CSJ61B Lecture 13

Tuesday, March 3, 2020

This marks the beginning of the second half of CS61B, in which we discuss how to write programs that make the most efficient possible use of our computer's time and memory. In the first half, we discussed how to make the best use of your time, the programmer, with strategies such as debugging, testing and ways to use various techniques like OOP.

We will only be discussing the basic mathematics under this in 61B, but there will be further resources for you to research if you'd like.

As an example of this part of the class, let's consider the following problem: we want to consider if the sorted array `A` has any indices `i` and `j` where `A[i] = A[j]`?

```
A = [-3, -1, 2, 4, 4, 8, 10, 12]
```

- One possible solution would be to consider every possible pair. This is not very efficient,
- because we can take advantage of its sorted nature to only consider `A[i]` and `A[i+1]`, returning `true` if we see a match.

We know for the 2nd approach is better, but our goal today is to derive some formal technique to get a sense of how to quantify this difference.

Runtime Characterization

Our goal is to somehow characterize the runtimes of these two functions, which implement the solutions described above:

```
public static boolean dup1(int[] A) {
  for (int i = 0; i < A.length; i += 1) {
    for (int j = i + 1; j < A.length; j += 1) {
      if (A[i] == A[j]) {
         return true;
      }
    }
  }
  return false;
}
```

```
public static boolean dup2(int[] A) {
  for (int i = 0; i < A.length - 1; i += 1) {
    if (A[i] == A[i + 1]) {
      return true;
    }
  }
  return false;
}
```

We want this characterization to be at once simple, but also mathematically rigorous. One possible simple characterization would be that `dup1` has two for loops, and `dup2` only has one, but this isn't terribly mathematically rigorous. In addition, we also want the characterization to demonstrate that one algorithm is better than another.

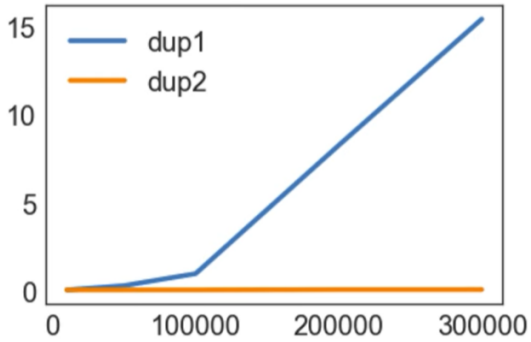
Techniques to Measure

Measure execution time

The simplest possible characterization is to simply run both methods on some array and measure the execution time, using either a stopwatch, the built-in Unix `time` command, or the Princeton Standard Library's `Stopwatch` class.

N	dup1	dup2
10000	0.08	0.08
50000	0.32	0.08
100000	1.00	0.08
200000	8.26	0.1
400000	15.4	0.1

Time to complete (in seconds)



This seems like a good approach: it's easy to measure, and its meaning is obvious. The problem is that it may require large amounts of computation time, and the execution time may vary with the machine, compiler, or input data.

Counting operations

So let's try another approach: can we, looking the code, build a model that counts the operations for an array of size 10,000:

```
for (int i = 0; i < A.length; i += 1) {
  for (int j = i+1; j < A.length; j += 1) {
    if (A[i] == A[j]) {
       return true;
    }
  }
}
return false;
```

The counts are tricky to compute. Work not shown.

operation	count, N=10000
i = 0	1
j = i + 1	1 to 10000
less than (<)	2 to 50,015,001
increment (+=1)	0 to 50,005,000
equals (==)	1 to 49,995,000
array accesses	2 to 99,990,000

The good thing with this approach is that this is machine independent, and the input dependence is captured by the model. The cons are that it is tedious to compute, the array size is arbitrary and it doesn't tell you the actual time the algorithm takes.

Counting operations in terms of array size `N`

```
for (int i = 0; i < A.length; i += 1) {
  for (int j = i+1; j<A.length; j += 1) {
    if (A[i] == A[j]) {
       return true;
    }
  }
}
return false;
```

operation	symbolic count	count, N=10000
i = 0	1	1
j = i + 1	1 to N	1 to 10000
less than (<)	2 to $(N^2+3N+2)/2$	2 to 50,015,001
increment (+=1)	0 to $(N^2+N)/2$	0 to 50,005,000
equals (==)	1 to $(N^2-N)/2$	1 to 49,995,000
array accesses	2 to N^2-N	2 to 99,990,000

Instead of using some arbitrary number of 10000, we can express our model in terms of `N`, the array size. This removes our problem where the array size is arbitrary, telling us how our algorithm scales, but the problem is that it's even more tedious to compute.

> **Practice**
>
> Perform this operation counting problem on the algorithm `dup2`.
>
> ```
> for (int i = 0; i < A.length - 1; i += 1){
>   if (A[i] == A[i + 1]) {
>     return true;
>   }
> }
> return false;
> ```
>
> | operation | sym. count | count, N=10000 |
> |---|---|---|
> | i = 0 | 1 | 1 |
> | less than (<) | | |
> | increment (+=1) | | |
> | equals (==) | | |
> | array accesses | | |
>
> **Solution**
>
> ```
> for (int i = 0; i < A.length - 1; i += 1) {
>   if (A[i] == A[i + 1]) {
>     return true;
>   }
> }
> return false;
> ```
>
> Especially observant folks may notice we didn't count everything, e.g. "- 1" and "+ 1" operations. We'll see why this omission is not a problem very shortly.
>
> | operation | symbolic count | count, N=10000 |
> |---|---|---|
> | i = 0 | 1 | 1 |
> | less than (<) | 0 to N | 0 to 10000 |
> | increment (+=1) | 0 to N - 1 | 0 to 9999 |
> | equals (==) | 1 to N - 1 | 1 to 9999 |
> | array accesses | 2 to 2N - 2 | 2 to 19998 |
>
> If you did this exercise but were off by one, that's fine. The exact numbers aren't that important.

Why Scaling Matters

So, using the two charts we made earlier, can you describe which algorithm is better, and why?

operation	symbolic count	count, N=10000
i = 0	1	1
j = i + 1	1 to N	1 to 10000
less than (<)	2 to $(N^2+3N+2)/2$	2 to 50,015,001
increment (+=1)	0 to $(N^2+N)/2$	0 to 50,005,000
equals (==)	1 to $(N^2-N)/2$	1 to 49,995,000
array accesses	2 to N^2-N	2 to 99,990,000

dup1

operation	symbolic count	count, N=10000
i = 0	1	1
less than (<)	0 to N	0 to 10000
increment (+=1)	0 to N - 1	0 to 9999
equals (==)	1 to N - 1	1 to 9999
array accesses	2 to 2N - 2	2 to 19998

dup2

`dup2` is better. Here are the reasons:

- It takes fewer operations to do the same work, e.g. 50,015,001 vs. 10,000 operations
- A better answer is that the algorithm scales better in the worst case:

$$\frac{N^2 + 3N + 2}{2} \text{ vs. } N$$

- And an even better answer (maybe vaguer) is that the parabolas (N squared) grow faster than lines (N).

And here's really the key idea: using the geometric interpretation to build our intuition for which algorithm is better.

Asymptotic Behavior

In most cases, we care only about asymptotic behavior: what happens for very large `N`. We will get the largest benefits from having "better" data structures when we are talking about very large data sets:

- Simulating billions of interacting particles
- Social network with billions of users
- Logging billions of transactions
- Encoding of billions of bytes of video data

Algorithms that scale well (look like lines) have better asymptotic runtime behavior than algorithms that scale relatively poorly.

Parabolas vs. Lines

Suppose we are trying to zerp-ify a collection of `N` items (whatever "zerp"-ing is), and we have two algorithms.

- `zerp1` takes

$$2N^2$$

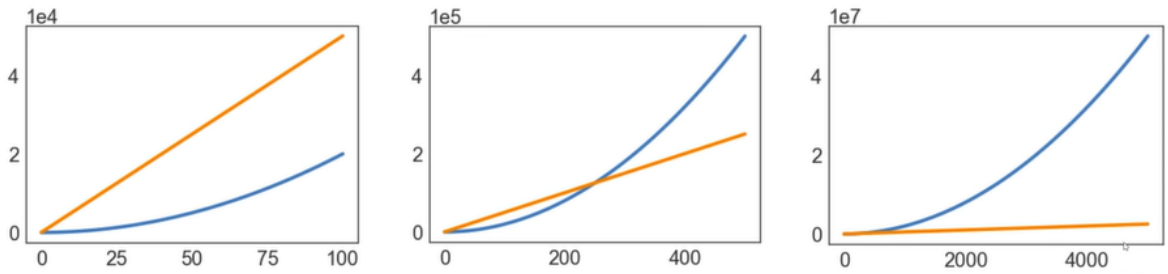
operations

- `zerp2` takes

$$500N$$

operations

For small `N`, `zerp1` might be faster, but as our data size grows, the parabolic algorithm is going to fall further and further behind.



We are going to informally refer to the "shape" on a runtime function as its "order of growth". For very small `N`, the difference between the algorithms is not going to make any difference. But the effect as `N` grows can be insanely big -- it often determines whether a problem can be solved at all.

	n	$n \log_2 n$	n^2	n^3	1.5^n	2^n	$n!$
$n = 10$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	4 sec
$n = 30$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	18 min	10^{25} years
$n = 50$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	11 min	36 years	very long
$n = 100$	< 1 sec	< 1 sec	< 1 sec	1 sec	12,892 years	10^{17} years	very long
$n = 1{,}000$	< 1 sec	< 1 sec	1 sec	18 min	very long	very long	very long
$n = 10{,}000$	< 1 sec	< 1 sec	2 min	12 days	very long	very long	very long
$n = 100{,}000$	< 1 sec	2 sec	3 hours	32 years	very long	very long	very long
$n = 1{,}000{,}000$	1 sec	20 sec	12 days	31,710 years	very long	very long	very long

Duplicate Finding

Our goal was to characterize the runtimes of the functions `dup1` and `dup2`. The model building technique we used did satisfy our condition that it should demonstrate the superiority of one algorithm over another, and it is pretty mathematically rigorous. but it was not simple.

Worst Case Orders of Growth

Let's be more careful about what we mean when we say one function is "like" a parabola, and another is "like" a parabola. To do that, we are actually going to make several simplifications.

Simplification 1: Consider only the worst case

The reason we do this is that we often only care about the worst case (there are counter-examples, however), because often, the most interesting "effects" or differences between algorithms.

operation	worst case count
i = 0	1
j = i + 1	N
less than (<)	$(N^2+3N+2)/2$
increment (+=1)	$(N^2+N)/2$
equals (==)	$(N^2-N)/2$
array accesses	N^2-N

In our `dup` example, we are only focusing on the case where there are no duplicates, because this is where there is a major performance difference.

Before we move on, do the following exercise:

Consider the algorithm below. What do you expect will be the **order of growth** of the runtime for the algorithm?

A. N [linear]
B. N^2 [quadratic]
C. N^3 [cubic]
D. N^6 [sextic]

operation	count
less than (<)	$100N^2 + 3N$
greater than (>)	$2N^3 + 1$
and (&&)	5,000

In other words, if we plotted total runtime vs. N, what shape would we expect?

Solution

The runtime is C -- cubic. The reasons are:

- Suppose < takes X nanoseconds, > takes Y nanoseconds, and && takes Z nanoseconds.
- The total time is:

  $$X(100N^2 + 3N) + Y(2N^3 + 1) + 5000Z$$

  nanoseconds.
- **For very large N, the**

  $$2YN^3$$

  term is much larger than the others.

Simplification 2: Restrict Attention to One Operation

We should pick some representative operation to act as a proxy for the overall runtime. In our previous example, this would be increment, or less than. A bad choice would be the assignment operations. We call this choice the "cost model":

operation	worst case count
i = 0	1
j = i + 1	N
less than (<)	$(N^2+3N+2)/2$
increment (+=1)	$(N^2+N)/2$
equals (==)	$(N^2-N)/2$
array accesses	N^2-N

Simplification 3: Eliminate low order terms

The N in

$$\frac{N^2 + N}{2}$$

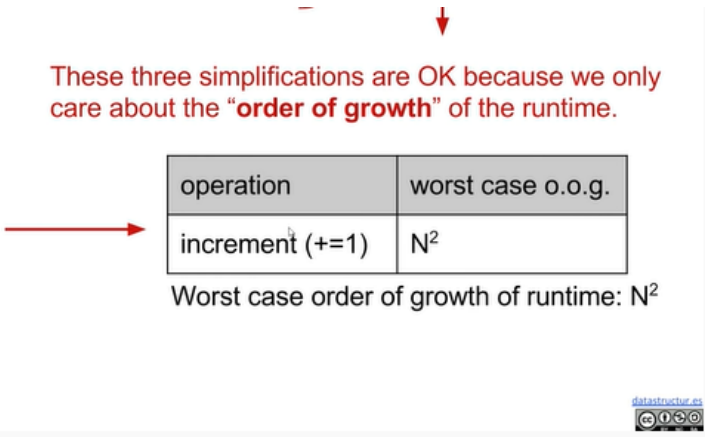
doesn't matter, because it is the N squared that grows exponentially.

Simplification 4: Eliminate multiplicative constants

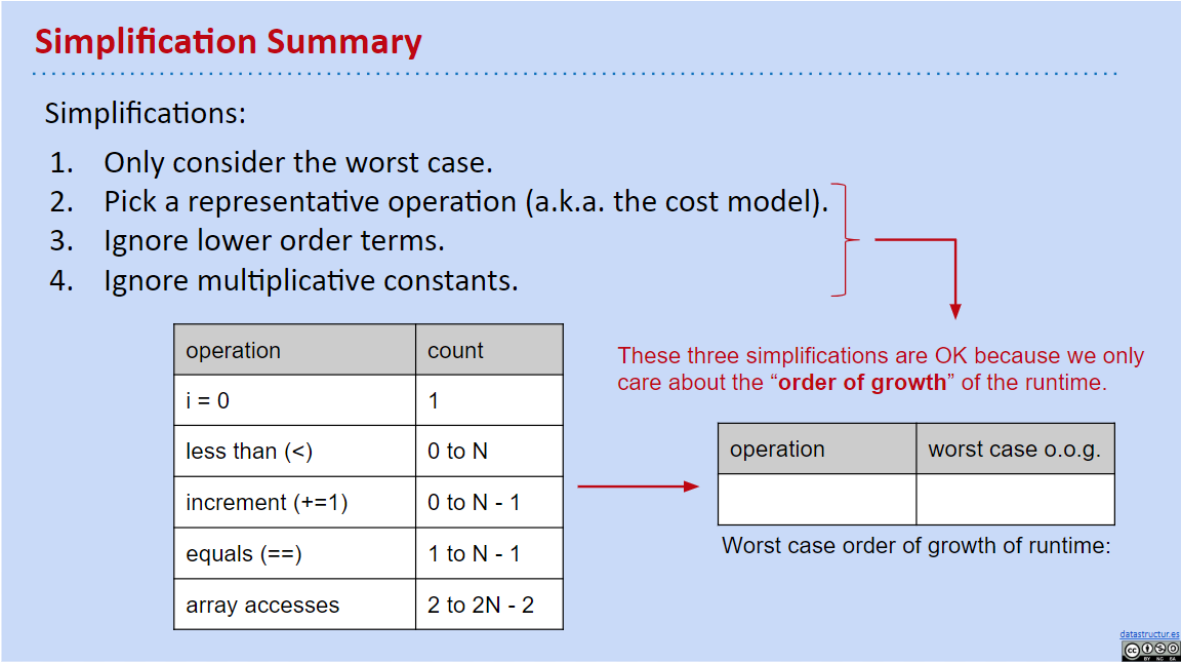
We should also ignore the multiplicative constants, because they have no real meaning. You might argue with this, saying that an operation that takes N is much faster than one that takes 1000N, but then again it's not like we haven't thrown away useful information in the prior simplifications. Unless you have a true reason to believe that this is needed, we can eliminate it for simplicity.

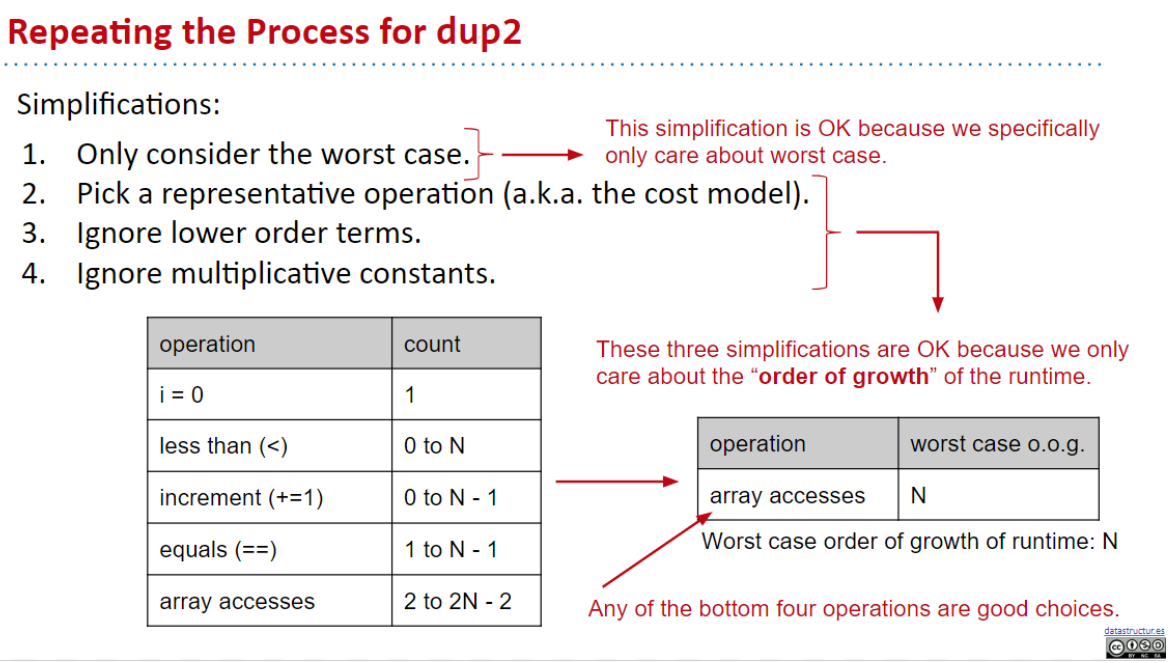
In these four simplifications, we have simplified our massive table into a much smaller one:

operation	count
i = 0	1
j = i + 1	1 to N
less than (<)	2 to $(N^2+3N+2)/2$
increment (+=1)	0 to $(N^2+N)/2$
equals (==)	1 to $(N^2-N)/2$
array accesses	2 to N^2-N



Practice





Summary

Our process is to:

- Construct a table of exact counts of all possible operations.
- Convert the table into a worst case order of growth using 4 simplifications.

It's still tedious, so our goal now is to begin using our simplifications from the outset, to avoid building this table at all.

Simplified Analysis

We are going to use our four simplifications to make a proposal of how to simplify our analysis and characterization of algorithms. Here's a general guideline of how we will do so:

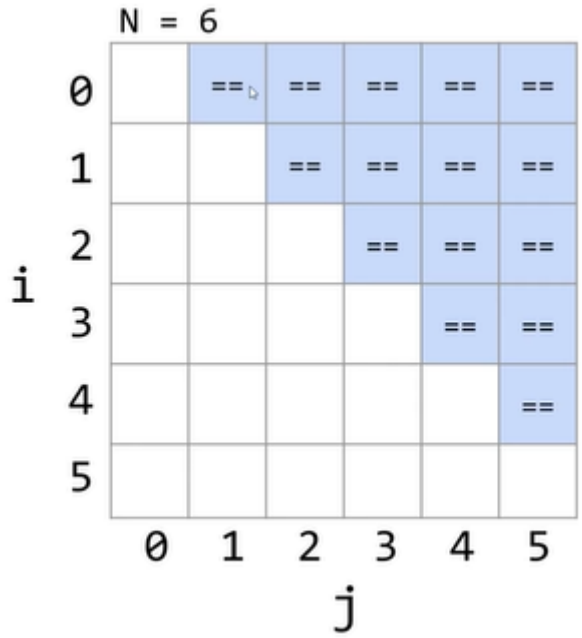
- Choose a representative operation to count (the cost model)
- Figure out the order of growth for the count of the representative operation by either:
    - Making an exact count, then discarding the unnecessary pieces
    - Using intuition and inspection to determine order of growth (only possible with lots of practice)

Let's do this with `dup1` again:

Analysis of Nested `for` Loops

Finding the order of growth of the worst case runtime of `dup1` requires that we first select the representative operation. In our analysis, we will select the equality operation, because we know that it lends itself to the easiest operation. You might select the wrong one at first, but eventually you will be able to have an intuition for what is most important.

How do we count it? The worst case number of `==` operations can be quantified with the following table:



It shows for which pairs of `i` and `j` values the values get compared. We can see that for some value `N`, we perform:

$$1 + 2 + 3 + \ldots + (N-2) + (N-1)$$

operations.

How do we solve this as some function of `N`? You've probably seen this trick before in maths:

$$C = N - 1 \times \frac{N}{2}$$

We will see that we end up with an

$$N^2$$

number of `==` operations once we have thrown away the lower order terms and multiplicative constants, so we know the worst case order of growth runtime was N-squared.

There's an alternate way, where we don't come up with the exact count at all. This is more of an art, and less of a science. We will use the geometry of the table above to make the argument that the worst case order of growth is

$$N^2$$

Formalizing Order of Growth

Given a function Q(N), we can apply our last two simplifications to yield the order of growth of Q(N). For example:

$$Q(N) = 3N^3 + N^2$$

We can describe its order of growth as:

$$N^3$$

Let's formalize this notation using something called Big Theta.

Big Theta

The math might seem a bit daunting at first, but the idea underneath is the same. Using "Big Theta" instead of "order of growth" does not change anything much from how we have been analyzing code. It's just the definition of it that rigorously, mathematically, captures the idea of what we've been trying to do.

Before we move to its formal definition, let's discuss how it is used. Suppose we have a function R(N) with order of growth f(N). In Big Theta notation, we write this as

$$R(N) \in \Theta(f(N))$$

We say it is a member of the family of functions whose growth is characterized by f(N).

Formally, it means there exist positive constants k1 and k2, such that:

$$k_1 \bullet f(N) \leq R(N) \leq k_2 \bullet f(N)$$

for all values of N greater than some very large N.

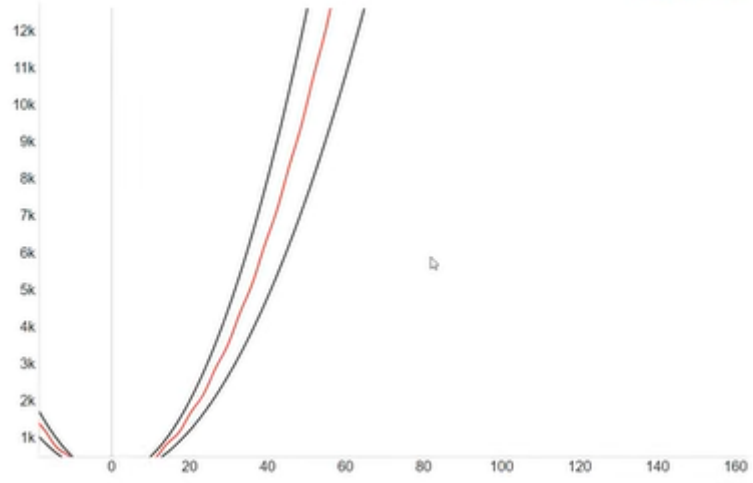
As an example, let's consider:

$$R(N) = 40sin(N) + 4N^2$$

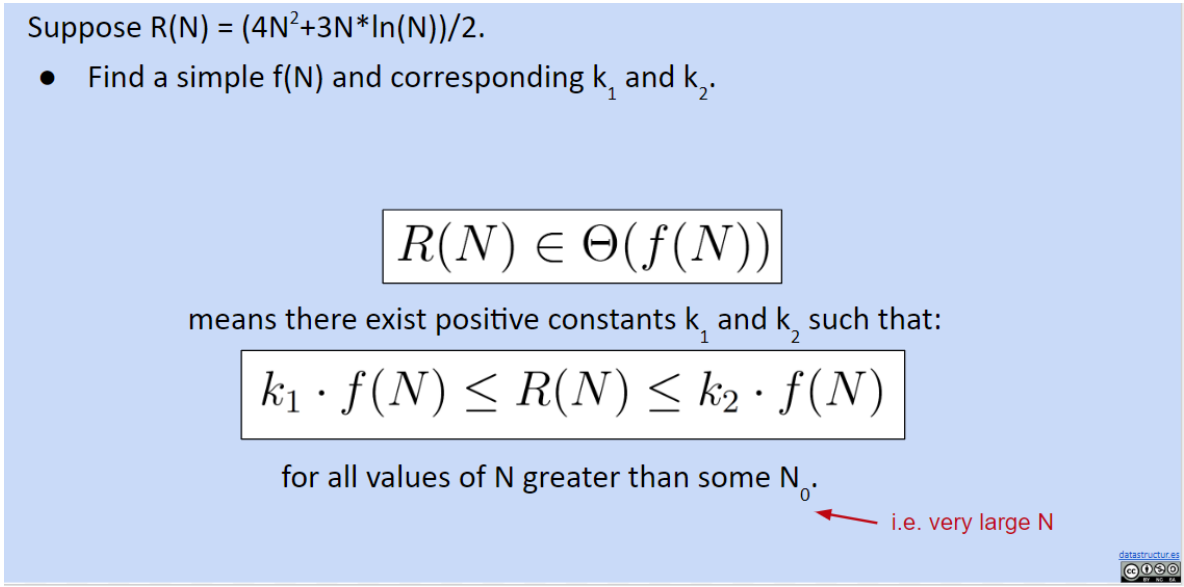
which belongs to the family

$$N^2$$

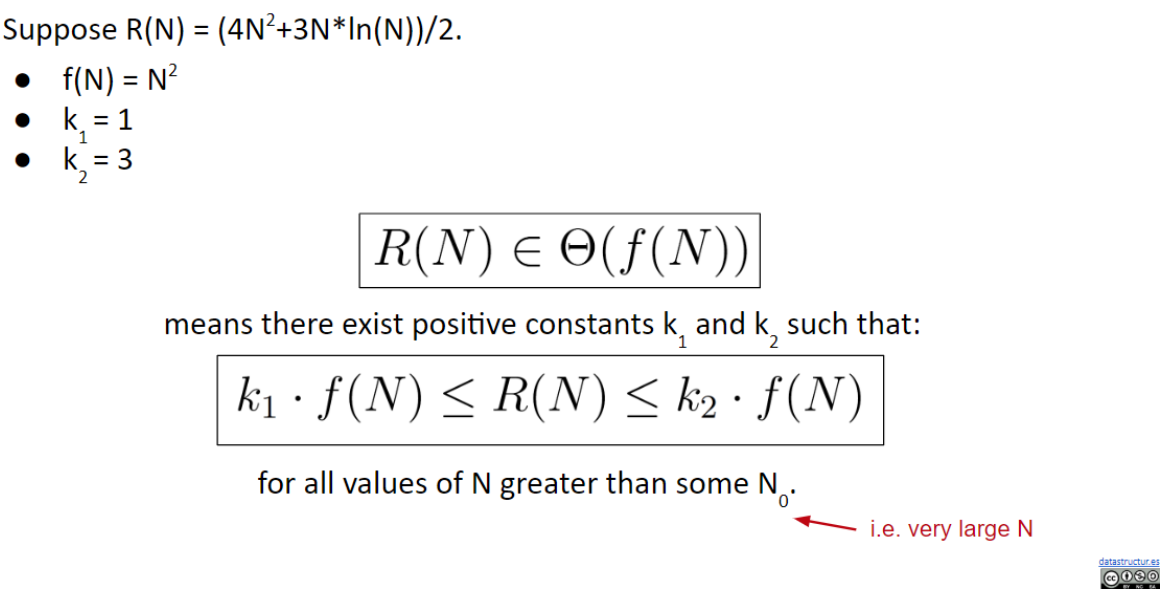
We can select k1 as 3, and k2 as 5. What's special about these choices is that for really large N, only the 4N^2 is really gonna matter, and our function is going to be between 3N^2 and 5N^2. Here's a visualization:



Practice



Solution



These values of k1 and k2 do not have to be integers, as 1.5 and 2.5 work well too, but for simplicity's sake we have selected integers.

Big O

So the last thing we want to talk about today is Big O, which is closely related to Big Theta. We used Big Theta to describe the order of growth of a function, or the rate of growth of the runtime of a piece of code.

Whereas Big Theta can be informally thought of something as like "equals", Big O can be thought of as "less than or equal to". For example, the following are all true:

- $N^3 + 3N^4 \in \Theta(N^4)$
- $N^3 + 3N^4 \in O(N^4)$
- $N^3 + 3N^4 \in O(N^6)$
- $N^3 + 3N^4 \in O(N!)$
- $N^3 + 3N^4 \in O(N^{N!})$

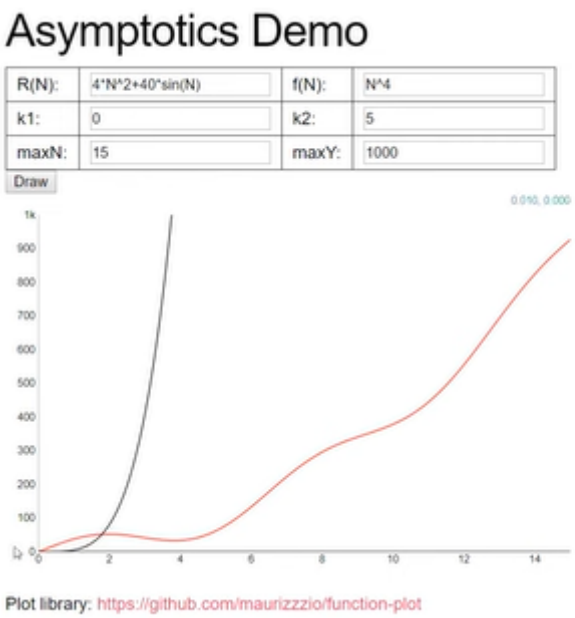
Formally,

$$R(N) \in O(f(N))$$

means there exists positive constant k2 such that:

$$R(N) \leq k_2 \bullet f(N)$$

for all values of N greater than some very large N.



	Informal meaning:	Family	Family Members
Big Theta $\Theta(f(N))$	Order of growth is f(N).	$\Theta(N^2)$	$N^2/2$ $2N^2$ $N^2 + 38N + N$
Big O $O(f(N))$	Order of growth is less than or equal to f(N).	$O(N^2)$	$N^2/2$ $2N^2$ $\lg(N)$

We will see why big O is practically useful in the next lecture.

Summary

- Given a code snippet, we can express its runtime as a function R(N), where N is some property of the input of the function (often the size of the input).
- Rather than finding R(N) exactly, we instead usually only care about the order of growth of R(N).
- One approach (not universal):
  - Choose a representative operation, and let C(N) be the count of how many times that operation occurs as a function of N.
  - Determine order of growth f(N) for C(N), i.e.

$$C(N) \in \Theta(f(N))$$

    - Often (but not always) we consider the worst case count.

- If operation takes constant time, then

$$R(N) \in \Theta(f(N))$$

- Can use O as an alternative for Theta. O is used for upper bounds.