analyzing vectorMax

```
int vectorMax(Vector<int> &v) {
    int currentMax = v[0];                   Initialize
    int n = v.size();                        Initialize
    for (int i = 1; i < n; i++) {
        if (currentMax < v[i]) {
            currentMax = v[i];
        }
    }
    return currentMax;
}
```

Analyzing vectorMax

```
int vectorMax(Vector<int> &v) {
    int currentMax = v[0];                      Initialize
    int n = v.size();                           Initialize
    for (int i = 1; i < n; i++) {               Initialize
        if (currentMax < v[i]) {
            currentMax = v[i];
        }
    }
    return currentMax;
}
```

Analyzing vectorMax

```
int vectorMax(Vector<int> &v) {
    int currentMax = v[0];                        Initialize
    int n = v.size();                             Initialize
    for (int i = 1; i < n; i++) {                 Initialize
        if (currentMax < v[i]) {                  Compare
            currentMax = v[i];
        }
    }
    return currentMax;
}
```

Analyzing vectorMax

```
int vectorMax(Vector<int> &v) {
    int currentMax = v[0];                  Initialize
    int n = v.size();                       Initialize
    for (int i = 1; i < n; i++) {           Initialize
        if (currentMax < v[i]) {            Compare
            currentMax = v[i];              Increment
        }
    }
    return currentMax;
}
```

Analyzing vectorMax

```
int vectorMax(Vector<int> &v) {
    int currentMax = v[0];                   Initialize
    int n = v.size();                        Initialize
    for (int i = 1; i < n; i++) {            Initialize
        if (currentMax < v[i]) {             Compare
            currentMax = v[i];               Increment
        }                                    Compare
    }
    return currentMax;
}
```

Analyzing vectorMax

```
int vectorMax(Vector<int> &v) {
    int currentMax = v[0];                      Initialize
    int n = v.size();                           Initialize
    for (int i = 1; i < n; i++) {               Initialize
        if (currentMax < v[i]) {                Compare
            currentMax = v[i];                  Increment
        }                                       Compare
    }                                           Reassign
    return currentMax;
}
```

Analyzing vectorMax

```
int vectorMax(Vector<int> &v) {
    int currentMax = v[0];                   Initialize
    int n = v.size();                        Initialize
    for (int i = 1; i < n; i++) {            Initialize
        if (currentMax < v[i]) {             Compare
            currentMax = v[i];               Increment
        }                                    Compare
    }                                        Reassign
    return currentMax;                       Return
}
```

Analyzing `vectorMax`

Now, how many times do we repeat each operation?

```
int vectorMax(Vector<int> &v) {
    int currentMax = v[0];
    int n = v.size();
    for (int i = 1; i < n; i++) {
        if (currentMax < v[i]) {
            currentMax = v[i];
        }
    }
    return currentMax;
}
```

Initialize
Initialize
Initialize
Compare
Increment
Compare
Reassign
Return

Analyzing vectorMax

Now, how many times do we repeat each operation?

```
int vectorMax(Vector<int> &v) {
    int currentMax = v[0];
    int n = v.size();
    for (int i = 1; i < n; i++) {
        if (currentMax < v[i]) {
            currentMax = v[i];
        }
    }
    return currentMax;
}
```

1 Initialize

Initialize

Initialize

Compare

Increment

Compare

Reassign

Return

Analyzing `vectorMax`

Now, how many times do we repeat each operation?

```
int vectorMax(Vector<int> &v) {
    int currentMax = v[0];
    int n = v.size();
    for (int i = 1; i < n; i++) {
        if (currentMax < v[i]) {
            currentMax = v[i];
        }
    }
    return currentMax;
}
```

1 Initialize

1 Initialize

Initialize

Compare

Increment

Compare

Reassign

Return

Analyzing vectorMax

```
int vectorMax(Vector<int> &v) {
    int currentMax = v[0];
    int n = v.size();
    for (int i = 1; i < n; i++) {
        if (currentMax < v[i]) {
            currentMax = v[i];
        }
    }
    return currentMax;
}
```

Now, how many times do we repeat each operation?

1 Initialize
1 Initialize
1 Initialize
Compare
Increment
Compare
Reassign
Return

Analyzing vectorMax

Now, how many times do we repeat each operation?

```
int vectorMax(Vector<int> &v) {
    int currentMax = v[0];
    int n = v.size();
    for (int i = 1; i < n; i++) {
        if (currentMax < v[i]) {
            currentMax = v[i];
        }
    }
    return currentMax;
}
```

1 Initialize

1 Initialize

1 Initialize

? Compare

Increment

Compare

Reassign

Return

Analyzing vectorMax

```
int vectorMax(Vector<int> &v) {
    int currentMax = v[0];
    int n = v.size();
    for (int i = 1; i < n; i++) {
        if (currentMax < v[i]) {

        }
    }
    return currentMax;
}
```

👥 How many times did we compare `i < n`?

Now, how many times do we repeat each operation?

1 Initialize
1 Initialize
1 Initialize
? Compare
Increment
Compare
Reassign
Return

Analyzing vectorMax

```
int vectorMax(Vector<int> &v) {
    int currentMax = v[0];
    int n = v.size();
    for (int i = 1; i < n; i++) {
        if (currentMax < v[i]) {

        }
    }
    return
}
```

```
1 < 4  // if n = 4,
2 < 4
3 < 4
4 < 4  // 4 times!
```

Now, how many times do we repeat each operation?

1 Initialize
1 Initialize
1 Initialize
? Compare
Increment
Compare
Reassign
Return

Analyzing vectorMax

```
int vectorMax(Vector<int> &v) {
    int currentMax = v[0];
    int n = v.size();
    for (int i = 1; i < n; i++) {
        if (currentMax < v[i]) {

        }
    }
    return currentMax;
}
```

```
1 < 2  // if n = 2,
2 < 2  // 2 times!
```

Now, how many times do we repeat each operation?

1 Initialize
1 Initialize
1 Initialize
? Compare
Increment
Compare
Reassign
Return

Analyzing vectorMax

```
int vectorMax(Vector<int> &v) {
    int currentMax = v[0];
    int n = v.size();
    for (int i = 1; i < n; i++) {
        if (currentMax < v[i]) {

        }
    }
    return
}
```

```
1 < n
2 < n
3 < n
...     // n times!
```

Now, how many times do we repeat each operation?

1 Initialize
1 Initialize
1 Initialize
? Compare
Increment
Compare
Reassign
Return

Analyzing vectorMax

Now, how many times do we repeat each operation?

```
int vectorMax(Vector<int> &v) {
    int currentMax = v[0];
    int n = v.size();
    for (int i = 1; i < n; i++) {
        if (currentMax < v[i]) {
            currentMax = v[i];
        }
    }
    return currentMax;
}
```

1 Initialize

1 Initialize

1 Initialize

n Compare

Increment

Compare

Reassign

Return

Analyzing `vectorMax`

Now, how many times do we repeat each operation?

```
int vectorMax(Vector<int> &v) {
    int currentMax = v[0];
    int n = v.size();
    for (int i = 1; i < n; i++) {
        if (currentMax < v[i]) {
            currentMax = v[i];
        }
    }
    return currentMax;
}
```

1 Initialize

1 Initialize

1 Initialize

n Compare

n - 1 Increment

Compare

Reassign

Return

Analyzing vectorMax

```
int vectorMax(Vector<int> &v) {
    int currentMax = v[0];
    int n = v.size();
    for (int i = 1; i < n; i++) {
        if (currentMax < v[i]) {
            currentMax = v[i];
        }
    }
    return currentMax;
}
```

Now, how many times do we repeat each operation?

1 Initialize
1 Initialize
1 Initialize
n Compare
n - 1 Increment
n - 1 Compare
Reassign
Return

Analyzing `vectorMax`

```
int vectorMax(Vector<int> &v) {
    int currentMax = v[0];
    int n = v.size();
    for (int i = 1; i < n; i++) {
        if (currentMax < v[i]) {
            currentMax = v[i];
        }
    }
    return currentMax;
}
```

Now, how many times do we repeat each operation?

1 Initialize
1 Initialize
1 Initialize
n Compare
n - 1 Increment
n - 1 Compare
(up to) n - 1 Reassign
Return

Analyzing `vectorMax`

```
int vectorMax(Vector<int> &v) {
    int currentMax = v[0];
    int n = v.size();
    for (int i = 1; i < n; i++) {
        if (currentMax < v[i]) {
            currentMax = v[i];
        }
    }
    return currentMax;
}
```

Now, how many times do we repeat each operation?

1 Initialize
1 Initialize
1 Initialize
n Compare
n - 1 Increment
n - 1 Compare
(up to) n - 1 Reassign
1 Return

Analyzing vectorMax

```
int vectorMax(Vector<int> &v) {
    int currentMax = v[0];
    int n = v.size();
    for (int i = 1; i < n; i++) {
        if (currentMax < v[i]) {
            currentMax = v[i];
        }
    }
    return currentMax;
}
```

Now, let's sum it up!

1 + 1 + 1 + n + n - 1 + n - 1 + n - 1 + 1 = **4n + 1**

Analyzing `vectorMax`

```
int vectorMax(Vector<int> &v) {
    int currentMax = v[0];
    int n = v.size();
    for (int i = 1; i < n; i++) {
        if (currentMax < v[i]) {
            currentMax = v[i];
        }
    }
    return currentMax;
}
```

This program takes at most 4n + 1 operations.

… what does this tell us?

Another Idea: Number of Operations

- We could count the number of operations, or steps it takes for a program to complete
- This doesn’t change across computers, as long as the input to our program is the same

This is still too much detail

Some of those constant operations might depend on your computer

The Big Idea: Big-O

- General enough to compare across different computer systems
- Focuses on how the runtime will grow with the input size
  - It’s all about growth rate
- This allows us to predict the runtime of future inputs

Calculating Big-O of `vectorMax`

```
4n + 1
```

Calculating Big-O of `vectorMax`

- Remove lower-order terms including constants

$$4n + 1$$

Calculating Big-O of `vectorMax`

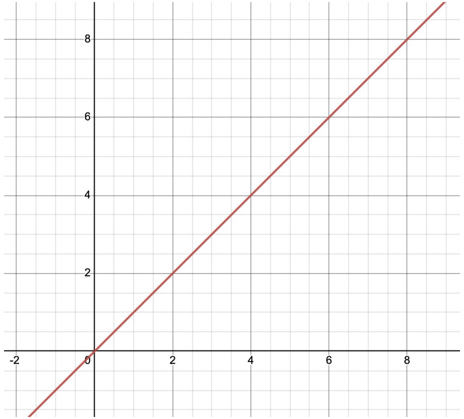
- Remove lower-order terms including constants
- Get rid of leading coefficients

$$4n + 1$$

Calculating Big-O of `vectorMax`

- Remove lower-order terms including constants
- Get rid of leading coefficients

`O(n)`

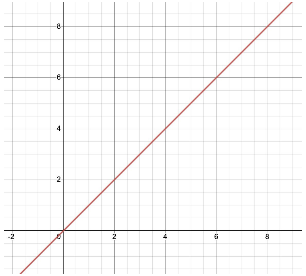


The runtime grows linearly with size of input vector

Let’s Make a Prediction

4	7	-3	6
0	1	2	3

O(n)



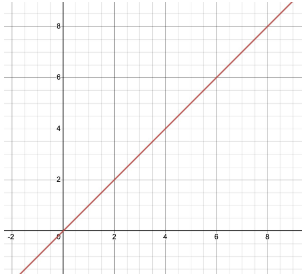
2	5	1	-10	8	3	14	2
0	1	2	3	4	5	6	7

How much longer will `vectorMax` take for a Vector of size 8, compared to size 4?

Let’s Make a Prediction

4	7	-3	6
0	1	2	3

O(n)

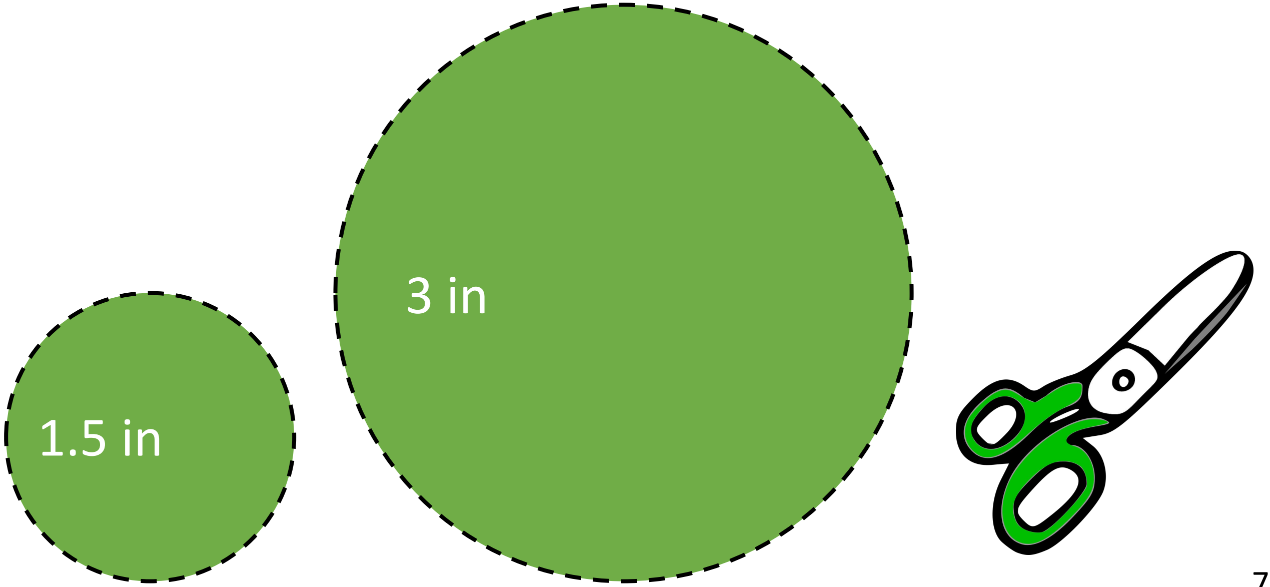


2	5	1	-10	8	3	14	2
0	1	2	3	4	5	6	7

> *Twice as long: doubling the size of the input doubles the runtime of* `vectorMax`

Other Growth Rates

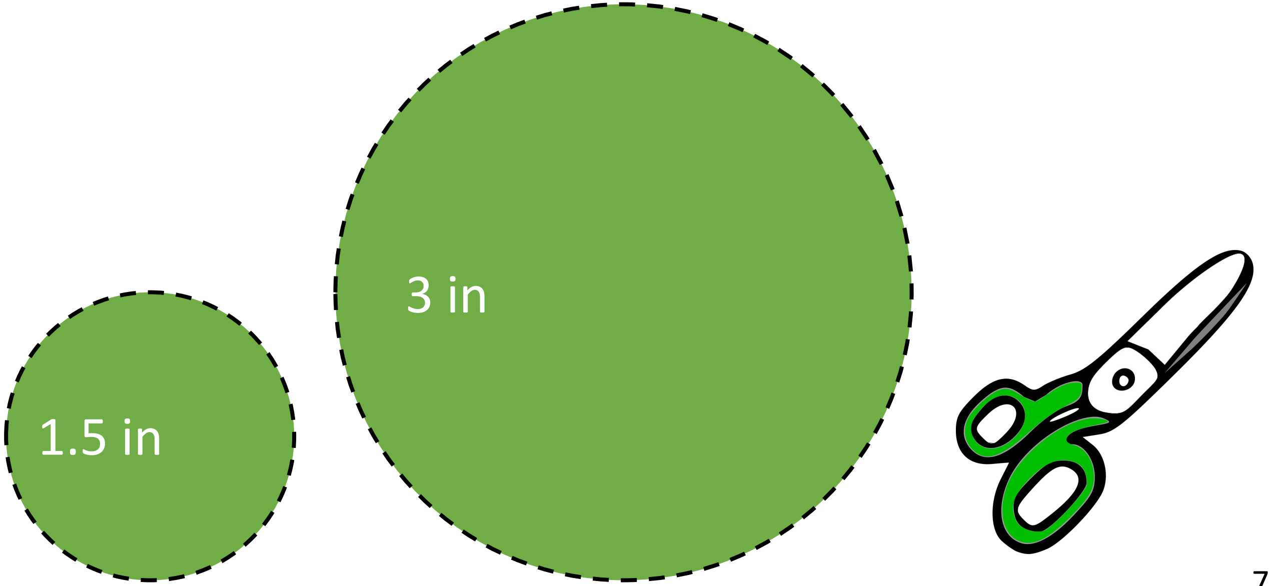
- How does a circle’s area scale with its radius?



Other Growth Rates

- How does a circle's area scale with its radius?

$$A=\pi r^2$$

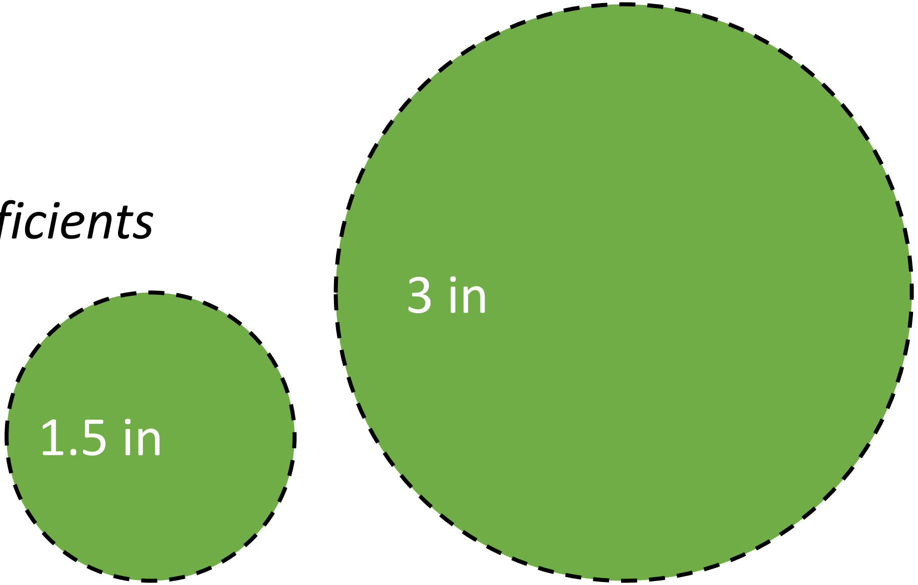


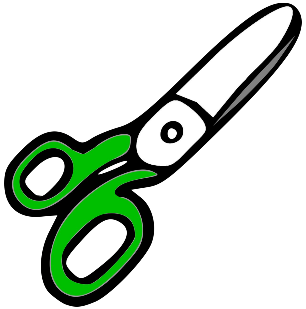
Other Growth Rates

- How does a circle’s area scale with its radius?

$A=\pi r^2$

Drop leading coefficients



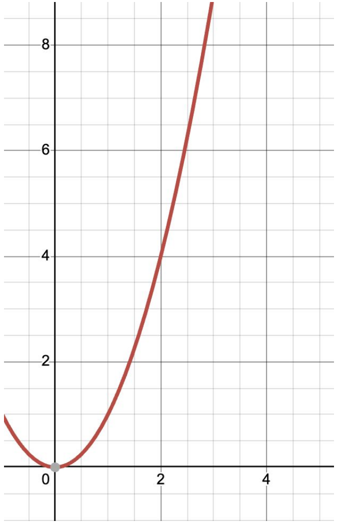


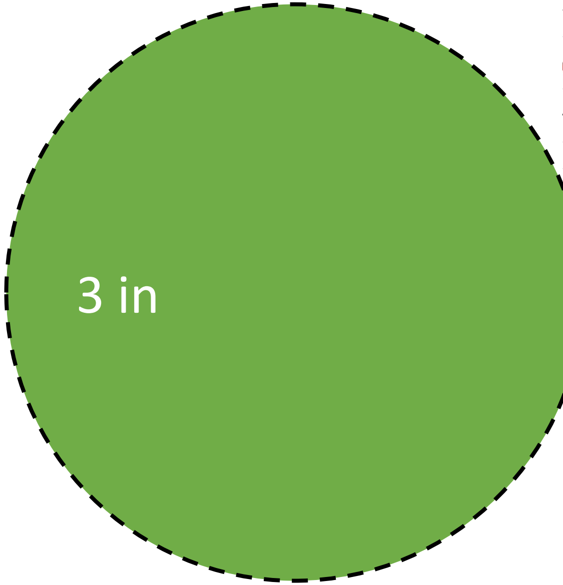
Other Growth Rates

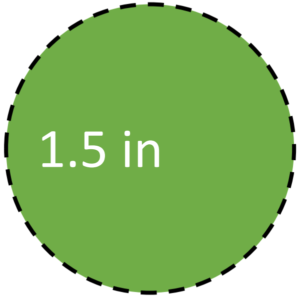
- How does a circle’s area scale with its radius?

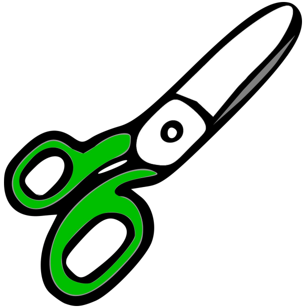
$O(n^2)$

The circle’s area grows quadratically with its radius





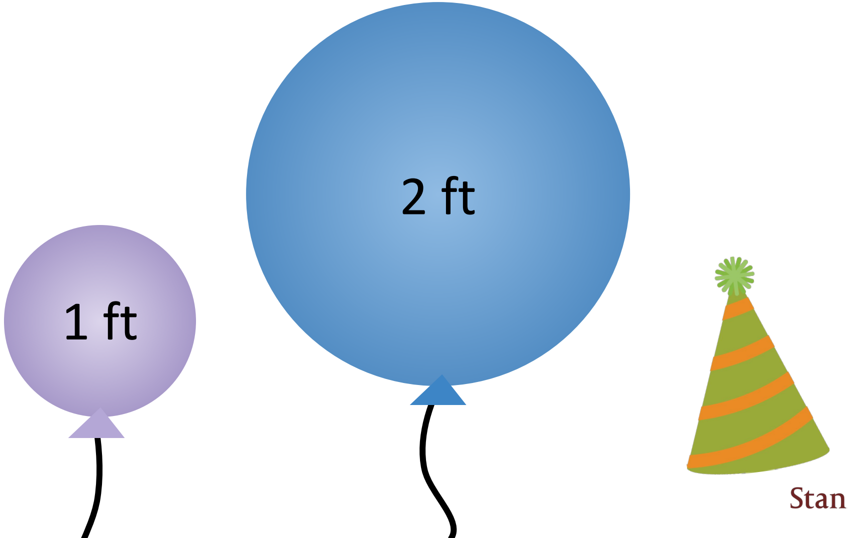




Other Growth Rates

- How does a sphere's volume scale with its radius?

$V=(4/3)\pi r^3$

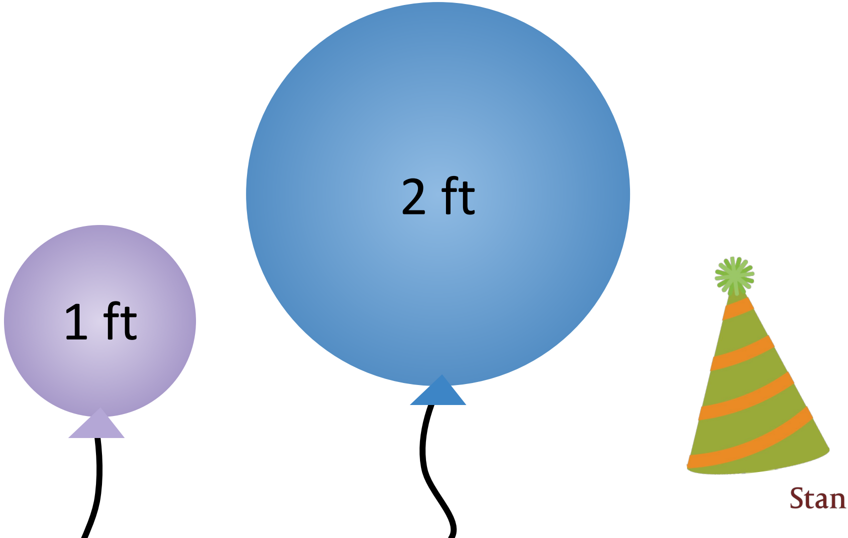


Other Growth Rates

- How does a sphere’s volume scale with its radius?

$V=(4/3)\pi r^3$

Drop leading coefficients

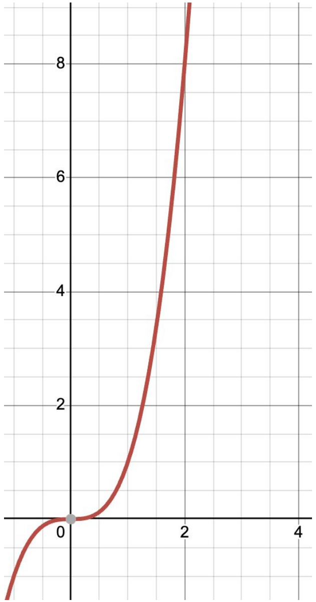


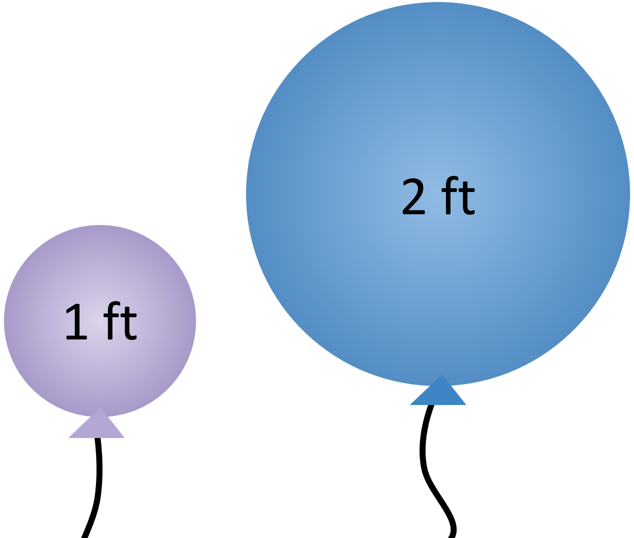
Other Growth Rates

- How does a sphere’s volume scale with its radius?

$O(n^3)$

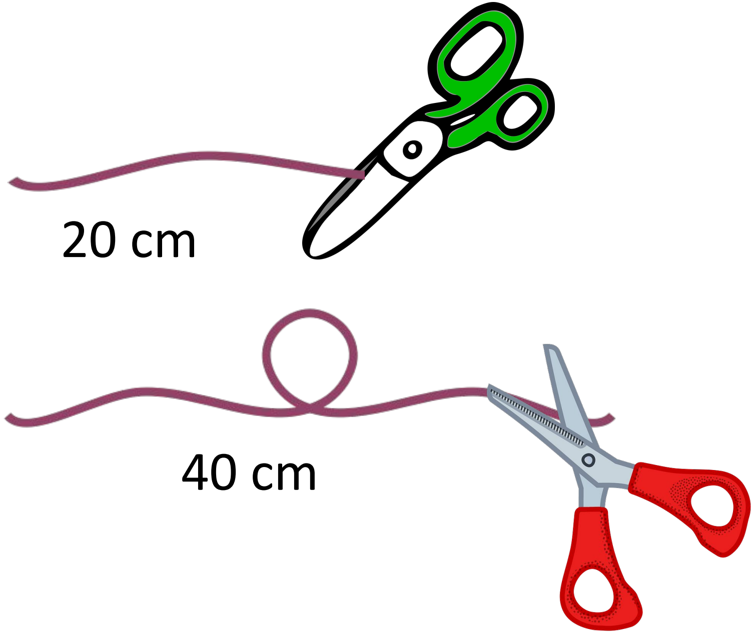
The sphere’s volume grows cubically with its radius





Other Growth Rates

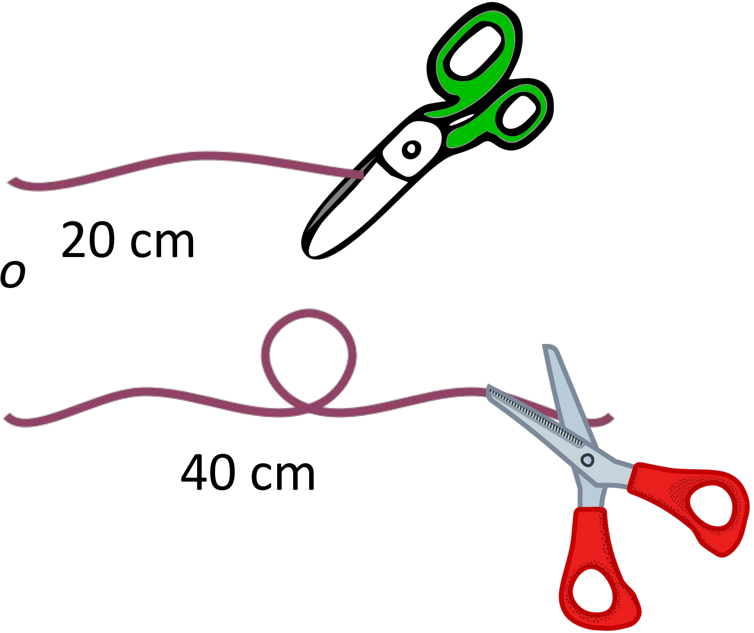
- How does the amount of effort needed to cut a piece of string scale with its length?



Other Growth Rates

- How does the amount of effort needed to cut a piece of string scale with its length?

It doesn't!
It takes the same amount of work to cut the string, no matter its length

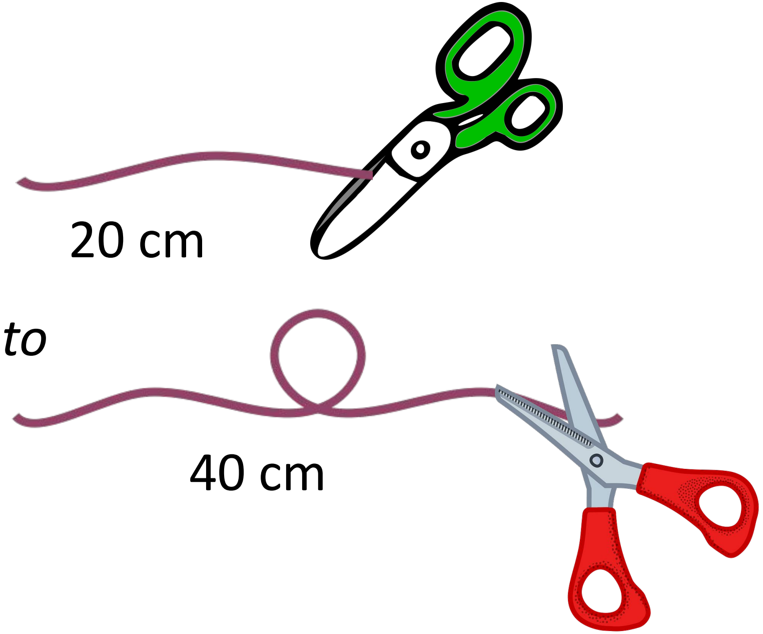


Other Growth Rates

- How does the amount of effort needed to cut a piece of string scale with its length?

O(1)

Cutting a piece of string requires a constant amount of work, relative to the string's length



Growth Rates We'll Explore

Constant	Logarithmic	Linear	n log n	Quadratic	Polynomial	Exponential
O(1)	O(log n)	O(n)	O(n log n)	$O(n^2)$	$O(n^k)$ $k \geq 1$	$O(a^n)$ $a > 1$

Big-O of ADT Operations

Vectors

- .size() – O(1)
- .add() – O(1)
- v[i] – O(1)
- .insert() – O(n)
- .remove() – O(n)
- .sublist() – O(n)
- traversal – O(n)

Grids

- .numRows() – O(1)
- .numCols() – O(1)
- grid[i][j] – O(1)
- .inBounds() – O(1)
- traversal – O(n^2)

Queues

- .size() – O(1)
- .peek() – O(1)
- .enqueue() – O(1)
- .dequeue() – O(1)
- .isEmpty() – O(1)
- traversal – O(n)

Stacks

- .size() – O(1)
- .peek() – O(1)
- .push() – O(1)
- .pop() – O(1)
- .isEmpty() – O(1)
- traversal – O(n)

Sets

- .size() –O(1)
- .isEmpty() –O(1)
- .add() –???
- .remove() –???
- .contains() –???
- traversal –O(n)

Maps

- .size() –O(1)
- .isEmpty() –O(1)
- m[key] –???
- .contains() –???
- traversal –O(n)

Big-O of ADT Operations

Vectors

- `.size() - O(1)`
- **`.add() - O(1)`**
- `v[i] - O(1)`
- `.insert() - O(n)`
- `.remove() - O(n)`
- `.sublist() - O(n)`
- `traversal - O(n)`

Grids

- `.numRows() - O(1)`
- `.numCols() - O(1)`
- `grid[i][j] - O(1)`
- `.inBounds() - O(1)`
- `traversal -` O(n^2)

Queues

- `.size() - O(1)`
- `.peek() - O(1)`
- `.enqueue() - O(1)`
- `.dequeue() - O(1)`
- `.isEmpty() - O(1)`
- `traversal - O(n)`

Stacks

- `.size() - O(1)`
- `.peek() - O(1)`
- `.push() - O(1)`
- `.pop() - O(1)`
- `.isEmpty() - O(1)`
- `traversal - O(n)`

Sets

- `.size() -O(1)`
- `.isEmpty() -O(1)`
- `.add() -???`
- `.remove() -???`
- `.contains() -???`
- `traversal -O(n)`

Maps

- `.size() -O(1)`
- `.isEmpty() -O(1)`
- `m[key] -???`
- `.contains() -???`
- `traversal -O(n)`

Big-O of ADT Operations

Vectors

- `.size() – O(1)`
- `.add() – O(1)`
- `v[i] – O(1)`
- **`.insert() – O(n)`**
- `.remove() – O(n)`
- `.sublist() – O(n)`
- `traversal – O(n)`

Grids

- `.numRows() – O(1)`
- `.numCols() – O(1)`
- `grid[i][j] – O(1)`
- `.inBounds() – O(1)`
- `traversal – O(`n^2`)`

Queues

- `.size() – O(1)`
- `.peek() – O(1)`
- `.enqueue() – O(1)`
- `.dequeue() – O(1)`
- `.isEmpty() – O(1)`
- `traversal – O(n)`

Stacks

- `.size() – O(1)`
- `.peek() – O(1)`
- `.push() – O(1)`
- `.pop() – O(1)`
- `.isEmpty() – O(1)`
- `traversal – O(n)`

Sets

- `.size() –O(1)`
- `.isEmpty() –O(1)`
- `.add() –???`
- `.remove() –???`
- `.contains() –???`
- `traversal –O(n)`

Maps

- `.size() –O(1)`
- `.isEmpty() –O(1)`
- `m[key] –???`
- `.contains() –???`
- `traversal –O(n)`

Big-O of ADT Operations

Vectors

- .size() - O(1)
- .add() - O(1)
- v[i] - O(1)
- **.insert() - O(n)**
- .remove() - O(n)
- .sublist() - O(n)
- trave

Grids

- .num
- .num
- grid[i][j] - O(1)
- .inBounds() - O(1)
- traversal - O(n^2)

Queues

- .size() - O(1)
- .peek() - O(1)
- .enqueue() - O(1)
- .dequeue() - O(1)
- .isEmpty() - O(1)

- .peek() - O(1)
- .push() - O(1)
- .pop() - O(1)
- .isEmpty() - O(1)
- traversal - O(n)

Sets

- .size() -O(1)
- .isEmpty() -O(1)
- .add() -???
- .remove() -???
- .contains() -???
- al -O(n)

-) -O(1)
- y() -O(1)
- m[key] -???
- .contains() -???
- traversal -O(n)

👥 Why does inserting into a Vector have linear time complexity? Think of the "worst case" scenario.

Big-O of ADT Operations

Vectors

- .size() - O(1)
- .add() - O(1)
- v[i] - O(1)
- **.insert() - O(n)**
- .remove() - O(n)
- .sublist() - O(n)
- trav

Grids

- .num
- .num
- grid[i][j] - O(1)
- .inBounds() - O(1)
- traversal - $O(n^2)$

Queues

- .size() - O(1)
- .peek() - O(1)
- .enqueue() - O(1)
- .dequeue() - O(1)
- .isEmpty() - O(1)

- .peek() - O(1)
- .push() - O(1)
- .pop() - O(1)
- .isEmpty() - O(1)
- traversal - O(n)

Sets

- .size() -O(1)
- .isEmpty() -O(1)
- .add() -???
- .remove() -???
- .contains() -???
- al -O(n)

-) -O(1)
- .isEmpty() -O(1)
- m[key] -???
- .contains() -???
- traversal -O(n)

In the worst case, we're inserting at the front, shifting the other n elements over by one position.

Is it Efficient?

Comparing Big-O runtimes

We’ll Use Big-O to Categorize Efficiency

Constant Time - $O(1)$

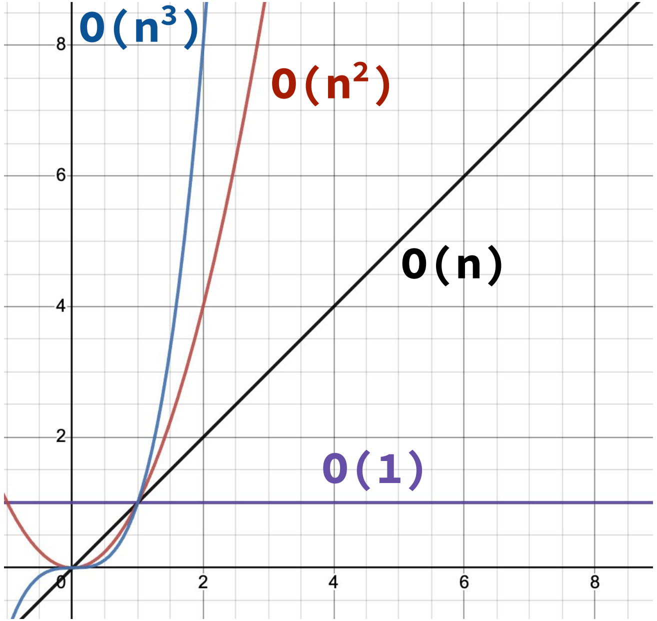
- The best we can do!
- Euclid's Algorithm for Perfect Numbers

Linear Time - $O(n)$

- This is okay, we can live with this

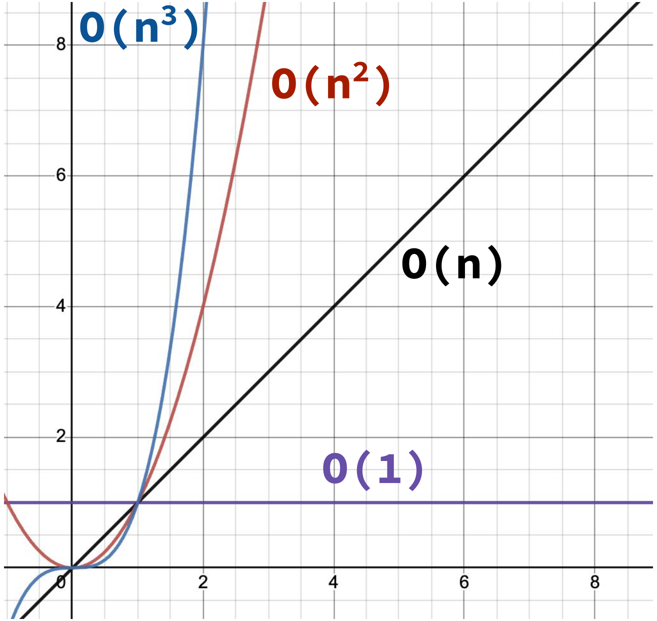
Quadratic Time - $O(n^2)$ and beyond

- This can start to slow down really quickly
- Exhaustive Search for Perfect Numbers



We'll Use Big-O to Categorize Efficiency

- Spoiler alert: not every problem is solvable in O(1) time
- We can use Big-O to compare different solutions to the same problem
- The "more efficient" solution gets the job done with a smaller Big-O



vectorMax, revisited

```
int vectorMax(Vector<int> &v) {
    for (int num: v) {
        bool seenLarger = false;
        for (int compareNum: v) {
            if (compareNum > num) {
                seenLarger = true;
            }
        }
        if (!seenLarger) {
            return num;
        }
    }
    return -1;
}
```

vectorMax, revisited

v:	4	7	-3	6
	0	1	2	3

```
int vectorMax(Vector<int> &v) {
    for (int num: v) {
        bool seenLarger = false;
        for (int compareNum: v) {
            if (compareNum > num) {
                seenLarger = true;
            }
        }
        if (!seenLarger) {
            return num;
        }
    }
    return -1;
}
```

vectorMax, revisited

```
int vectorMax(Vector<int> &v) {
    for (int num: v) {
        bool seenLarger = false;
        for (int compareNum: v) {
            if (compareNum > num) {
                seenLarger = true;
            }
        }
        if (!seenLarger) {
            return num;
        }
    }
    return -1;
}
```

v:	**4**	7	-3	6
	0	1	2	3

num: 4

vectorMax, revisited

```
int vectorMax(Vector<int> &v) {
    for (int num: v) {
        bool seenLarger = false;
        for (int compareNum: v) {
            if (compareNum > num) {
                seenLarger = true;
            }
        }
        if (!seenLarger) {
            return num;
        }
    }
    return -1;
}
```

v:

4	7	-3	6
0	1	2	3

num: 4

seenLarger: false

vectorMax, revisited

```
int vectorMax(Vector<int> &v) {
    for (int num: v) {
        bool seenLarger = false;
        for (int compareNum: v) {
            if (compareNum > num) {
                seenLarger = true;
            }
        }
        if (!seenLarger) {
            return num;
        }
    }
    return -1;
}
```

v:

v:	**4**	7	-3	6
	0	1	2	3

num: 4

seenLarger: false

compareNum: 4

vectorMax, revisited

```
int vectorMax(Vector<int> &v) {
    for (int num: v) {
        bool seenLarger = false;
        for (int compareNum: v) {
            if (compareNum > num) {
                seenLarger = true;
            }
        }
        if (!seenLarger) {
            return num;
        }
    }
    return -1;
}
```

v:

4	7	-3	6
0	1	2	3

num: **4**

seenLarger: false

compareNum: **4**

vectorMax, revisited

```
int vectorMax(Vector<int> &v) {
    for (int num: v) {
        bool seenLarger = false;
        for (int compareNum: v) {
            if (compareNum > num) {
                seenLarger = true;
            }
        }
        if (!seenLarger) {
            return num;
        }
    }
    return -1;
}
```

v:	4	**7**	-3	6
	0	1	2	3

num: 4

seenLarger: false

compareNum: **7**

vectorMax, revisited

```
int vectorMax(Vector<int> &v) {
    for (int num: v) {
        bool seenLarger = false;
        for (int compareNum: v) {
            if (compareNum > num) {
                seenLarger = true;
            }
        }
        if (!seenLarger) {
            return num;
        }
    }
    return -1;
}
```

v:

4	7	-3	6
0	1	2	3

num: **4**

seenLarger: false

compareNum: **7**

vectorMax, revisited

```
int vectorMax(Vector<int> &v) {
    for (int num: v) {
        bool seenLarger = false;
        for (int compareNum: v) {
            if (compareNum > num) {
                seenLarger = true;
            }
        }
        if (!seenLarger) {
            return num;
        }
    }
    return -1;
}
```

v:

4	7	-3	6
0	1	2	3

num: 4

seenLarger: **true**

compareNum: 7

vectorMax, revisited

```
int vectorMax(Vector<int> &v) {
    for (int num: v) {
        bool seenLarger = false;
        for (int compareNum: v) {
            if (compareNum > num) {
                seenLarger = true;
            }
        }
        if (!seenLarger) {
            return num;
        }
    }
    return -1;
}
```

v:	4	7	**-3**	6
	0	1	2	3

num: 4

seenLarger: true

compareNum: **-3**

vectorMax, revisited

```
int vectorMax(Vector<int> &v) {
    for (int num: v) {
        bool seenLarger = false;
        for (int compareNum: v) {
            if (compareNum > num) {
                seenLarger = true;
            }
        }
        if (!seenLarger) {
            return num;
        }
    }
    return -1;
}
```

v:

4	7	-3	6
0	1	2	3

num: **4**

seenLarger: true

compareNum: **-3**

vectorMax, revisited

```
int vectorMax(Vector<int> &v) {
    for (int num: v) {
        bool seenLarger = false;
        for (int compareNum: v) {
            if (compareNum > num) {
                seenLarger = true;
            }
        }
        if (!seenLarger) {
            return num;
        }
    }
    return -1;
}
```

v:	4	7	-3	**6**
	0	1	2	3

num: 4

seenLarger: true

compareNum: **6**

vectorMax, revisited

```
int vectorMax(Vector<int> &v) {
    for (int num: v) {
        bool seenLarger = false;
        for (int compareNum: v) {
            if (compareNum > num) {
                seenLarger = true;
            }
        }
        if (!seenLarger) {
            return num;
        }
    }
    return -1;
}
```

v:	4	7	-3	6
	0	1	2	3

num: **4**

seenLarger: true

compareNum: **6**

vectorMax, revisited

```
int vectorMax(Vector<int> &v) {
    for (int num: v) {
        bool seenLarger = false;
        for (int compareNum: v) {
            if (compareNum > num) {
                seenLarger = true;
            }
        }
        if (!seenLarger) {
            return num;
        }
    }
    return -1;
}
```

v:

4	7	-3	6
0	1	2	3

num: 4

seenLarger: **true**

compareNum: 6

vectorMax, revisited

```
int vectorMax(Vector<int> &v) {
    for (int num: v) {
        bool seenLarger = false;
        for (int compareNum: v) {
            if (compareNum > num) {
                seenLarger = true;
            }
        }
        if (!seenLarger) {
            return num;
        }
    }
    return -1;
}
```

v:

4	7	-3	6
0	1	2	3

num: 4

seenLarger: **true**

compareNum: 6

vectorMax, revisited

```
int vectorMax(Vector<int> &v) {
    for (int num: v) {
        bool seenLarger = false;
        for (int compareNum: v) {
            if (compareNum > num) {
                seenLarger = true;
            }
        }
        if (!seenLarger) {
            return num;
        }
    }
    return -1;
}
```

v:

4	**7**	-3	6
0	1	2	3

num: **7**

seenLarger: true

compareNum: 6

vectorMax, revisited

```
int vectorMax(Vector<int> &v) {
    for (int num: v) {
        bool seenLarger = false;
        for (int compareNum: v) {
            if (compareNum > num) {
                seenLarger = true;
            }
        }
        if (!seenLarger) {
            return num;
        }
    }
    return -1;
}
```

v:

4	7	-3	6
0	1	2	3

num: 7

seenLarger: **false**

compareNum: 6

vectorMax, revisited

```
int vectorMax(Vector<int> &v) {
    for (int num: v) {
        bool seenLarger = false;
        for (int compareNum: v) {
            if (compareNum > num) {
                seenLarger = true;
            }
        }
        if (!seenLarger) {
            return num;
        }
    }
    return -1;
}
```

v:	**4**	7	-3	6
	0	1	2	3

num: 7

seenLarger: false

compareNum: **4**

vectorMax, revisited

```
int vectorMax(Vector<int> &v) {
    for (int num: v) {
        bool seenLarger = false;
        for (int compareNum: v) {
            if (compareNum > num) {
                seenLarger = true;
            }
        }
        if (!seenLarger) {
            return num;
        }
    }
    return -1;
}
```

v:	4	7	-3	6
	0	1	2	3

num: **7**

seenLarger: false

compareNum: **4**

vectorMax, revisited

```
int vectorMax(Vector<int> &v) {
    for (int num: v) {
        bool seenLarger = false;
        for (int compareNum: v) {
            if (compareNum > num) {
                seenLarger = true;
            }
        }
        if (!seenLarger) {
            return num;
        }
    }
    return -1;
}
```

v:

4	**7**	-3	6
0	1	2	3

num: 7

seenLarger: false

compareNum: **7**

vectorMax, revisited

```
int vectorMax(Vector<int> &v) {
    for (int num: v) {
        bool seenLarger = false;
        for (int compareNum: v) {
            if (compareNum > num) {
                seenLarger = true;
            }
        }
        if (!seenLarger) {
            return num;
        }
    }
    return -1;
}
```

v:	4	7	-3	6
	0	1	2	3

num: **7**

seenLarger: false

compareNum: **7**

vectorMax, revisited

```
int vectorMax(Vector<int> &v) {
    for (int num: v) {
        bool seenLarger = false;
        for (int compareNum: v) {
            if (compareNum > num) {
                seenLarger = true;
            }
        }
        if (!seenLarger) {
            return num;
        }
    }
    return -1;
}
```

v:	4	7	**-3**	6
	0	1	2	3

num: 7

seenLarger: false

compareNum: **-3**

vectorMax, revisited

```
int vectorMax(Vector<int> &v) {
    for (int num: v) {
        bool seenLarger = false;
        for (int compareNum: v) {
            if (compareNum > num) {
                seenLarger = true;
            }
        }
        if (!seenLarger) {
            return num;
        }
    }
    return -1;
}
```

v:

4	7	-3	6
0	1	2	3

num: **7**

seenLarger: false

compareNum: **-3**

vectorMax, revisited

```
int vectorMax(Vector<int> &v) {
    for (int num: v) {
        bool seenLarger = false;
        for (int compareNum: v) {
            if (compareNum > num) {
                seenLarger = true;
            }
        }
        if (!seenLarger) {
            return num;
        }
    }
    return -1;
}
```

v:	4	7	-3	**6**
	0	1	2	3

num: 7

seenLarger: false

compareNum: **6**

vectorMax, revisited

```
int vectorMax(Vector<int> &v) {
    for (int num: v) {
        bool seenLarger = false;
        for (int compareNum: v) {
            if (compareNum > num) {
                seenLarger = true;
            }
        }
        if (!seenLarger) {
            return num;
        }
    }
    return -1;
}
```

v:	4	7	-3	6
	0	1	2	3

num: **7**

seenLarger: false

compareNum: **6**

vectorMax, revisited

```
int vectorMax(Vector<int> &v) {
    for (int num: v) {
        bool seenLarger = false;
        for (int compareNum: v) {
            if (compareNum > num) {
                seenLarger = true;
            }
        }
        if (!seenLarger) {
            return num;
        }
    }
    return -1;
}
```

v:

4	7	-3	6
0	1	2	3

num: 7

seenLarger: **false**

compareNum: 6

vectorMax, revisited

```
int vectorMax(Vector<int> &v) {
    for (int num: v) {
        bool seenLarger = false;
        for (int compareNum: v) {
            if (compareNum > num) {
                seenLarger = true;
            }
        }
        if (!seenLarger) {
            return num;
        }
    }
    return -1;
}
```

v:

4	7	-3	6
0	1	2	3

num: **7**

seenLarger: false

compareNum: 6

vectorMax, revisited

v:	4	7	-3	6
	0	1	2	3

```
int vectorMax(Vector<int> &v) {
    for (int num: v) {
        bool seenLarger = false;
        for (i
            if

            }
        }
        if (!seenLarger) {
            return num;
        }
    }
    return -1;
}
```

👥 Does this algorithm seem more or less efficient than the other one?

vectorMax, revisited

v:	4	7	−3	6
	0	1	2	3

```
int vectorMax(Vector<int> &v) {
    for (int num: v) {
        bool seenLarger = false;
        for (int compareNum: v) {
            if (compareNum > num) {
                seenLarger = true;
            }
        }
        if (!seenLarger) {
            return num;
        }
    }
    return -1;
}
```

How many operations?

vectorMax, revisited

```
int vectorMax(Vector<int> &v) {
    for (int num: v) {
        bool seenLarger = false;
        for (int compareNum: v) {
            if (compareNum > num) {
                seenLarger = true;
            }
        }
        if (!seenLarger) {
            return num;
        }
    }
    return -1;
}
```

v:	4	-3	6	7
	0	1	2	3

Big-O considers worst case runtime. What if our Vector looked like this instead?

Consider what happens if we have to loop the max number of times.

vectorMax, revisited

How many operations?

```
int vectorMax(Vector<int> &v) {
    for (int num: v) {
        bool seenLarger = false;
        for (int compareNum: v) {
            if (compareNum > num) {
                seenLarger = true;
            }
        }
        if (!seenLarger) {
            return num;
        }
    }
    return -1;
}
```

vectorMax, revisited

How many operations?

```
int vectorMax(Vector<int> &v) {
    for (int num: v) {
        bool seenLarger = false;
        for (int compareNum: v) {
            if (compareNum > num) {
                seenLarger = true;
            }
        }
        if (!seenLarger) {
            return num;
        }
    }
    return -1;
}
```

n Initialize

vectorMax, revisited

How many operations?

```
int vectorMax(Vector<int> &v) {
    for (int num: v) {
        bool seenLarger = false;
        for (int compareNum: v) {
            if (compareNum > num) {
                seenLarger = true;
            }
        }
        if (!seenLarger) {
            return num;
        }
    }
    return -1;
}
```

n Initialize

n Initialize

vectorMax, revisited

```
int vectorMax(Vector<int> &v) {
    for (int num: v) {
        bool seenLarger = false;
        for (int compareNum: v) {
            if (compareNum > num) {
                seenLarger = true;
```

```
            return num;
        }
    }
    return -1;
}
```

👥 How many times do we initialize compareNum in this function?

How many operations?

n Initialize

n Initialize

? Initialize

vectorMax, revisited

```
int vectorMax(Vector<int> &v) {
    for (int num: v) {
        bool seenLarger = false;
        for (int compareNum: v) {
            if (compareNum > num) {
                seenLarger = true;
            }
        }
        if (!seenLarger) {
            return num;
        }
    }
    return -1;
}
```

How many operations?

n Initialize

n Initialize

n^2 Initialize

`vectorMax`, revisited

How many operations?

```
int vectorMax(Vector<int> &v) {
    for (int num: v) {
        bool seenLarger = false;
        for (int compareNum: v) {
            if (compareNum > num) {
                seenLarger = true;
            }
        }
        if (!seenLarger) {
            return num;
        }
    }
    return -1;
}
```

n Initialize

n Initialize

n^2 Initialize

n^2 Compare

vectorMax, revisited

```
int vectorMax(Vector<int> &v) {
    for (int num: v) {
        bool seenLarger = false;
        for (int compareNum: v) {
            if (compareNum > num) {
                seenLarger = true;
            }
        }
        if (!seenLarger) {
            return num;
        }
    }
    return -1;
}
```

How many operations?

n Initialize

n Initialize

n^2 Initialize

n^2 Compare

n^2 Reassign

vectorMax, revisited

How many operations?

```
int vectorMax(Vector<int> &v) {
    for (int num: v) {
        bool seenLarger = false;
        for (int compareNum: v) {
            if (compareNum > num) {
                seenLarger = true;
            }
        }
        if (!seenLarger) {
            return num;
        }
    }
    return -1;
}
```

n Initialize

n Initialize

n^2 Initialize

n^2 Compare

n^2 Reassign

n Evaluate

vectorMax, revisited

```
int vectorMax(Vector<int> &v) {
    for (int num: v) {
        bool seenLarger = false;
        for (int compareNum: v) {
            if (compareNum > num) {
                seenLarger = true;
            }
        }
        if (!seenLarger) {
            return num;
        }
    }
    return -1;
}
```

How many operations?

n Initialize

n Initialize

n^2 Initialize

n^2 Compare

n^2 Reassign

n Evaluate

1 Return

vectorMax, revisited

```
int vectorMax(Vector<int> &v) {
    for (int num: v) {
        bool seenLarger = false;
        for (int compareNum: v) {
            if (compareNum > num) {
                seenLarger = true;
            }
        }
        if (!seenLarger) {
            return num;
        }
    }
    return -1;
}
```

$3n + 3n^2 + 1$ operations

`vectorMax`, revisited

```
int vectorMax(Vector<int> &v) {
    for (int num: v) {
        bool seenLarger = false;
        for (int compareNum: v) {
            if (compareNum > num) {
                seenLarger = true;
            }
        }
        if (!seenLarger) {
            return num;
        }
    }
    return -1;
}
```

$3n + 3n^2 + 1$

Remove lower order terms

vectorMax, revisited

```
int vectorMax(Vector<int> &v) {
    for (int num: v) {
        bool seenLarger = false;
        for (int compareNum: v) {
            if (compareNum > num) {
                seenLarger = true;
            }
        }
        if (!seenLarger) {
            return num;
        }
    }
    return -1;
}
```

$3n + 3n^2 + 1$

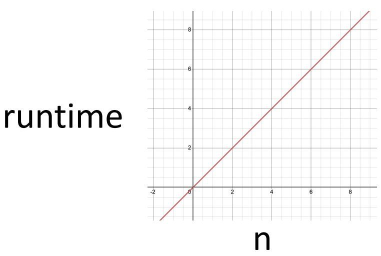
Remove leading coefficients

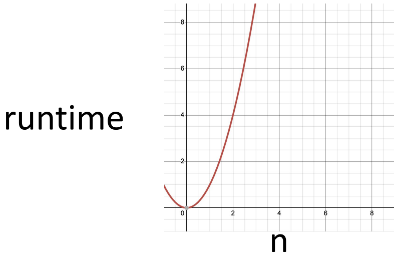
vectorMax, revisited

What's the Big-O?

$O(n^2)$

```
int vectorMax(Vector<int> &v) {
    for (int num: v) {
        bool seenLarger = false;
        for (int compareNum: v) {
            if (compareNum > num) {
                seenLarger = true;
            }
        }
        if (!seenLarger) {
            return num;
        }
    }
    return -1;
}
```



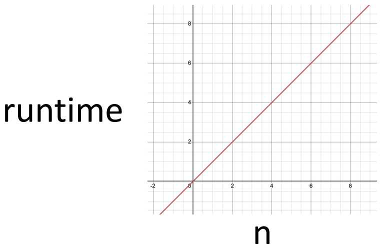


O(n)

```
int vectorMax(Vector<int> &v) {
    int currentMax = v[0];
    int n = v.size();
    for (int i = 1; i < n; i++) {
        if (currentMax < v[i]) {
            currentMax = v[i];
        }
    }
    return currentMax;
}
```

O(n^2)

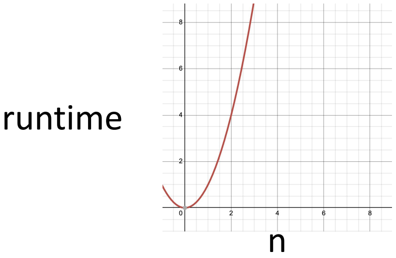
```
int vectorMax(Vector<int> &v) {
    for (int num: v) {
        bool seenLarger = false;
        for (int compareNum: v) {
            if (compareNum > num) {
                seenLarger = true;
            }
        }
        if (!seenLarger) {
            return num;
        }
    }
    return -1;
}
```



O(n)

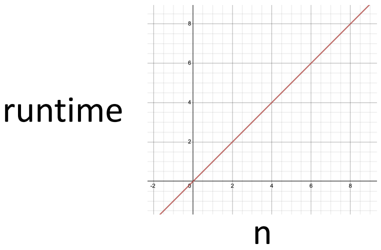
```
int vectorMax(Vector<int> &v) {
    int currentMax = v[0];
    int n = v.size();
    for (int i = 1; i < n;
        if (currentMax < v
            currentMax = v[i];
        }
    }
    return currentMax;
}
```

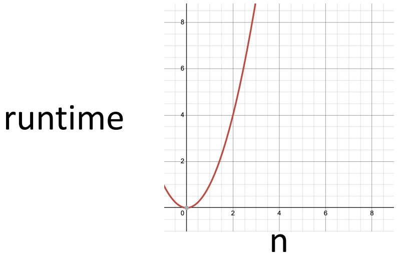
Let’s try it! 💻



O(n^2)

```
int vectorMax(Vector<int> &v) {
    for (int num: v) {
         ool seenLarger = false;
         or (int compareNum: v) {
            if (compareNum > num) {
                seenLarger = true;
            }
        }
        if (!seenLarger) {
            return num;
        }
    }
    return -1;
}
```





O(n)

```
int vectorMax(Vector<int> &v) {
    int currentMax = v[0];
```

MORE EFFICIENT 😎

```
                currentMax = v[i];
            }
    }
    return currentMax;
}
```

O(n^2)

```
int vectorMax(Vector<int> &v) {
    for (int num: v) {
        bool seenLarger = false;
        for (int compareNum: v) {
            if (compareNum > num) {
                seenLarger = true;
            }
        }
        if (!seenLarger) {
            return num;
        }
    }
    return -1;
}
```

Is it Better?

It depends…

“Better” is Subjective

Do you care about:

- Runtime?
- Memory usage?
- Code readability?

Beyond Algorithmic Analysis

Based on slides by Katie Creel

Big-O Efficiency Matters

- Consider an algorithm that runs in O(log n) time
- If it takes 10 milliseconds to process an input of size 1000…

Constant	Logarithmic	Linear	n log n	Quadratic	Polynomial	Exponential
1 ms	10 ms	1 s	10 s	17 minutes	277 hours	Heat death of the universe

Algorithmic efficiency can be the difference between a program that runs in a few seconds and one that won't finish before the heat death of the universe

Green Computing

- Computation requires energy

BAY AREA

Stanford power outage: University preparing for a restoration that could ‘take days’

Annie Vainshtein

June 22, 2022 | Updated: June 22, 2022 6:36 p.m.

BUSINESS • GOOGLE

The Secret Cost of Google's Data Centers: Billions of Gallons of Water to Cool Servers

Bitcoin consumes 'more electricity than Argentina'

By Cristina Criddle

Technology reporter

10 February 2021

Green Computing

- Computation requires energy
- "Green computing": a commitment to decreasing the environmental impact of computing
  - Decreasing carbon footprint of data centers
  - Recycling and reducing use of raw materials during manufacturing
  - Reducing energy consumption of computation itself, including by increasing algorithmic efficiency!

But Efficiency Isn’t Everything…

Case Study: Indiana Welfare Modernization

- In 2006, State of Indiana pays IBM $1b to modernize welfare management system
- 19 months later, the system is failing:
  - Welfare applicants waited 20-30 minutes on hold, only to be denied benefits after their limited cell phone minutes were used up
  - Households receiving food stamps in some counties went down by 7%, while requests for food assistance in Indiana had increased by 4%

But Efficiency Isn’t Everything…

Case Study: Indiana Welfare Modernization

- The State of Indiana canceled its contract with IBM and sued IBM for breach of contract
- IBM argued that it was not responsible; the contract only stated that a successful system would **increase efficiency** and **reduce costs**
  - IBM’s system *did* reduce costs, but it denied Indiana residents the benefits they needed

But Efficiency Isn’t Everything…

Case Study: Indiana Welfare Modernization

- The State of Indiana canceled its contract

👥 Were the engineers at IBM responsible for considering the social impacts of the system they designed?

- IBM argued that it was not responsible; the contract only stated that a successful system would **increase efficiency** and **reduce costs**
    - IBM’s system *did* reduce costs, but it denied Indiana residents the benefits they needed

But Efficiency Isn't Everything…

Case Study: Password Encryption

- What prevents a hacker from guessing passwords randomly, perhaps millions of times per minute, until they guess correctly?

  ~ Algorithmic Inefficiency ~

- `bcrypt` and other popular encryption functions are intentionally designed to be slow, memory intensive, or both, making guessing more costly

Beyond Algorithmic Analysis

- As programmers, we make choices about what to optimize for
- Efficiency can be incredibly important, but it's not everything
- Carefully consider what you want to prioritize when you design a system; in real life, there's rarely a right answer

Recap

- ADTs and Assignment 2 preview
- Attempting to measure program speed
  - Runtime → # operations → big-O
- Introducing big-O
  - How to calculate big-O
  - Common big-O classes
- Beyond algorithmic efficiency
  - Why efficiency is important
  - Why efficiency isn’t everything

Have a great weekend! ☀️