# Complexity
*(part 2)*

# The story so far

Big-O notation: drops constant factors in algorithm runtime

- $O(n^2)$: time proportional to square of input size (e.g. ???)
- $O(n)$: time proportional to input size (e.g. ???)
- $O(\log n)$: time proportional to log of input size, or: time proportional to $n$, for input of size $2^n$ (e.g. ???)

We also accept *answers that are too big* so something that is $O(n)$ is also $O(n^2)$

# The story so far

Big-O notation: drops constant factors in algorithm runtime

- $O(n^2)$: time proportional to square of input size (e.g. naïve dynamic arrays)
- $O(n)$: time proportional to input size (e.g. linear search, good dynamic arrays)
- $O(\log n)$: time proportional to log of input size, or: time proportional to $n$, for input of size $2^n$ (e.g. binary search)

We also accept *answers that are too big* so something that is $O(n)$ is also $O(n^2)$

# The story so far

## Hierarchy

- $O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3) < O(2^n)$
- Adding together terms gives you the *biggest* one
- e.g., $O(\log n) + O(n^2) + O(n) = O(n^2)$

## Computing big-O using hierarchy:

- $2n^2 + 3n + 2 = ???$

# The story so far

## Hierarchy

- $O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3) < O(2^n)$
- Adding together terms gives you the *biggest* one
- e.g., $O(\log n) + O(n^2) + O(n) = O(n^2)$

## Computing big-O using hierarchy:

- $2n^2 + 3n + 2 = O(n^2) + O(n) + O(1) = O(n^2)$

# Multiplying big O

$O(f(n)) \times O(g(n)) = O(f(n) \times g(n))$

- e.g., $O(n^2) \times O(\log n) = O(n^2 \log n)$

You can drop constant factors:

- $k \times O(f(n)) = O(f(n))$, if k is constant
- e.g. $2 \times O(n) = O(n)$

(Exercise: show that these are true)

# The rules

There are three rules you need for calculating big O:

- Addition (hierarchy)
- Multiplication
- Replacing a term with a *bigger* term

# Quiz

What is $(n^2 + 3)(2^n \times n) + \log_{10} n$ in Big O notation?

# Answer

$(n^2 + 3)(2^n \times n) + \log_{10} n$

$= O(n^2) \times O(2^n \times n) + O(\log n)$

$= O(2^n \times n^3) + O(\log n)$ (multiplication)

$= O(2^n \times n^3)$ (hierarchy)

# Example of replacing a term with a bigger term

Suppose we want to prove from scratch the rules for adding big-O:

- $O(n^2) + O(n^3) = O(n^3)$

We know $n^2 < n^3$

$O(n^2) + O(n^3)$

$\rightarrow O(n^3) + O(n^3)$ (since $n^2 < n^3$)

$= 2O(n^3)$

$= O(n^3)$ (throw out constant factors)

# Complexity of a program

```
boolean unique(Object[] a) {
  for(int i = 0; i < a.length; i++)
    for (int j = 0; j < a.length; j++)
      if (a[i].equals(a[j]) && i != j)
        return false;
  return true;
}
```

# Complexity of a program

```
boolean unique(Object[] a) {
  for(int i = 0; i < a.le
    for (int j = 0; j < a.len
      if (a[i].equals(a[j]) && i != j)
        e;
  ret
}
```

Outer loop runs
n times:
$O(n) \times O(n) = O(n^2)$

Inner loop runs
n times:
$O(n) \times O(1) = O(n)$

Loop body:
$O(1)$

# Complexity of loops

The complexity of a loop is:
is the number of times it runs
times the complexity of the body

Or:
If a loop runs $O(f(n))$ times
and the body takes $O(g(n))$ time
then the loop takes $O(f(n) \times g(n))$

# What about this one?

```
void function(int n) {
  for(int i = 0; i < n*n; i++)
    for (int j = 0; j < n; j++)
      “something taking O(1) time”
}
```

# What about this one?

```
void function(int n) {
  for(int i = 0; i < n*n,
    for (int j = 0; j < n; j+ )
      “s ething taking O(1) time”
}
```

Outer loop runs $n^2$ times: $\mathbf{O(n^2)} \times O(n) = O(n^3)$

Inner loop runs n times: $O(n) \times O(1) = O(n)$

Loop body: $O(1)$

# What about this one?

```
void function(int n) {
  for(int i = 0; i < n*n; i++)
    for (int j = 0; j < n/2; j++)
      "something taking O(1) time"
}
```

# What about this one?

```
void function(int n) {
  for(int i = 0; i < n*n,
    for (int j = 0; j < n/2; j
      “s   ething taking O(1) time”
}
```

Outer loop runs $n^2$ times:
$O(n^2) \times O(n) = O(n^3)$

Inner loop runs
$n/2$ = **$O(n)$** times:
$O(n) \times O(1) = O(n)$

Loop body:
$O(1)$

# Here's a new one

```
boolean unique(Object[] a) {
  for(int i = 0; i < a.length; i++)
    for (int j = 0; j < i; j++)
      if (a[i].equals(a[j]))
        return false;
  return true;
}
```

# Here's a new one

```
boolean unique(Object[] a) {
  for(int i = 0; i < a.length; i++)
    for (int j = 0; j < i; j++)
      if (a[i].equals(a[j]))
                e;
  ret
}
```

Inner loop is
$i \times O(1) = O(i)$??
But it should be
in terms of n?

Body is O(1)

# Here's a new one

```
boolean unique(Object[] a) {
  for(int i = 0; i < a.length; i++)
    for (int j = 0; j < i; j++)
      if (a[i].equals(a[j]))
          e;
  ret
}
```

i < n, so **i is O(n)**
So loop runs **O(n)**
times, complexity:
$O(n) \times O(1) = O(n)$

Body is O(1)

# Here's a new one

```
boolean unique(Object[] a) {
  for(int i = 0; i < a.le
    for (int j = 0; j < i; j+ )
      i  (a[i] equals(a[j]))
                e;
  ret
}
```

Outer loop runs n times: $O(n) \times O(n) = O(n^2)$

i < n, so **i is O(n)**
So loop runs **O(n)** times, complexity:
$O(n) \times O(1) = O(n)$

Body is O(1)

# What's the complexity?

```
void something(Object[] a) {
  for(int i = 0; i < a.length; i++)
    for (int j = 1; j < a.length; j *= 2)
      … // something taking O(1) time
}
```

# What's the complexity?

Outer loop is O(n log n)

Inner loop is O(log n)

```
void something(Object[] a) {
  for(int i = 0; i < a.length; i++)
    for (int j = 1; j < a.length; j *= 2)
      … // something taking O(1) time
}
```

A loop running through i = 1, 2, 4, …, n runs **O(log n)** times!

# While loops

```
long squareRoot(long n) {
    long i = 0;
    long j = n+1;
    while (i + 1 != j) {
        long k = (i + j) / 2;
        if (k*k <= n) i = k;
        else j = k;
    }
    return i;
}
```

# While loops

```
long squareRoot(long n) {
    long i = 0;
    long j = n+1;
    while (i + 1 != j) {
        long k = (i + j) / 2;
        if (k*k <= n) i = k;
        else j = k;
    }
    return i;
}
```

Each iteration takes O(1) time

...and halves `j-i`, so **O(log n)** iterations

# Summary: loops

## Basic rule for complexity of loops:

- Number of iterations times complexity of body
- for (int i = 0; i < n; i++) ...: n iterations
- for (int i = 1; i ≤ n; i *= 2): O(log n) iterations
- While loops: same rule, but can be trickier to count number of iterations

## If the complexity of the body depends on the value of the loop counter:

- e.g. O(i), where 0 ≤ i < n
- round it up to O(n)!

# Sequences of statements

What's the complexity here?
(Assume that the loop bodies are O(1))

```
for (int i = 0; i < n; i++) …
for (int i = 1; i < n; i *= 2) …
```

# Sequences of statements

What's the complexity here?
(Assume that the loop bodies are O(1))

```
for (int i = 0; i < n; i++) …
for (int i = 1; i < n; i *= 2) …
```

First loop: **O(n)**
Second loop: **O(log n)**
Total: O(n) + O(log n) = **O(n)**

For sequences, add the complexities!

# A familiar scene

```
int[] array = {};
for (int i = 0; i < n; i++) {
  int[] newArray =
    new int[array.length+1];
  for (int j = 0; j < i; j++)
    newArray[j] = array[j];
  newArray = array;
}
```

# A familiar scene

```
int[] array = {};
for (int i = 0; i < n; i
  int[] newArray =
    new int[array.length+1];
  for (int j = 0; j < i; j++)
    newArray[j] = array[j];
  newArray =
}
```

Rest of loop body
**O(1)**,
so loop body
O(1) + O(n) = **O(n)**

Outer loop:
n iterations,
O(n) body,
so **$O(n^2)$**

Inner loop
**O(n)**

# A familiar scene, take 2

```
int[] array = {};
for (int i = 0; i < n; i+=100) {
  int[] newArray =
    new int[array.length+100];
  for (int j = 0; j < i; j++)
    newArray[j] = array[j];
  newArray = array;
}
```

# A familiar scene, take 2

```
int[] array = {};
for (int i = 0; i < n; i+=100) {
  int[] newArray =
    new int[array.length+100];
  for (int j = 0; j < i; j++)
    newArray[j] = array[j];
  newArray =
}
```

Outer loop:
n/100 iterations,
which is O(n)
O(n) body,
so **O(n²)** still

# A familiar scene, take 3

```
int[] array = {0};
for (int i = 1; i <= n; i*=2) {
  int[] newArray =
    new int[array.length*2];
  for (int j = 0; j < i; j++)
    newArray[j] = array[j];
  newArray = array;
}
```

# A familiar scene, take 3

```
int[] array = {0};
for (int i = 1; i <= n; i*=2) {
  int[] newArray =
    new int[array.length*2];
  for (int j = 0; j < i; j++)
    newArray[j] = array[j];
  newArray =
}
```

Outer loop:
log n iterations,
O(n) body,
so **O(n log n)**??

# A familiar scene, take 3

```
int[] array = {0};
for (int i = 1; i <= n; i*=2) {
  int[] newArray =
    new int[array.length*2];
  for (int j = 0; j < i; j++)
    newArray[j] = array[j];
  newArray =
}
```

Here we "round up" O(i) to O(n). This causes an overestimate!

# A complication

Our algorithm has O(n) complexity, but we've calculated O(n log n)

- An overestimate, but not a severe one

  (If n = 1000000 then n log n = 20n)
- This can happen but is normally not severe
- To get the right answer: do the maths

Good news: for “normal” loops this doesn't happen

- If all bounds are n, or $n^2$, or another loop variable, or a loop variable squared, or …

Main exception: loop variable *i* doubles every time, body complexity depends on *i*

# Doing the sums

In our example:

- The inner loop's complexity is O(i)
- In the outer loop, i ranges over 1, 2, 4, 8, ..., $2^a$

Instead of rounding up, we will add up the time for all the iterations of the loop:

$1 + 2 + 4 + 8 + \ldots + 2^a$

$= 2^{a+1} - 1 < 2 \times 2^a$

Since $2^a \leq n$, the total time is at most 2n, which is O(n)

# A last example

```
for (int i = 1; i <= n; i *= 2) {
   for (int j = 0; j < n*n; j++)
      for (int k = 0; k <= j; k++)
         // O(1)
   for (int j = 0; j < n; j++)
      // O(1)
}
```

# A last example

The outer loop runs O(log n) times

The j-loop runs $n^2$ times

```
for (int i = 1; i <= n; i *= 2) {
    for (int j = 0; j < n*n; j++)
        for (int k = 0; k <= j; k++)
            …
    for (int j = 0; j < n; j++)
        …
}
```

`k <= j < n*n` so this loop is $O(n^2)$

This loop is $O(n)$

Total: $O(\log n) \times (O(n^2) \times O(n^2) + O(n))$
$= O(n^4 \log n)$

# A trick: sums are almost integrals

$$\sum_{x=a}^{b} f(x) \approx \int_{a}^{b} f(x)$$

For example:

$$\sum_{0}^{n} n = n(n+1)/2 \qquad \int_{0}^{x} x = x^2/2$$

Not quite the same, but close!

This trick is accurate enough to give you the right complexity class

See: “Finite calculus: a tutorial for solving nasty sums”

# Summary

Big O complexity

- Calculate runtime without doing hard sums!
- Lots of "rules of thumb" that work almost all of the time
- Very occasionally, still need to do hard sums :(
- Ignoring constant factors: seems to be a good tradeoff

Weiss chapter 5