Introduction to asymptotic analysis

Or: How computer science is like cooking

[AUT CSC]

AUT Computer Science Club

Koz Ross

March 30, 2017

Overview

- **What are algorithms, and why do we care?**
- **Building up**
- **Going asymptotic**
- **Questions**

What are algorithms, and why do we care?

Definition of an algorithm

“A procedure, described in a finite number of instructions, for transforming inputs into an output to solve a specific problem. The instructions must be unambiguous, guaranteed to terminate, and solve the problem correctly in all cases.”

Definition of an algorithm

> *"A procedure, described in a finite number of instructions, for transforming inputs into an output to solve a specific problem. The instructions must be unambiguous, guaranteed to terminate, and solve the problem correctly in all cases."*

Alternatively, you can think of an algorithm as a *recipe*:

Definition of an algorithm

> *"A procedure, described in a finite number of instructions, for transforming inputs into an output to solve a specific problem. The instructions must be unambiguous, guaranteed to terminate, and solve the problem correctly in all cases."*

Alternatively, you can think of an algorithm as a *recipe*:

- The ingredients are the *inputs*

Definition of an algorithm

> *"A procedure, described in a finite number of instructions, for transforming inputs into an output to solve a specific problem. The instructions must be unambiguous, guaranteed to terminate, and solve the problem correctly in all cases."*

Alternatively, you can think of an algorithm as a *recipe*:

- The ingredients are the *inputs*
- The recipe directions are the *instructions*

Definition of an algorithm

"A procedure, described in a finite number of instructions, for transforming inputs into an output to solve a specific problem. The instructions must be unambiguous, guaranteed to terminate, and solve the problem correctly in all cases."

Alternatively, you can think of an algorithm as a *recipe*:

- The ingredients are the *inputs*
- The recipe directions are the *instructions*
- The (hopefully!) resulting food is the *output*

Why we care

- Algorithms are *very* fundamental to computer science (which is why it’s all about cooking!)

Why we care

- Algorithms are *very* fundamental to computer science (which is why it's all about cooking!)
- They define what we can do *at all*, and what we can do *efficiently*

Why we care

- Algorithms are *very* fundamental to computer science (which is why it’s all about cooking!)
- They define what we can do *at all*, and what we can do *efficiently*
- Improved algorithms lead to improved everything-else

Why we care

- Algorithms are *very* fundamental to computer science (which is why it's all about cooking!)
- They define what we can do *at all*, and what we can do *efficiently*
- Improved algorithms lead to improved everything-else
- 'Ready-baked' efficient solutions to many problems (theoretical *and* practical)

Why we care

- Algorithms are *very* fundamental to computer science (which is why it’s all about cooking!)
- They define what we can do *at all*, and what we can do *efficiently*
- Improved algorithms lead to improved everything-else
- ‘Ready-baked’ efficient solutions to many problems (theoretical *and* practical)
- If you don’t know this stuff, everything you do on a computer will run like hot garbage, and you won’t know *why*, or what to do about it

Why we care

- Algorithms are *very* fundamental to computer science (which is why it's all about cooking!)
- They define what we can do *at all*, and what we can do *efficiently*
- Improved algorithms lead to improved everything-else
- 'Ready-baked' efficient solutions to many problems (theoretical *and* practical)
- If you don't know this stuff, everything you do on a computer will run like hot garbage, and you won't know *why*, or what to do about it

Thus, we really need to be able to *compare* algorithms to each other, and also *understand* why they run the way they do, if we have any hope of making them do our bidding.

What do we mean by 'efficient'?

Efficiency is ultimately doing more with less effort. We can measure the 'effort' required to run an algorithm in two ways:

What do we mean by ‘efficient’?

Efficiency is ultimately doing more with less effort. We can measure the ‘effort’ required to run an algorithm in two ways:

- How much *time* we need to solve a problem instance; and

What do we mean by 'efficient'?

Efficiency is ultimately doing more with less effort. We can measure the 'effort' required to run an algorithm in two ways:

- How much *time* we need to solve a problem instance; and
- How much *space* (in terms of memory) we would have to use as part of this process.

What do we mean by ‘efficient’?

Efficiency is ultimately doing more with less effort. We can measure the ‘effort’ required to run an algorithm in two ways:

- How much *time* we need to solve a problem instance; and
- How much *space* (in terms of memory) we would have to use as part of this process.

Determining these two factors is an important part of *algorithm analysis*.

What do we mean by 'efficient'?

Efficiency is ultimately doing more with less effort. We can measure the 'effort' required to run an algorithm in two ways:

- How much *time* we need to solve a problem instance; and
- How much *space* (in terms of memory) we would have to use as part of this process.

Determining these two factors is an important part of *algorithm analysis*. Additionally, any method we use must explain *why* we get the performance it claims.

Building up

Analysis by implementation

"To analyze the efficiency of an algorithm, implement it, run it on some inputs, and measure the time required and the memory used."

Analysis by implementation

> *"To analyze the efficiency of an algorithm, implement it, run it on some inputs, and measure the time required and the memory used."*

Not bad *per se*, but has a few notable downsides:

Analysis by implementation

> *"To analyze the efficiency of an algorithm, implement it, run it on some inputs, and measure the time required and the memory used."*

Not bad *per se*, but has a few notable downsides:

- Very labour-intensive (programming is hard!)

Analysis by implementation

> *“To analyze the efficiency of an algorithm, implement it, run it on some inputs, and measure the time required and the memory used.”*

Not bad *per se*, but has a few notable downsides:

- Very labour-intensive (programming is hard!)
- Too many external factors (programmer skill, language, OS, hardware, workload. . .)

Analysis by implementation

> *"To analyze the efficiency of an algorithm, implement it, run it on some inputs, and measure the time required and the memory used."*

Not bad *per se*, but has a few notable downsides:

- Very labour-intensive (programming is hard!)
- Too many external factors (programmer skill, language, OS, hardware, workload. . .)
- The test data problem (what do we test against?)

Analysis by implementation

> *"To analyze the efficiency of an algorithm, implement it, run it on some inputs, and measure the time required and the memory used."*

Not bad *per se*, but has a few notable downsides:

- Very labour-intensive (programming is hard!)
- Too many external factors (programmer skill, language, OS, hardware, workload. . .)
- The test data problem (what do we test against?)

While this method is useful at the *end* of the process, it's not a very good analytical tool in general.

Analysis by implementation

> *"To analyze the efficiency of an algorithm, implement it, run it on some inputs, and measure the time required and the memory used."*

Not bad *per se*, but has a few notable downsides:

- Very labour-intensive (programming is hard!)
- Too many external factors (programmer skill, language, OS, hardware, workload. . .)
- The test data problem (what do we test against?)

While this method is useful at the *end* of the process, it's not a very good analytical tool in general. To do better than this, we need a different view of the problem, as well as a different method.

A model

In sciences (of all sorts), a *model* is a way of looking at the world which:

A model

In sciences (of all sorts), a *model* is a way of looking at the world which:

- *Minimizes* the impact of factors we *don't* care about

A model

In sciences (of all sorts), a *model* is a way of looking at the world which:

- *Minimizes* the impact of factors we *don't* care about
- Stays *true to the reality* of factors we *do* care about

A model

In sciences (of all sorts), a *model* is a way of looking at the world which:

- *Minimizes* the impact of factors we *don't* care about
- Stays *true to the reality* of factors we *do* care about
- *Explains* why phenomena occur like they do.

A model

In sciences (of all sorts), a *model* is a way of looking at the world which:

- *Minimizes* the impact of factors we *don't* care about
- Stays *true to the reality* of factors we *do* care about
- *Explains* why phenomena occur like they do.

Examples of models include:

A model

In sciences (of all sorts), a *model* is a way of looking at the world which:

- *Minimizes* the impact of factors we *don't* care about
- Stays *true to the reality* of factors we *do* care about
- *Explains* why phenomena occur like they do.

Examples of models include:

- The Standard Model

A model

In sciences (of all sorts), a *model* is a way of looking at the world which:

- *Minimizes* the impact of factors we *don't* care about
- Stays *true to the reality* of factors we *do* care about
- *Explains* why phenomena occur like they do.

Examples of models include:

- The Standard Model
- The orbital model

A model

In sciences (of all sorts), a *model* is a way of looking at the world which:

- *Minimizes* the impact of factors we *don't* care about
- Stays *true to the reality* of factors we *do* care about
- *Explains* why phenomena occur like they do.

Examples of models include:

- The Standard Model
- The orbital model
- Statistical models

A model

In sciences (of all sorts), a *model* is a way of looking at the world which:

- *Minimizes* the impact of factors we *don't* care about
- Stays *true to the reality* of factors we *do* care about
- *Explains* why phenomena occur like they do.

Examples of models include:

- The Standard Model
- The orbital model
- Statistical models

Models allow us to *analyze* and *understand* their subject matter without having to 'manually verify' everything.

A model *of a computer*

Because algorithms ultimately run on computers, our model must be of a computer.

A model *of a computer*

Because algorithms ultimately run on computers, our model must be of a computer.

There are *many* choices of model, depending on what kind of computer we want to study, as well as what aspects of it interest us.

A model *of a computer*

Because algorithms ultimately run on computers, our model must be of a computer.

There are *many* choices of model, depending on what kind of computer we want to study, as well as what aspects of it interest us.

The most traditional (and foundational) model for algorithm analysis is the *random access model* (RAM).

A model *of a computer*

Because algorithms ultimately run on computers, our model must be of a computer.

There are *many* choices of model, depending on what kind of computer we want to study, as well as what aspects of it interest us.

The most traditional (and foundational) model for algorithm analysis is the *random access model* (RAM). This closely represents a computer at the time when algorithm analysis first became a topic in its own right (around 1950).

The random access model

In RAM, we have access to:

The random access model

In RAM, we have access to:

- A single, sequential *processing unit* (PU)

The random access model

In RAM, we have access to:

- A single, sequential *processing unit* (PU)
- *Addressable memory*, divided into words of w bits

The random access model

In RAM, we have access to:

- A single, sequential *processing unit* (PU)
- *Addressable memory*, divided into words of w bits

A *primitive instruction* is any of the following:

The random access model

In RAM, we have access to:

- A single, sequential *processing unit* (PU)
- *Addressable memory*, divided into words of w bits

A *primitive instruction* is any of the following:

- Arithmetic $(+, -, \cdot, \div)$ on w-bit integers

The random access model

In RAM, we have access to:

- A single, sequential *processing unit* (PU)
- *Addressable memory*, divided into words of w bits

A *primitive instruction* is any of the following:

- Arithmetic ($+, -, \cdot, \div$) on w-bit integers
- Comparison ($<, >, \leq, \geq, =, \neq$) of two w-bit integers or flags

The random access model

In RAM, we have access to:

- A single, sequential *processing unit* (PU)
- *Addressable memory*, divided into words of w bits

A *primitive instruction* is any of the following:

- Arithmetic $(+, -, \cdot, \div)$ on w-bit integers
- Comparison $(<, >, \leq, \geq, =, \neq)$ of two w-bit integers or flags
- Allocation of any number of consecutively-addressed words

The random access model

In RAM, we have access to:

- A single, sequential *processing unit* (PU)
- *Addressable memory*, divided into words of w bits

A *primitive instruction* is any of the following:

- Arithmetic $(+, -, \cdot, \div)$ on w-bit integers
- Comparison $(<, >, \leq, \geq, =, \neq)$ of two w-bit integers or flags
- Allocation of any number of consecutively-addressed words
- Given an address, a one-word read or write there

The random access model

In RAM, we have access to:

- A single, sequential *processing unit* (PU)
- *Addressable memory*, divided into words of w bits

A *primitive instruction* is any of the following:

- Arithmetic $(+, -, \cdot, \div)$ on w-bit integers
- Comparison $(<, >, \leq, \geq, =, \neq)$ of two w-bit integers or flags
- Allocation of any number of consecutively-addressed words
- Given an address, a one-word read or write there
- Conditional or unconditional branch

The random access model

In RAM, we have access to:

- A single, sequential *processing unit* (PU)
- *Addressable memory*, divided into words of w bits

A *primitive instruction* is any of the following:

- Arithmetic $(+, -, \cdot, \div)$ on w-bit integers
- Comparison $(<, >, \leq, \geq, =, \neq)$ of two w-bit integers or flags
- Allocation of any number of consecutively-addressed words
- Given an address, a one-word read or write there
- Conditional or unconditional branch
- A procedure call

The random access model

In RAM, we have access to:

- A single, sequential *processing unit* (PU)
- *Addressable memory*, divided into words of w bits

A *primitive instruction* is any of the following:

- Arithmetic $(+, -, \cdot, \div)$ on w-bit integers
- Comparison $(<, >, \leq, \geq, =, \neq)$ of two w-bit integers or flags
- Allocation of any number of consecutively-addressed words
- Given an address, a one-word read or write there
- Conditional or unconditional branch
- A procedure call
- A return of a value

The random access model

In RAM, we have access to:

- A single, sequential *processing unit* (PU)
- *Addressable memory*, divided into words of w bits

A *primitive instruction* is any of the following:

- Arithmetic ($+, -, \cdot, \div$) on w-bit integers
- Comparison ($<, >, \leq, \geq, =, \neq$) of two w-bit integers or flags
- Allocation of any number of consecutively-addressed words
- Given an address, a one-word read or write there
- Conditional or unconditional branch
- A procedure call
- A return of a value

Any primitive instruction requires one *time unit* to execute.

Time and space complexity

Let A be an algorithm, and n be the size of the largest input to A.

Time and space complexity

Let A be an algorithm, and n be the size of the largest input to A.

We define the *time complexity of A for inputs of size n* (written $T_A(n)$) as the number of primitive instructions that the PU must execute to produce correct output for A from any inputs of largest size n.

Time and space complexity

Let A be an algorithm, and n be the size of the largest input to A.

We define the *time complexity of* A *for inputs of size* n (written $T_A(n)$) as the number of primitive instructions that the PU must execute to produce correct output for A from any inputs of largest size n.

We define the *space complexity of* A *for inputs of size* n (written $S_A(n)$) as the number of words of memory that we must allocate to produce correct output for A from any inputs of largest size n.

Time and space complexity

Let A be an algorithm, and n be the size of the largest input to A.

We define the *time complexity of A for inputs of size n* (written $T_A(n)$) as the number of primitive instructions that the PU must execute to produce correct output for A from any inputs of largest size n.

We define the *space complexity of A for inputs of size n* (written $S_A(n)$) as the number of words of memory that we must allocate to produce correct output for A from any inputs of largest size n.

If there are multiple possible values for $T_A(n), S_A(n)$ for a particular n, we take the *highest* possible value.

Time and space complexity

Let A be an algorithm, and n be the size of the largest input to A.

We define the *time complexity of A for inputs of size n* (written $T_A(n)$) as the number of primitive instructions that the PU must execute to produce correct output for A from any inputs of largest size n.

We define the *space complexity of A for inputs of size n* (written $S_A(n)$) as the number of words of memory that we must allocate to produce correct output for A from any inputs of largest size n.

If there are multiple possible values for $T_A(n), S_A(n)$ for a particular n, we take the *highest* possible value. Thus, our analysis is *worst-case*.

Time and space complexity

Let A be an algorithm, and n be the size of the largest input to A.

We define the *time complexity of A for inputs of size n* (written $T_A(n)$) as the number of primitive instructions that the PU must execute to produce correct output for A from any inputs of largest size n.

We define the *space complexity of A for inputs of size n* (written $S_A(n)$) as the number of words of memory that we must allocate to produce correct output for A from any inputs of largest size n.

If there are multiple possible values for $T_A(n), S_A(n)$ for a particular n, we take the *highest* possible value. Thus, our analysis is *worst-case*.

An example algorithm

Problem: *Given a non-empty array* `arr` *of one-word integers, find the largest integer in* `arr`.

An example algorithm

Problem: *Given a non-empty array* arr *of one-word integers, find the largest integer in* arr.

function Max(arr)
    $\mathsf{x} \leftarrow \mathsf{arr}[1]$
    for $\mathsf{i} \in 2, 3, \ldots, \mathsf{length}(\mathsf{arr})$ **do**
        if $\mathsf{x} < \mathsf{arr}[\mathsf{i}]$ **then**
            $\mathsf{x} \leftarrow \mathsf{arr}[\mathsf{i}]$
        end if
    end for
    return x
end function

An example algorithm

Problem: *Given a non-empty array* arr *of one-word integers, find the largest integer in* arr.

function Max(arr)
  x $\leftarrow$ arr$[1]$
  for i $\in 2, 3, \ldots,$ length(arr) **do**
    if x $<$ arr$[$i$]$ **then**
      x $\leftarrow$ arr$[$i$]$
    end if
  end for
  return x
end function

$$T_{\mathsf{Max}}(n) =$$

An example algorithm

Problem: *Given a non-empty array* arr *of one-word integers, find the largest integer in* arr.

function Max(arr)
    x $\leftarrow$ arr$[1]$
    for i $\in 2, 3, \ldots,$ length(arr) **do**
        if x $<$ arr$[$i$]$ **then**
            x $\leftarrow$ arr$[$i$]$
        end if
    end for
    return x
end function

$$T_{\mathsf{Max}}(n) = 4 + 6(n-1) + 1 = 6n - 1$$

An example algorithm

Problem: *Given a non-empty array* arr *of one-word integers, find the largest integer in* arr.

function Max(arr)
  x $\leftarrow$ arr$[1]$
  for i $\in 2, 3, \ldots,$ length(arr) **do**
    if x $<$ arr$[$i$]$ **then**
      x $\leftarrow$ arr$[$i$]$
    end if
  end for
  return x
end function

$$T_{\mathsf{Max}}(n) = 4 + 6(n-1) + 1 = 6n - 1$$

$$S_{\mathsf{Max}}(n) =$$

An example algorithm

Problem: *Given a non-empty array* arr *of one-word integers, find the largest integer in* arr.

```
function Max(arr)
    x ← arr[1]
    for i ∈ 2, 3, . . . , length(arr) do
        if x < arr[i] then
            x ← arr[i]
        end if
    end for
    return x
end function
```

$$T_{\mathsf{Max}}(n) = 4 + 6(n-1) + 1 = 6n - 1$$

$$S_{\mathsf{Max}}(n) = 2$$

Going asymptotic

Asymptotic analysis

Ultimately, we don't actually care about what $T(n), S(n)$ *are*.

Asymptotic analysis

Ultimately, we don't actually care about what $T(n), S(n)$ *are*. What really matters is *how they behave as their input sizes grow*.

Asymptotic analysis

Ultimately, we don't actually care about what $T(n), S(n)$ *are*. What really matters is *how they behave as their input sizes grow*. Our current method focuses too much on the former, and not enough on the latter.

Thus, we make the following assumptions:

Asymptotic analysis

Ultimately, we don't actually care about what $T(n), S(n)$ *are*. What really matters is *how they behave as their input sizes grow*. Our current method focuses too much on the former, and not enough on the latter.

Thus, we make the following assumptions:

- Input sizes can grow arbitrarily large

Asymptotic analysis

Ultimately, we don't actually care about what $T(n), S(n)$ *are*. What really matters is *how they behave as their input sizes grow*. Our current method focuses too much on the former, and not enough on the latter.

Thus, we make the following assumptions:

- Input sizes can grow arbitrarily large
- Constant additions or factors are irrelevant

Asymptotic analysis

Ultimately, we don't actually care about what $T(n), S(n)$ *are*. What really matters is *how they behave as their input sizes grow*. Our current method focuses too much on the former, and not enough on the latter.

Thus, we make the following assumptions:

- Input sizes can grow arbitrarily large
- Constant additions or factors are irrelevant
- Out of multiple n-terms, only the *largest* matters

Asymptotic analysis

Ultimately, we don't actually care about what $T(n), S(n)$ *are*. What really matters is *how they behave as their input sizes grow*. Our current method focuses too much on the former, and not enough on the latter.

Thus, we make the following assumptions:

- Input sizes can grow arbitrarily large
- Constant additions or factors are irrelevant
- Out of multiple n-terms, only the *largest* matters
- We will talk about rate of growth, *not* value

Asymptotic analysis

Ultimately, we don't actually care about what $T(n), S(n)$ *are*. What really matters is *how they behave as their input sizes grow*. Our current method focuses too much on the former, and not enough on the latter.

Thus, we make the following assumptions:

- Input sizes can grow arbitrarily large
- Constant additions or factors are irrelevant
- Out of multiple n-terms, only the *largest* matters
- We will talk about rate of growth, *not* value

We call this kind of analysis *asymptotic*.

Asymptotic analysis

Ultimately, we don't actually care about what $T(n), S(n)$ *are*. What really matters is *how they behave as their input sizes grow*. Our current method focuses too much on the former, and not enough on the latter.

Thus, we make the following assumptions:

- Input sizes can grow arbitrarily large
- Constant additions or factors are irrelevant
- Out of multiple n-terms, only the *largest* matters
- We will talk about rate of growth, *not* value

We call this kind of analysis *asymptotic*. This is because the input can grow as big as we like, and we're only interested in how the running time (or space) changes with the growth of the input.

Reviewing the example

function Max(arr)
  x ← arr[1]
  for i ∈ 2, 3, . . . , length(arr) **do**
    if x < arr[i] **then**
      x ← arr[i]
    end if
  end for
  return x
end function

$$T_{\mathsf{Max}}(n) = 6n - 1$$

$$S_{\mathsf{Max}}(n) = 2$$

Reviewing the example

function Max(arr)
    x $\leftarrow$ arr$[1]$
    for i $\in 2, 3, \ldots,$ length(arr) **do**
        if x $<$ arr$[$i$]$ **then**
            x $\leftarrow$ arr$[$i$]$
        end if
    end for
    return x
end function

$$T_{\text{Max}}(n) = \cancel{6}n \cancel{- 1}$$

$$S_{\text{Max}}(n) = 2$$

Reviewing the example

function Max(arr)
    x ← arr[1]
    for i ∈ 2, 3, . . . , length(arr) **do**
        if x < arr[i] **then**
            x ← arr[i]
        end if
    end for
    return x
end function

$$T_{\mathsf{Max}}(n) = \cancel{6}n \cancel{-1}$$

$$S_{\mathsf{Max}}(n) = \cancel{2}$$

Reviewing the example

```
function Max(arr)
    x ← arr[1]
    for i ∈ 2, 3, . . . , length(arr) do
        if x < arr[i] then
            x ← arr[i]
        end if
    end for
    return x
end function
```

$T_{\text{Max}}(n) = \cancel{6n - 1}$ WRONG!

$S_{\text{Max}}(n) = \cancel{2}$ WRONG!

Reviewing the example

function $\textsf{Max}(\textsf{arr})$
  $\textsf{x} \leftarrow \textsf{arr}[1]$
  for $\textsf{i} \in 2, 3, \ldots, \textsf{length}(\textsf{arr})$ **do**
    if $\textsf{x} < \textsf{arr}[\textsf{i}]$ **then**
      $\textsf{x} \leftarrow \textsf{arr}[\textsf{i}]$
    end if
  end for
  return $\textsf{x}$
end function

$T_{\textsf{Max}}(n)$ is $O(n)$

$S_{\textsf{Max}}(n)$ is $O(1)$

Reviewing the example

function Max(arr)
  x $\leftarrow$ arr$[1]$
  for i $\in 2, 3, \ldots,$ length(arr) **do**
    if x $<$ arr$[$i$]$ **then**
      x $\leftarrow$ arr$[$i$]$
    end if
  end for
  return x
end function

$T_{\text{Max}}(n)$ is $O(n)$

$S_{\text{Max}}(n)$ is $O(1)$

We also say "the time complexity of Max is $O(n)$", or even "Max is $O(n)$".

Reviewing the example

function Max(arr)
  x $\leftarrow$ arr$[1]$
  for i $\in 2, 3, \ldots,$ length(arr) **do**
    if x $<$ arr$[$i$]$ **then**
      x $\leftarrow$ arr$[$i$]$
    end if
  end for
  return x
end function

$T_{\text{Max}}(n)$ is $O(n)$

$S_{\text{Max}}(n)$ is $O(1)$

We also say "the time complexity of Max is $O(n)$", or even "Max is $O(n)$". The second one is a bit unclear, so I'd avoid it.

Benefits of asymptotic complexity

- *Describes* performance on the basis of how our time and space requirements will grow relative the size of the problem, not pegged to a particular machine (or even model!)

Benefits of asymptotic complexity

- *Describes* performance on the basis of how our time and space requirements will grow relative the size of the problem, not pegged to a particular machine (or even model!)
- *Explains* performance by showing us what causes the growth to be as bad (or good) as it is in the algorithm itself

Benefits of asymptotic complexity

- *Describes* performance on the basis of how our time and space requirements will grow relative the size of the problem, not pegged to a particular machine (or even model!)
- *Explains* performance by showing us what causes the growth to be as bad (or good) as it is in the algorithm itself
- Eliminates ‘noise’ constants, focusing our attention where it really counts

Benefits of asymptotic complexity

- *Describes* performance on the basis of how our time and space requirements will grow relative the size of the problem, not pegged to a particular machine (or even model!)
- *Explains* performance by showing us what causes the growth to be as bad (or good) as it is in the algorithm itself
- Eliminates ‘noise’ constants, focusing our attention where it really counts
- Separates algorithms into well-defined groupings

Benefits of asymptotic complexity

- *Describes* performance on the basis of how our time and space requirements will grow relative the size of the problem, not pegged to a particular machine (or even model!)
- *Explains* performance by showing us what causes the growth to be as bad (or good) as it is in the algorithm itself
- Eliminates ‘noise’ constants, focusing our attention where it really counts
- Separates algorithms into well-defined groupings
- Can be determined (and verified) very quickly

Benefits of asymptotic complexity

- *Describes* performance on the basis of how our time and space requirements will grow relative the size of the problem, not pegged to a particular machine (or even model!)
- *Explains* performance by showing us what causes the growth to be as bad (or good) as it is in the algorithm itself
- Eliminates ‘noise’ constants, focusing our attention where it really counts
- Separates algorithms into well-defined groupings
- Can be determined (and verified) very quickly

It’s definitely not perfect though — but that’s material for another talk.

Questions