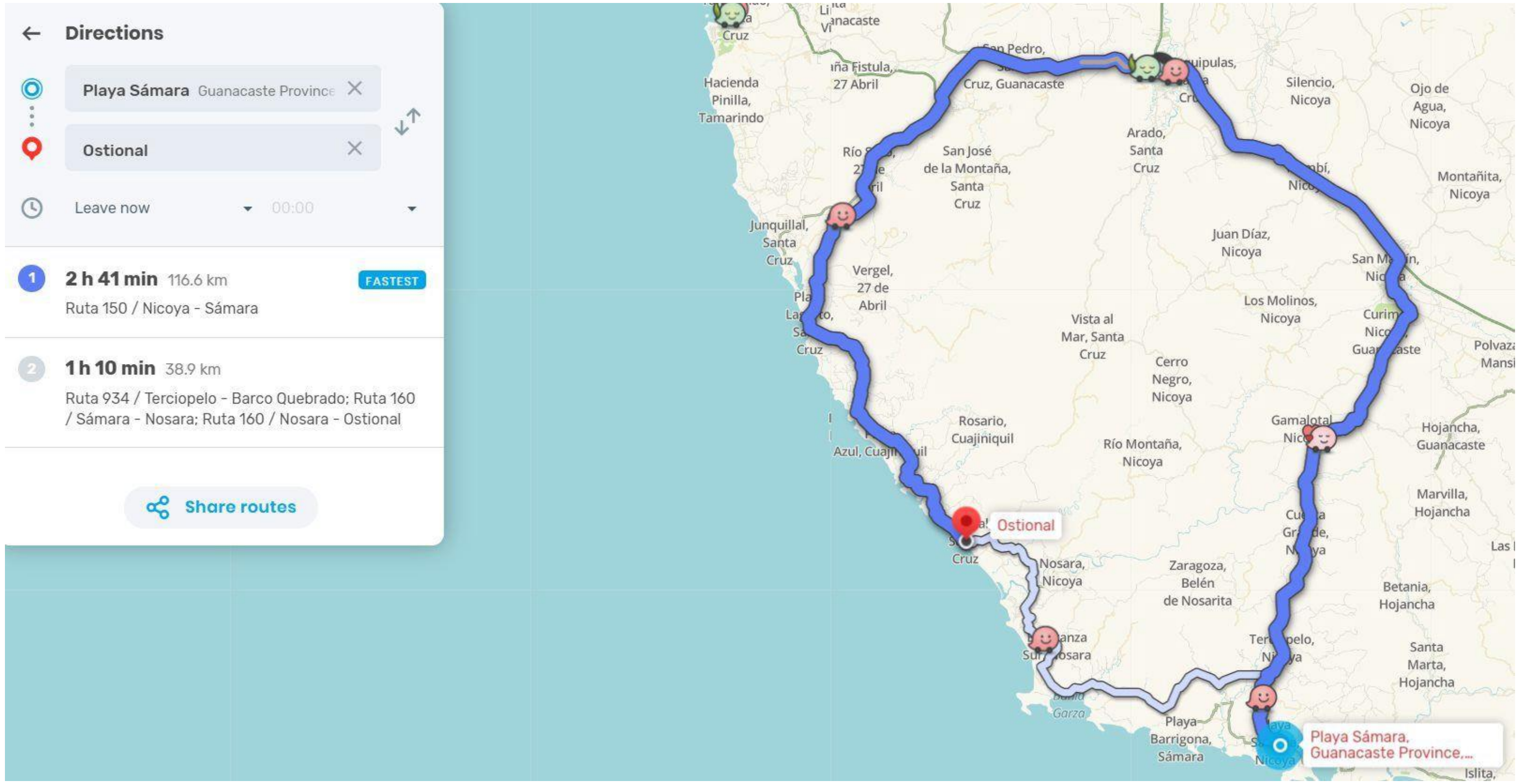
Complexity (Big O)

Using materials from Stanford CS 106B Summer 2022
(Instructors: Jenny Han and Kylie Jue)

How can we formalize the notion of efficiency for algorithms?

Why do we care about efficiency?

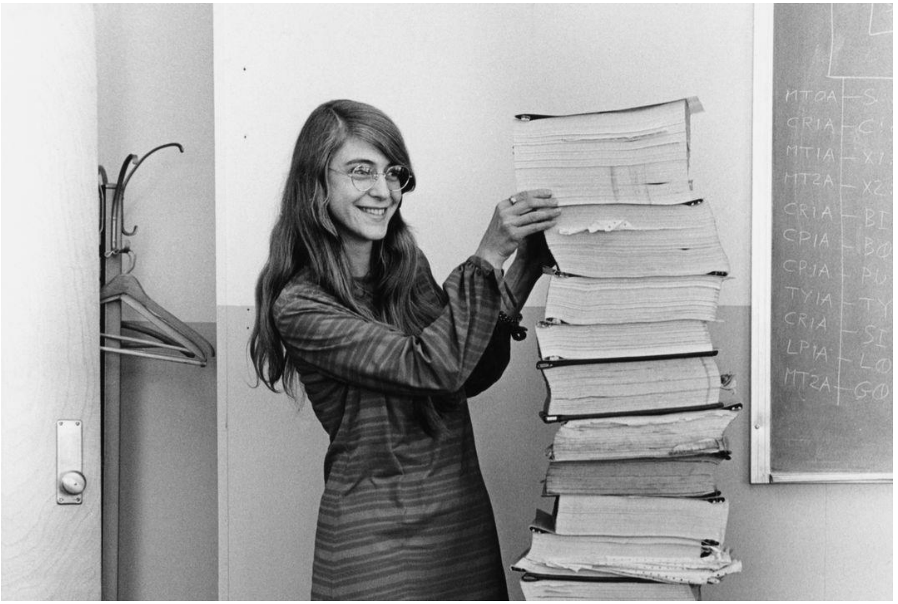
- Implementing inefficient algorithms may make solving certain tasks impossible, even with unlimited resources



Why do we care about efficiency?

- Implementing inefficient algorithms may make solving certain tasks impossible, even with unlimited resources

- Implementing efficient algorithms allows us to solve important problems, often with limited resources available



Questions programmers ask:

1. Does it work?
2. Is it fast?

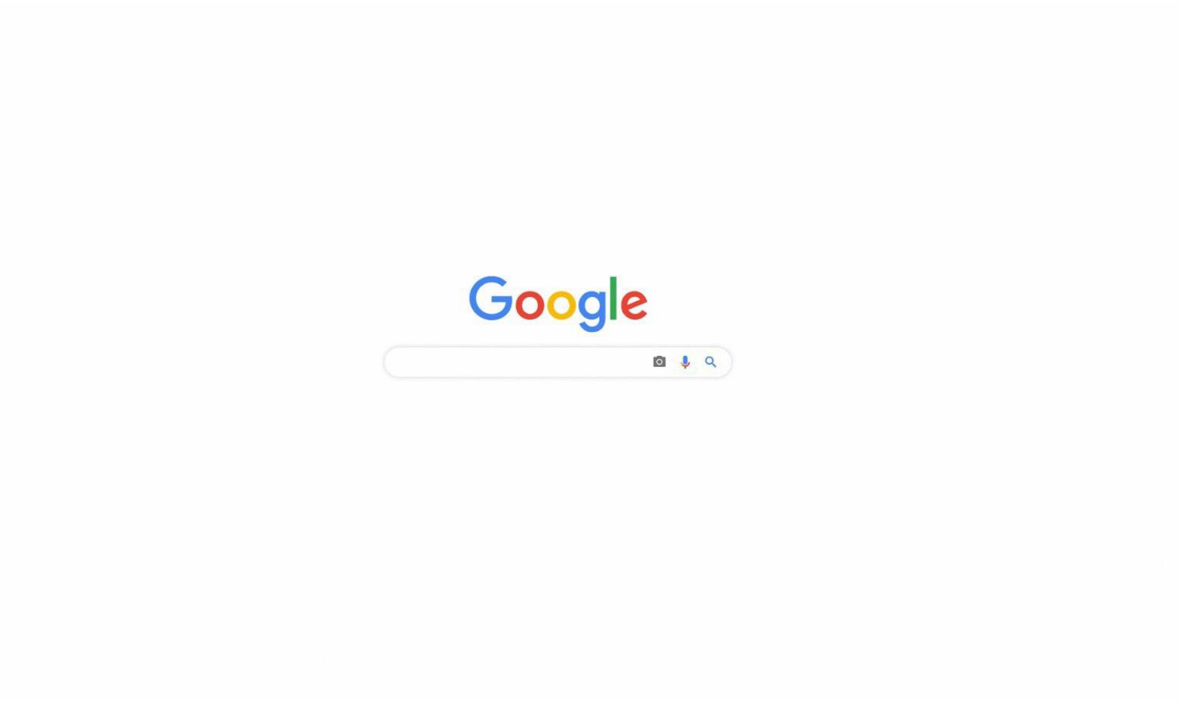
Credit: CS161

Why do programmers care about efficiency?

We solve problems at scale.

Google Search

3.8 million searches per minute



Why do we care about efficiency?

- Implementing inefficient algorithms may make solving certain tasks impossible, even with unlimited resources

- Implementing efficient algorithms allows us to solve important problems, often with limited resources available

- If we can quantify the efficiency of an algorithm, we can understand and predict its behavior when we apply it to unseen problems

Big-O Notation

Big-O Notation

- **Big-O notation** is a way of quantifying the rate at which some quantity grows.

Big-O Notation

- **Big-O notation** is a way of quantifying the rate at which some quantity grows.
- Example:
  - A square of side length `r` has area $O(r^2)$.

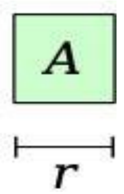
Big-O Notation

- **Big-O notation** is a way of quantifying the rate at which some quantity grows.
- Example:
  - A square of side length $\mathbf{r}$ has area $\mathtt{O}(\mathbf{r}^2)$.

*The "O" stands for "on the order of", which is **a growth prediction**, not an exact formula*

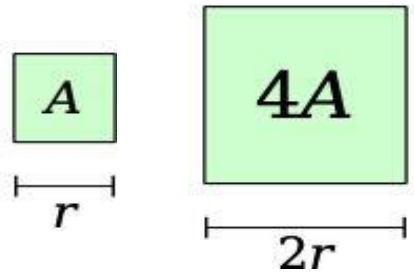
Big-O Notation

- **Big-O notation** is a way of quantifying the rate at which some quantity grows.
- Example:
  - A square of side length r has area $O(r^2)$.



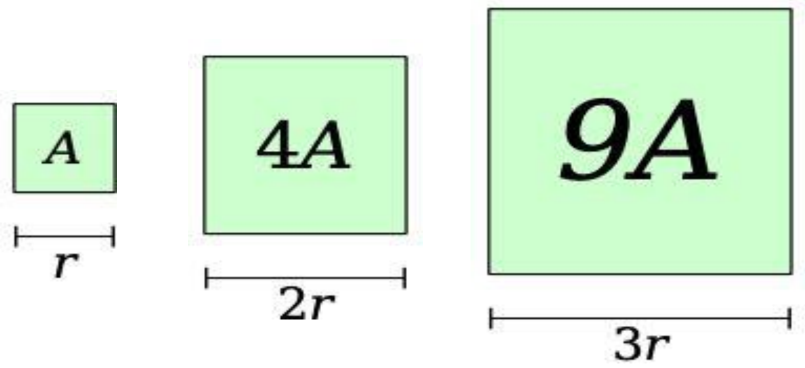
Big-O Notation

- **Big-O notation** is a way of quantifying the rate at which some quantity grows.
- Example:
  - A square of side length `r` has area `O(r²)`.



Big-O Notation

- **Big-O notation** is a way of quantifying the rate at which some quantity grows.
- Example:
  - A square of side length `r` has area `O(r²)`.

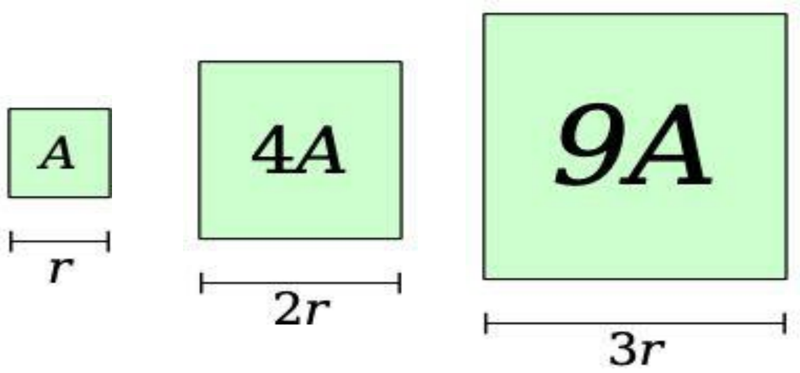


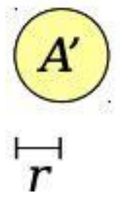
Doubling r increases area 4x

Tripling r increases area 9x

Big-O Notation

- **Big-O notation** is a way of quantifying the rate at which some quantity grows.
- Example:
  - A square of side length `r` has area `O(r²)`.
  - A circle of radius `r` has area `O(r²)`.



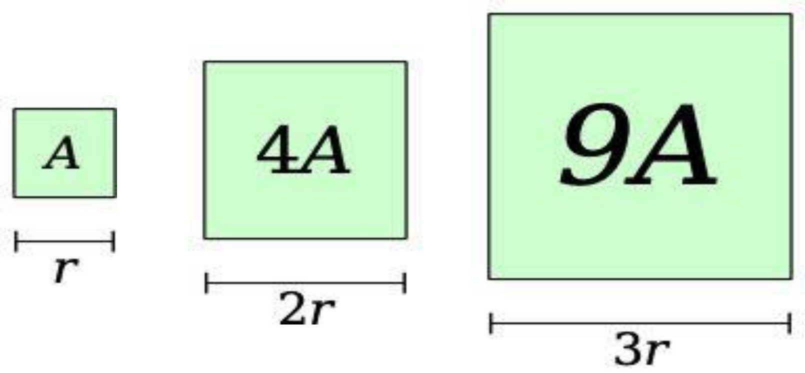


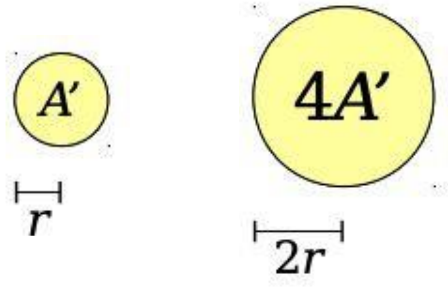
Doubling r increases area 4x

Tripling r increases area 9x

Big-O Notation

- **Big-O notation** is a way of quantifying the rate at which some quantity grows.
- Example:
  - A square of side length `r` has area `O(r²)`.
  - A circle of radius `r` has area `O(r²)`.



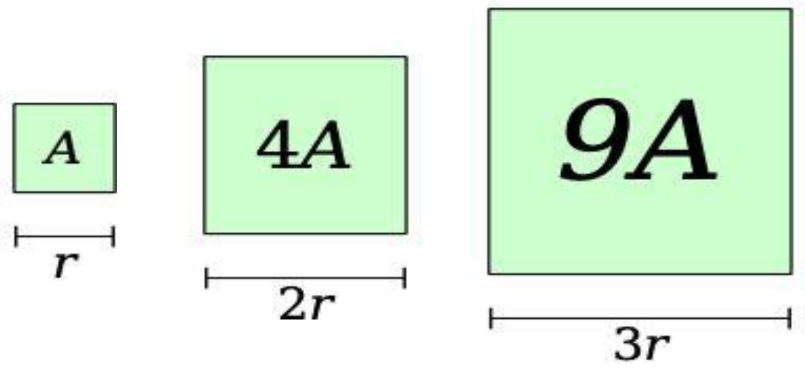


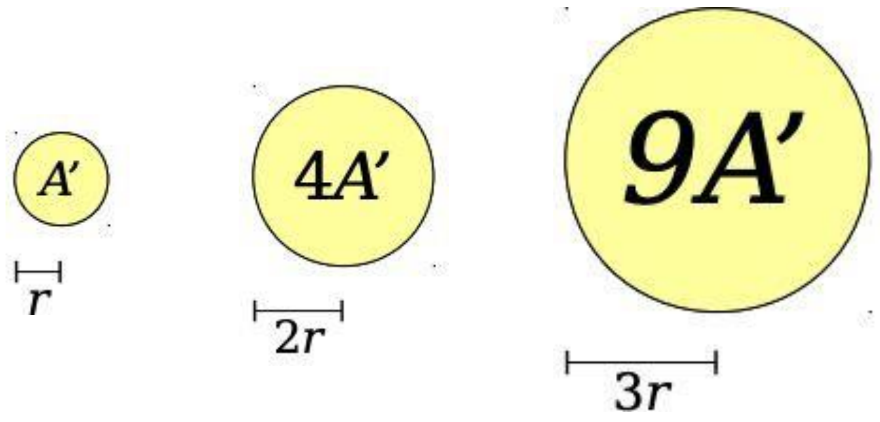
Doubling r increases area 4x

Tripling r increases area 9x

Big-O Notation

- **Big-O notation** is a way of quantifying the rate at which some quantity grows.
- Example:
  - A square of side length `r` has area $O(r^2)$.
  - A circle of radius `r` has area $O(r^2)$.



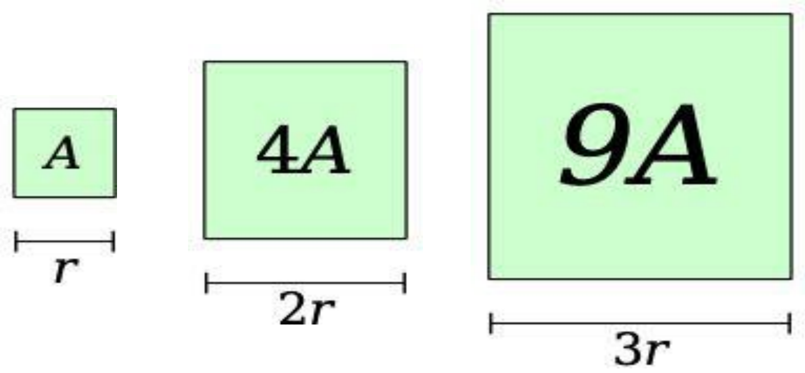


Doubling r increases area 4x

Tripling r increases area 9x

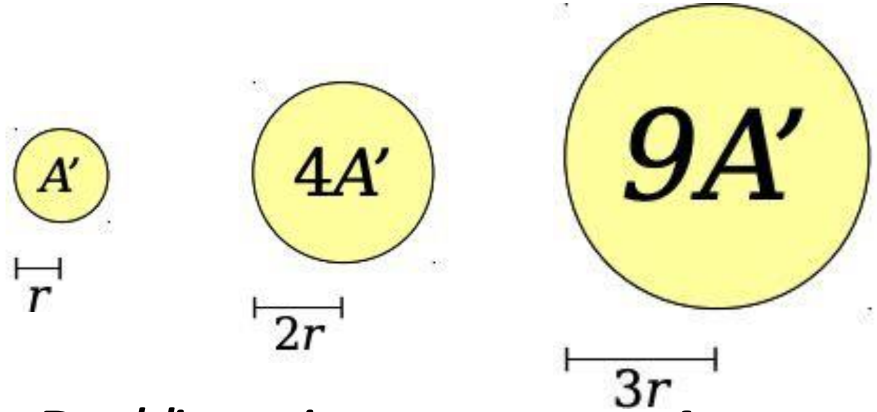
Big-O Notation

- **Big-O notation** is a way of quantifying the rate at which some quantity grows.
- Example:
  - A square of side length `r` has area `O(r²)`.
  - A circle of radius `r` has area `O(r²)`.



Doubling r increases area 4x

Tripling r increases area 9x



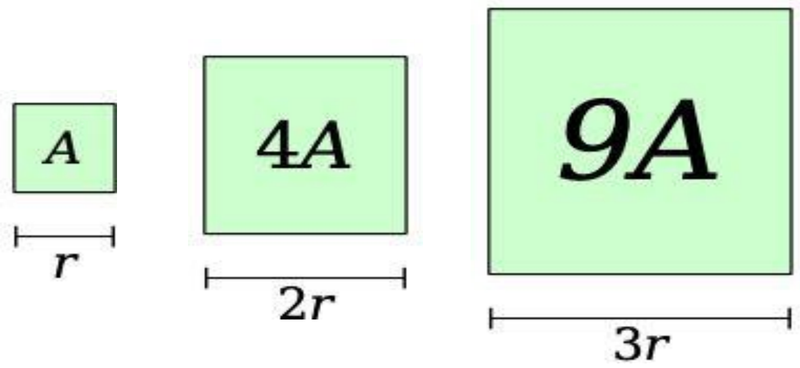
Doubling r increases area 4x

Tripling r increases area 9x

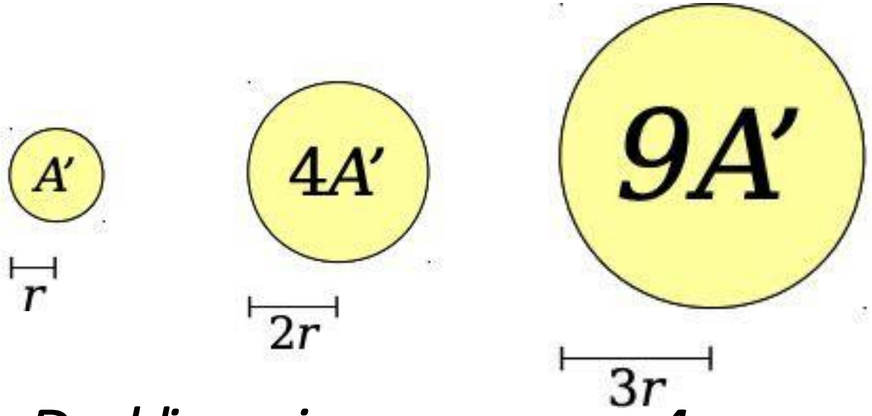
Big-O Notation

- **Big-O notation** is a way of quantifying the rate at which some quantity grows.
- Example:
  - A square of side length r has area $O(r^2)$.
  - A circle of radius r has area $O(r^2)$.

This just says that these quantities grow at the same relative rates. It does not say that they're equal!



Doubling r increases area 4x
Tripling r increases area 9x



Doubling r increases area 4x
Tripling r increases area 9x

Big-O in the Real World

Big-O Example: Network Value

- Metcalfe’s Law
  - The value of a communications network with **`n`** users is $O(n^2)$.

Big-O Example: Network Value

- Metcalfe’s Law
  - The value of a communications network with **`n`** users is **`O(n`**^{2}**`)`**.
- Imagine a social network has 10,000,000 users and is worth $10,000,000. Estimate how many users it needs to have to be worth $1,000,000,000.
- **Reasonable guess:** The network needs to grow its value 100×. Since value grows quadratically with size, it needs to grow its user base 10×, requiring 100,000,000 users.

Big-O Example: Cell Size

- Question: Why are cells tiny?

Big-O Example: Cell Size

- Question: Why are cells tiny?
- Assumption: Cells are spheres

Big-O Example: Cell Size

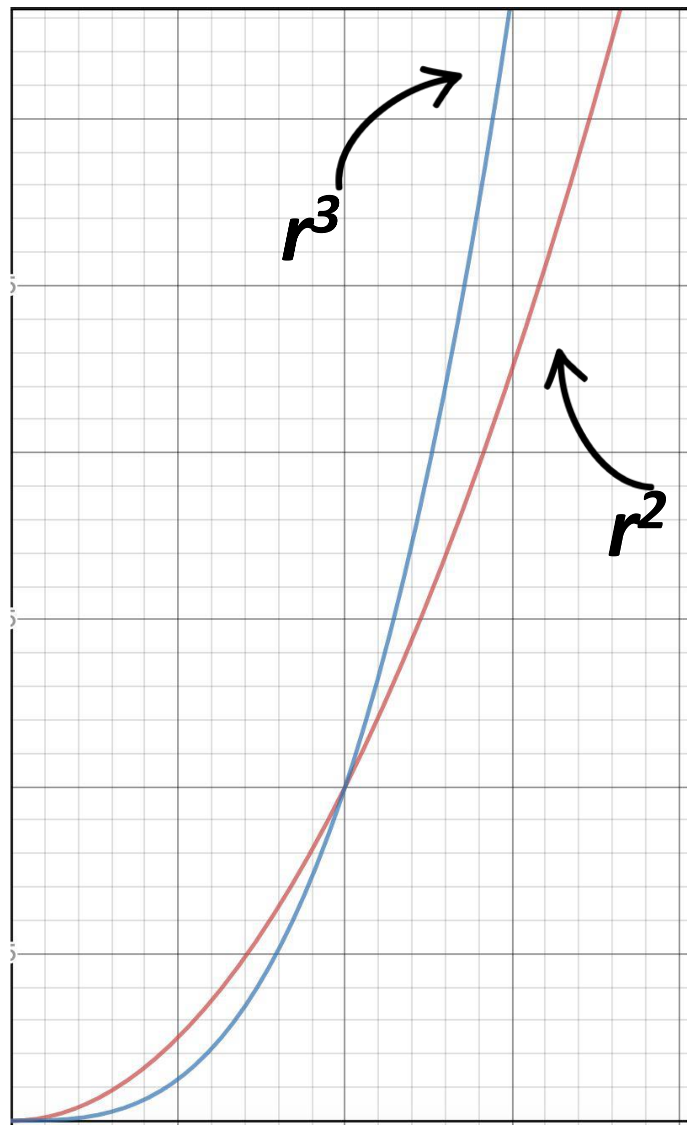
- Question: Why are cells tiny?
- Assumption: Cells are spheres
- A cell absorbs nutrients from its environment through its surface area.
    - Surface area of the cell: $O(r^2)$

Big-O Example: Cell Size

- Question: Why are cells tiny?
- Assumption: Cells are spheres
- A cell absorbs nutrients from its environment through its surface area.
  - Surface area of the cell: $O(r^2)$
- A cell needs to provide nutrients all throughout its volume
  - Volume of the cell: $O(r^3)$

Big-O Example: Cell Size

- Question: Why are cells tiny?
- Assumption: Cells are spheres
- A cell absorbs nutrients from its environment through its surface area.
    - Surface area of the cell: $O(r^2)$
- A cell needs to provide nutrients all throughout its volume
    - Volume of the cell: $O(r^3)$
- As a cell gets bigger, its resource *intake* grows slower than its resource *consumption*, so each part of the cell gets less energy.



Big-O Example: Manufacturing

- You’re working at a company producing cat toys. It costs you some amount of money to produce a cat toy, and there was some one-time cost to set up the factory.
- What data would you need to gather to estimate the cost of producing ten million cat toys?

Big-O Example: Manufacturing

- You're working at a company producing cat toys. It costs you some amount of money to produce a cat toy, and there was some one-time cost to set up the factory.
- What data would you need to gather to estimate the cost of producing ten million cat toys?

```
Cost(n) = n × costPerToy + startupCost
```

Big-O Example: Manufacturing

- You're working at a company producing cat toys. It costs you some amount of money to produce a cat toy, and there was some one-time cost to set up the factory.
- What data would you need to gather to estimate the cost of producing ten million cat toys?

This term grows as a function of n

```
Cost(n) = n × costPerToy + startupCost
```

Big-O Example: Manufacturing

- You're working at a company producing cat toys. It costs you some amount of money to produce a cat toy, and there was some one-time cost to set up the factory.
- What data would you need to gather to estimate the cost of producing ten million cat toys?

This term grows as a function of n

This term does not grow

`Cost(n) = n × costPerToy + startupCost`

Big-O Example: Manufacturing

- You’re working at a company producing cat toys. It costs you some amount of money to produce a cat toy, and there was some one-time cost to set up the factory.
- What data would you need to gather to estimate the cost of producing ten million cat toys?

This term grows as a function of n

This term does not grow

```
Cost(n) = n × costPerToy + startupCost
        = O(n)
```

Trick to calculating Big-O

Throw out all the leading coefficients and lower-order terms (including constants).

```
Cost(n) = $2 × n + $500
```

$$\cancel{\$2 \times} \ n \ \cancel{+ \ \$500}$$

```
Cost(n) = O(n)
```

Nuances of Big-O

- Big-O notation is designed to capture **the rate at which a quantity grows.** It <u>does not</u> capture information about
    - leading coefficients: the area of a square and a circle are both $O(r^2)$.
    - lower-order terms: there may be other factors contributing to growth that get glossed over.

- However, it's still a **very powerful tool for predicting behavior.**

- For processes that can have different outcomes, depending on the input, Big-O represents the rate for the **worst case** (i.e. largest possible rate)

Analyzing Code

How can we apply Big-O to analyze code?

Answering “is it fast?”

- We could use runtime
  - Runtime is the amount of time it takes for a program to run

Answering “is it fast?”

- What is runtime?
  - Runtime is the amount of time it takes for a program to run

```
[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
You passed 1 of 1 tests. Keep it up!
```

Old 2012
MacBook

Why runtime isn’t enough

- What is runtime?
    - Runtime is the amount of time it takes for a program to run

```
[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
You passed 1 of 1 tests. Keep it up!
```

Old computer

```
[SimpleTest] ---- Tests from main.cpp -----
[SimpleTest] starting (PROVIDED_TEST, line  36) timing vectorMax on 20,00...  =  Correct
    Line 42 Time vectorMax(v) (size =10000000) completed in    0.181 secs
    Line 43 Time vectorMax(v) (size =10000000) completed in    0.181 secs
    Line 44 Time vectorMax(v) (size =10000000) completed in    0.183 secs
You passed 1 of 1 tests. Que bien!
```

New computer

Why runtime isn’t enough

- Measuring wall-clock runtime is less than ideal, since
  - It depends on what computer you’re using,
  - What else is running on that computer,
  - Whether that computer is conserving power,
  - Etc.

It’s very hard to standardize.

Why runtime isn't enough

- Measuring wall-clock runtime is less than ideal, since
    - It depends on what computer you're using,
    - What else is running on that computer,
    - Whether that computer is conserving power,
    - Etc.

- Worse, **individual runtimes can't predict future runtimes**.

Answering “Is it fast?”

- We need a standardized way to think about rate of algorithms
- That doesn’t make assumptions about our computer, our circumstances, our inputs, etc.

Answering “Is it fast?”

- We need a standardized way to think about rate of algorithms
- That doesn’t make assumptions about our computer, our circumstances, our inputs, etc.

Idea: count the number of executions in an algorithm.

- number of times a single operation is done (access an element, compare two items)
- We can analyze this before we even run the program!

Analyzing Code: **`vectorMax()`**

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;
}
```

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;
 }
```

Assume any individual statement takes one unit of time to execute.

If the input **`vector`** *has* **`n`** *elements, how many executions (time units) will this code take to run?*

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;
}
```

Total time based on # of repetitions

1 time unit

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;
}
```

Total time based on # of repetitions

1 time unit

1 time unit

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;
}
```

Total time based on # of repetitions

1 time unit

1 time unit

1 time unit

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;
}
```

Total time based on # of repetitions

1 time unit

1 time unit

1 time unit

N time units

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;
}
```

Total time based on # of repetitions

1 time unit

1 time unit

1 time unit

N time units

N-1 time units

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;
}
```

Total time based on # of repetitions

1 time unit
1 time unit
1 time unit
N time units
N-1 time units
N-1 time units

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;
}
```

Total time based on # of repetitions

1 time unit
1 time unit
1 time unit
N time units
N-1 time units
N-1 time units
(up to) N-1 time units

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;
}
```

Total time based on # of repetitions

1 time unit

1 time unit

1 time unit

N time units

N-1 time units

N-1 time units

(up to) N-1 time units

1 time unit

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;
}
```

Total amount of time

$4N + 1$

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;
}
```

Total amount of time

$4N + 1$

Is this useful?

What does this tell us?

Answering “Is it fast?”

- We need a standardized way to think about rate of algorithms
- That doesn’t make assumptions about our computer, our circumstances, our inputs, etc.

Idea: count the number of executions in an algorithm.

- Maybe this is still too much detail
- Constant factors might still depend on the system

Answering “Is it fast?”

- We need a standardized way to think about rate of algorithms
- That doesn’t make assumptions about our computer, our circumstances, our inputs, etc.

Better idea: find the Big-O of this algorithm.

- General enough to help us compare across computers
- **It’s a rate that represents:** As the input size grows, how does the runtime grow?
- A computer-independent metric for efficiency!

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;
}
```

Total amount of time

$4N + 1$

Is this useful?

What does this tell us?

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;
 }
```

Total amount of time

O(n)

More practical: Doubling the size of the input roughly doubles the runtime. Therefore, the input and runtime have a linear (O(n)) relationship.

Analyzing Code: `printStars()`

printStars()

```
void printStars(int n) {
    for  (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            cout << '*' << endl;
        }
    }
}
```

How much time will it take for this code to run, as a function of n? Answer using big-O notation.

printStars()

```
void printStars(int n) {
    for  (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            cout << '*' << endl;
        }
    }
}
```

How much time will it take for this code to run, as a function of n?
Answer using big-O notation.

printStars()

```
void printStars(int n) {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            // do a fixed amount of work
        }
    }
}
```

How much time will it take for this code to run, as a function of n?
Answer using big-O notation.

printStars()

```
void printStars(int n) {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            // do a fixed amount of work
        }
    }
}
```

How much time will it take for this code to run, as a function of n?
Answer using big-O notation.

printStars()

```
void printStars(int n) {
    for (int i = 0; i < n; i++) {

        // do O(n) time units of work

    }
}
```

How much time will it take for this code to run, as a function of n?
Answer using big-O notation.

printStars()

```
void printStars(int n) {
    for (int i = 0; i < n; i++) {

        // do O(n) time units of work

    }
}
```

How much time will it take for this code to run, as a function of n? Answer using big-O notation.

printStars()

```
void printStars(int n) {

    // do O(n²) time units of work

}
```

How much time will it take for this code to run, as a function of n?
Answer using big-O notation.

printStars()

```
void printStars(int n) {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            cout << '*' << endl;
        }
    }
}
```

$O(n^2)$

hmmThatsStrange()

```
void hmmThatsStrange(int n) {
    cout << "Mirth and Whimsy" << n << endl;
}
```

The runtime is completely independent of the value **n**.

hmmThatsStrange()

```
void hmmThatsStrange(int n) {
    cout << "Mirth and Whimsy" << n << endl;
}
```

How much time will it take for this code to run, as a function of n?
Answer using big-O notation.

hmmThatsStrange()

```
void hmmThatsStrange(int n) {
    cout << "Mirth and Whimsy" << n << endl;
}
```

O(1)

Example: One Loop

```
//return true if vec contains a
bool contains(const vector<int>& vec, const int a){
    for(int i = 0; i < vec.size(); ++i){
        if(vec[i] == a){
            return true;
        }
    }
    return false;
}
```

Complexity?

(A) O(1)

(B) O(log n)

(C) O(n)

(D) $O(n^2)$

Example: Two Loops in Sequence

```
//return true if either vec1 or vec2 contain a
bool contains2(const vector<int>& vec1, const vector<int>& vec2, const int a){
    for(int i = 0; i < vec1.size(); ++i){
        if(vec1[i] == a){
            return true;
        }
    }
    for(int i = 0; i < vec2.size(); ++i){
        if(vec2[i] == a){
            return true;
        }
    }
    return false;
}
```

Complexity?

(A) O(1)

(B) O(log n)

(C) O(n)

(D) O(n^2)

Example: Nested Loops

```
//return true if vec1 or vec2 have a number in common
bool numberInCommon(const vector<int>& vec1, const vector<int>& vec2){
    for(int i = 0; i < vec1.size(); ++i){
        for(int j = 0; j < vec2.size(); ++j){
            if(vec1[i] == vec2[j]){
                return true;
            }
        }
    }
    return false;
}
```

Complexity?

(A) O(1)

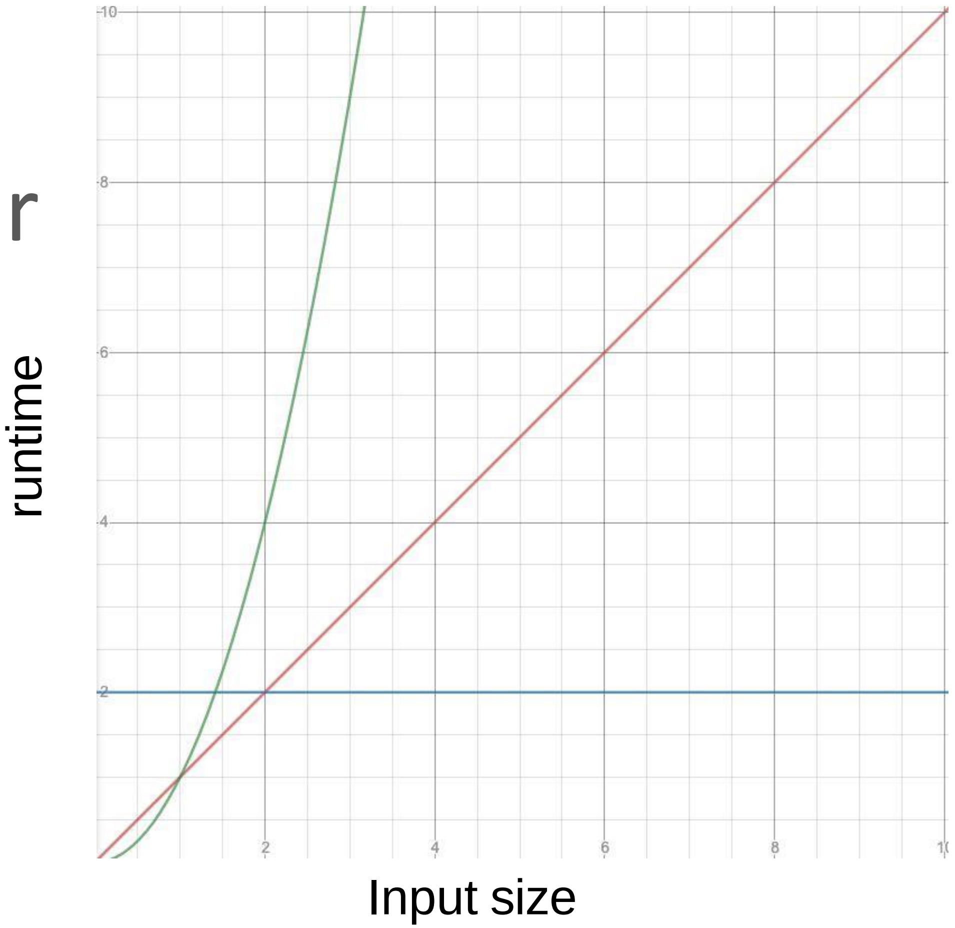
(B) O(log n)

(C) O(n)

(D) O(n^2)

Efficiency Categorizations So Far

- Constant Time – O(1)
  - Super fast, this is the best we can hope for!
- Linear Time – O(n)
  - This is okay, we can live with this
- Quadratic Time – $O(n^2)$
  - This can start to slow down really quickly



Big-O Terminology

constant	logarithmic	linear	n log n	quadratic	polynomial (other than n^2)	exponential
$O(1)$	$O(\log n)$	$O(n)$	$O(n \log n)$	$O(n^2)$	$O(n^k)$ $(k \geq 1)$	$O(a^n)$ $(a > 1)$

A fast algorithm is when the worst-case run-time grows **SLOWLY** with the input size.

Ramifications of Big O Differences

- If we have an algorithm that has 1000 elements, and the O(log n) version runs in 10 milliseconds…

constant	logarithmic	linear	n log n	quadratic	polynomial (other than n^2, (n^2))	exponential
1 milliseconds	10 milliseconds	1 second	10 seconds	17 minutes	277 hours	heat death of the universe

Algorithmic complexity analysis 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.

(credit: Katie Creel)

Homework

- Recursion tutorial: https://www.learncpp.com/cpp-tutorial/recursion/
- Homework 4