Introduction and Asymptotic Analysis

E. Han S. Jain W. Suksompong

School of Computing
National University of Singapore

CS3230 Lec01b; Thu, 16 Jan 2025

Overview

Algorithm

Algorithm

"A sequence of unambiguous and executable instructions for solving a problem (given a valid input, obtain a valid output)"

Algorithm

Algorithm

"A sequence of unambiguous and executable instructions for solving a problem (given a valid input, obtain a valid output)"

Let's elaborate:

- What are the valid inputs?
- What is the meaning of unambiguous instructions?
- What is the meaning of executable instructions?
- Are all algorithms deterministic?
- Do all algorithms terminate?

Details

We assume that the algorithm does not need to concern itself with invalid input, e.g., for $Fib(n)$ later, we will assume that n will always be a non-negative Integer

Unambiguous instructions: precisely stated, no room for doubt

Instructions should be executable (implementable) on the target machine, i.e., no magic

*Deterministic: Most of the time, we expect each instruction to be deterministic, though in some cases we allow randomness or nondeterminism (we will talk about randomness/nondeterminism when we deal with them)

*Termination: The algorithm should terminate after finitely many instructions are executed (exception: case by case, explictly stated)

Pseudocode

We can give an algorithm already written as a program in a particular programming language, pros and cons:

- Unambiguous (unless we do not understand that language)
- Clear
- Quite tedious
- Harder to understand

Pseudocode

We can give an algorithm already written as a program in a particular programming language, pros and cons:

- Unambiguous (unless we do not understand that language)
- Clear
- Quite tedious
- Harder to understand

Alternative: Pseudocode (we will use this going forward)

- Slightly informal
- Still precise enough to understand exactly what instructions are, and how to implement it in some programming language

An Example

In Python (source code)

```
A = [(1, 2, 3), (4, 5, 6)]
[*zip(*A)]
```

Do you know what is this?

An Example

In Python (source code)

```
A = [(1, 2, 3), (4, 5, 6)]
[*zip(*A)]
```

Do you know what is this?

In Pseudocode:

An Example

In Python (source code)

```
A = [(1, 2, 3), (4, 5, 6)]
[*zip(*A)]
```

Do you know what is this?

In Pseudocode:

Given a 2D matrix of size $n \times m$, transpose it into an $m \times n$ matrix

Some Properties of Good Algorithms

There can be many possible algorithms for solving a problem

Given the choices, we prefer:

- Correctness (the most important property)
- Efficiency (time/space/resources)
- Generality: Applicable to a wide range of inputs and not dependent on a particular computer/device
- Usability as a 'subroutine' for other problems
- Simplicity: so that it is easy to code, understand, debug, etc.
- Well documented (easy to understand and to extend it)

Some objectives may have trade-offs: simplicity vs efficiency

Design and Analysis of Algorithms

Designing an algorithm is both science and art
You need to know the relevant techniques
But you also need creativity, intuition, perseverance

There is no formula for designing a good efficient algorithm
Every new problem may need a fresh approach
So, learn lots of techniques/strategies/paradigms
By observing the properties of a problem and using the techniques, one can often design a good algorithm for the given problem

Paradigms

- Complete Search (for example, using brute force, backtracking, branch and bound)
- Divide and Conquer (D&C)
- Dynamic Programming (DP)
- Greedy Algorithm
- Deterministic versus non-deterministic strategies
- Iterative Improvement

Problem-Solving

The general steps:

1. Understand the problem
2. Design a method to solve the problem
3. Convert it into an algorithm/pseudocode
4. Choose data structures
5. Prove correctness of the algorithm
6. Analyze the complexity of the algorithm (time/space/resources needed)
7. PS: Implement that correct and efficient algorithm

Fibonacci Numbers

- $Fib(0) = 0$
- $Fib(1) = 1$
- For $n > 1$, $Fib(n) = Fib(n-1) + Fib(n-2)$
- First 10 terms: $0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, \ldots$

Problem: Given n as input, compute $Fib(n)$

Fibonacci Numbers

- $Fib(0) = 0$
- $Fib(1) = 1$
- For $n > 1$, $Fib(n) = Fib(n-1) + Fib(n-2)$
- First 10 terms: $0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, \ldots$

Problem: Given n as input, compute $Fib(n)$

We will look at two algorithms:

- Recursive algorithm
- Iterative algorithm

PS: Yes, there are other (faster) algorithms

Recursive algorithm to compute *Fib*(n)

```
define Fib(n)
  if n <= 1
    return n
  else
    return Fib(n-1)+Fib(n-2)
```

Simple, direct recursive implementation from the *Fib*(n) definition

Problem Solving - with recursive $Fib(n)$

Given K $(1 \leq K \leq 45)$ output $Fib(K-1)$ and $Fib(K)$

If you implement above using your favourite programming language, using the algorithm given in the previous slide, it is likely to take too much time
(we will discuss more on this later)

Iterative algorithm to compute *Fib*(n)

```
define IFib(n)
  if n <= 1
    return n
  else
    prev2 = 0
    prev1 = 1
    for i = 2 to n
      temp = prev1
      prev1 = prev1+prev2
      prev2 = temp
  return prev1
```

Problem Solving - with iterative $IFib(n)$

Given K $(1 \leq K \leq 45)$ output $Fib(K-1)$ and $Fib(K)$

Even if your computer is slow, above is likely to give answer quickly. Why?

Analysis of an Algorithm

We analyze the resources needed by an algorithm:

- Time – in this course, we will mostly concentrate on time
- Space – in this course, we assume all data fits in memory

Analysis of an Algorithm

We analyze the resources needed by an algorithm:

- Time – in this course, we will mostly concentrate on time
- Space – in this course, we assume all data fits in memory

Sometimes, we do trade-offs:

- If space is not an issue, most of the time, we sacrifice (or use more) space to gain faster time
- For some applications (e.g., Big Data), we may have to sacrifice time so that we are able to process the data

Analysis of an Algorithm

We analyze the resources needed by an algorithm:

- Time – in this course, we will mostly concentrate on time
- Space – in this course, we assume all data fits in memory

Sometimes, we do trade-offs:

- If space is not an issue, most of the time, we sacrifice (or use more) space to gain faster time
- For some applications (e.g., Big Data), we may have to sacrifice time so that we are able to process the data

Actual time needed to run an algorithm depends on the machine used, and this is not easy to calculate/measure

Model of Computation: RAM

Random-Access Machine (RAM) model is simple
and close to how real computers work:

- Each instruction takes a constant amount of time: fetch the instruction, execute, store back the results in the memory
- We count the number of basic instructions needed
- The time complexity is based on input size (more details soon)

RAM, Continued

- Word is basic unit of memory
  In this course, you can usually assume each number (or relevant item) can be stored in one word
- RAM is an array of words, storing instructions and data
  It takes one unit of time to access <u>any</u> word (this is important)
- Each arithmetic or logical operation (+, -, *, /, mod, AND, OR, NOT, etc) takes a constant amount of time (note: exponent operation is not constant time – see Divide and Conquer lecture later)
- Details of word size and different time taken by different instructions are important, but USUALLY do not have a large impact; so we usually ignore it, unless it makes a difference
- We need to be careful: when numbers are very large (and thus cannot fit in one word), the complexity depends on the number of bits/words needed to store the number

For our $Fib(n)$ and $IFib(n)$ analysis

For large n, $Fib(n)$, can be very large

To address the above, one can consider computing the Fibonacci numbers modulo some m (for example $2^{wordsize}$)

We omit this detail in our first analysis to simplify discussion

Analysis of recursive algorithm to compute *Fib*(n)

```
define Fib(n)
  if n <= 1
    return n
  else
    return Fib(n-1)+Fib(n-2)
```

Analysis of recursive algorithm to compute *Fib*(n)

```
define Fib(n)
  if n <= 1
    return n
  else
    return Fib(n-1)+Fib(n-2)
```

This ‘brute force’ *Fib*(n) is bad

Analysis of recursive algorithm to compute *Fib*(n)

Let $T(n)$ be the number of operations done by *Fib*(n)

```
define Fib(n)
  if n <= 1
    return n
  else
    return Fib(n-1)+Fib(n-2)
```

This ‘brute force’ *Fib*(n) is bad

Analysis of recursive algorithm to compute *Fib*(n)

```
define Fib(n)
  if n <= 1
    return n
  else
    return Fib(n-1)+Fib(n-2)
```

This 'brute force' *Fib*(n) is bad

Let $T(n)$ be the number of operations done by *Fib*(n)

$T(0) = T(1) = 2$
(if+return)
For $n \geq 2$, $T(n) =$
$T(n-1) + T(n-2) + 8$
(if+else+two function calls+add+two subtractions, return)

Analysis of recursive algorithm to compute $Fib(n)$

```
define Fib(n)
  if n <= 1
    return n
  else
    return Fib(n-1)+Fib(n-2)
```

This 'brute force' $Fib(n)$ is bad

Let $T(n)$ be the number of operations done by $Fib(n)$

$T(0) = T(1) = 2$
(if+return)
For $n \geq 2$, $T(n) = T(n-1) + T(n-2) + 8$
(if+else+two function calls+add+two subtractions, return)

So $T(n) \geq Fib(n)$

Analysis of recursive algorithm to compute $Fib(n)$

```
define Fib(n)
  if n <= 1
    return n
  else
    return Fib(n-1)+Fib(n-2)
```

This 'brute force' $Fib(n)$ is bad

Let $T(n)$ be the number of operations done by $Fib(n)$

$T(0) = T(1) = 2$ (if+return)
For $n \geq 2$, $T(n) = T(n-1) + T(n-2) + 8$ (if+else+two function calls+add+two subtractions, return)

So $T(n) \geq Fib(n)$

We can show that $Fib(n) \geq 2^{\frac{n-2}{2}}$ (How?)
$T(n)$ is exponential in n

How to show that $Fib(n) \geq 2^{\frac{n-2}{2}}$?

$Fib(1) = 1$
Notice that for $n \geq 2$, (including $n \geq 3$)
$Fib(n) = Fib(n-1) + Fib(n-2)$,

How to show that $Fib(n) \geq 2^{\frac{n-2}{2}}$?

$Fib(1) = 1$
Notice that for $n \geq 2$, (including $n \geq 3$)
$Fib(n) = Fib(n-1) + Fib(n-2)$,

$Fib(n) \geq Fib(n-2) + Fib(n-2)$,

How to show that $Fib(n) \geq 2^{\frac{n-2}{2}}$?

$Fib(1) = 1$
Notice that for $n \geq 2$, (including $n \geq 3$)
$Fib(n) = Fib(n-1) + Fib(n-2)$,

$Fib(n) \geq Fib(n-2) + Fib(n-2)$,

$Fib(n) \geq 2 \cdot Fib(n-2)$,
i.e., after two terms, the value of $Fib(n)$ will at least double, i.e.,
$Fib(1), Fib(3), Fib(5), Fib(7), Fib(9), \ldots = 1, 2, 5, 13, 34, \ldots$

How to show that $Fib(n) \geq 2^{\frac{n-2}{2}}$?

$Fib(1) = 1$
Notice that for $n \geq 2$, (including $n \geq 3$)
$Fib(n) = Fib(n-1) + Fib(n-2)$,

$Fib(n) \geq Fib(n-2) + Fib(n-2)$,

$Fib(n) \geq 2 \cdot Fib(n-2)$,
i.e., after two terms, the value of $Fib(n)$ will at least double, i.e.,
$Fib(1), Fib(3), Fib(5), Fib(7), Fib(9), \ldots = 1, 2, 5, 13, 34, \ldots$

Between 1 to n, there are $\lceil \frac{n-2}{2} \rceil$ doubling steps
This takes care of odd vs even n cases

Hence $Fib(n) \geq 2^{(n-2)/2}$

NUS National University of Singapore | Computing

Analysis of iterative algorithm to compute *Fib*(n)

```
define IFib(n)
  if n <= 1
    return n
  else
    prev2 = 0
    prev1 = 1
    for i = 2 to n
      temp = prev1
      prev1 = prev1+prev2
      prev2 = temp
  return prev1
```

This is ‘Dynamic Programming’ (DP) (to be revisited later)

Analysis of iterative algorithm to compute $Fib(n)$

```
define IFib(n)
  if n <= 1
    return n
  else
    prev2 = 0
    prev1 = 1
    for i = 2 to n
      temp = prev1
      prev1 = prev1+prev2
      prev2 = temp
  return prev1
```

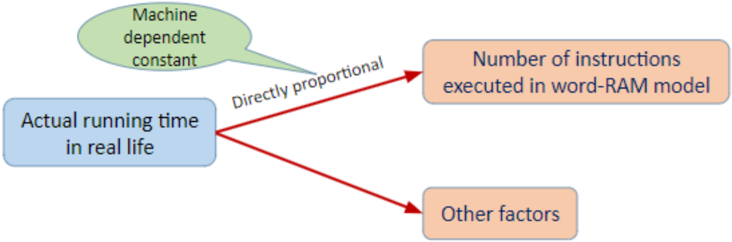
This is 'Dynamic Programming' (DP) (to be revisited later)

For $n \geq 2$,
$T(n) \approx 4 + (n-1) * 5 + 1$
(if+else+two assignments + $(n-1)$ iterations, each takes ≈ 5 steps +return)

So $T(n) \approx 5n$, linear in n

This is much faster than the recursive version that runs exponential in n

Actual Running Time



Running Time of an Algorithm

- We often give the running time in terms of the size of the input (usually parameter n)
- Size of the input can be the number of items (e.g., sorting n Integers) or length of inputs coded in binary (e.g., Integer n in $Fib(n)$ requires $\log n$ bits encoding – details in the second half)

Running Time of an Algorithm

- We often give the running time in terms of the size of the input (usually parameter n)
- Size of the input can be the number of items (e.g., sorting n Integers) or length of inputs coded in binary (e.g., Integer n in $Fib(n)$ requires $\log n$ bits encoding – details in the second half)
- We usually perform these analysis:
  - Worst-case analysis: $T(n)$ is the maximum time needed for any input of size (at most) n
  - Average-case analysis: $T(n)$ is the expected time taken over all inputs of size n; either all inputs are equally probable, or we know the probability distribution over the inputs of size n
  - We usually do not consider best-case analysis, as inputs that trigger best-case are usually not the typical ones

Running Time of an Algorithm

- We often give the running time in terms of the size of the input (usually parameter n)
- Size of the input can be the number of items (e.g., sorting n Integers) or length of inputs coded in binary (e.g., Integer n in $Fib(n)$ requires $\log n$ bits encoding – details in the second half)
- We usually perform these analysis:
  - Worst-case analysis: $T(n)$ is the maximum time needed for any input of size (at most) n
  - Average-case analysis: $T(n)$ is the expected time taken over all inputs of size n; either all inputs are equally probable, or we know the probability distribution over the inputs of size n
  - We usually do not consider best-case analysis, as inputs that trigger best-case are usually not the typical ones
- It is difficult to compute the exact number of operations (as seen earlier), thus we often give upper bounds instead

Question

Which algorithm is more efficient?

Algorithm 1:
$T1(n) = 100n + 1000$

Algorithm 2:
$T2(n) = n^2 + 5$

Question

Which algorithm is more efficient?

Algorithm 1:
$T1(n) = 100n + 1000$

Algorithm 2:
$T2(n) = n^2 + 5$

The answer is 'it depends' (see helper Excel file)

Algorithm 2 can be more efficient on small n, i.e., when $n < 110$

Algorithm 1 is more efficient on large n, especially when $n \geq 110$ (this is more important)

Question

Which algorithm is more efficient?

Algorithm 1:
$T1(n) = 100n + 1000$

Algorithm 2:
$T2(n) = n^2 + 5$

The answer is 'it depends' (see helper Excel file)

Algorithm 2 can be more efficient on small n, i.e., when $n < 110$

Algorithm 1 is more efficient on large n, especially when $n \geq 110$ (this is more important)

Time complexity is MORE IMPORTANT for large-sized input, thus we only compare for asymptotically large values of input size

Asymptotic Analysis

Why we do not measure the actual run time:

- Different machines have different speeds,
  i.e., new gaming desktop is fast vs 10-years old laptop is slow
- Different programming languages have different runtimes,
  i.e., C++ is fast vs Python is slow

Asymptotic Analysis

Why we do not measure the actual run time:

- Different machines have different speeds,
  i.e., new gaming desktop is fast vs 10-years old laptop is slow
- Different programming languages have different runtimes,
  i.e., C++ is fast vs Python is slow

We prefer to do asymptotic analysis:

- For large inputs, how does the runtime behave?
- Comparison of algorithms is based on the asymptotic analysis
- We often ignore lower terms and constant multiplicative factors in the asymptotic analysis

Most common asymptotic notation: Big O (upper bound)

For the following discussion on asymptotics, assume f and g are functions of one parameter n

$f \in O(g)$ if there exists constant $c > 0$ and $n_0 > 0$ such that for all $n \geq n_0 : 0 \leq f(n) \leq c \cdot g(n)$

Interpretation: g is an upper bound on f

Most common asymptotic notation: Big O (upper bound)

For the following discussion on asymptotics, assume f and g are functions of one parameter n

$f \in O(g)$ if there exists constant $c > 0$ and $n_0 > 0$ such that for all $n \geq n_0 : 0 \leq f(n) \leq c \cdot g(n)$

Interpretation: g is an upper bound on f

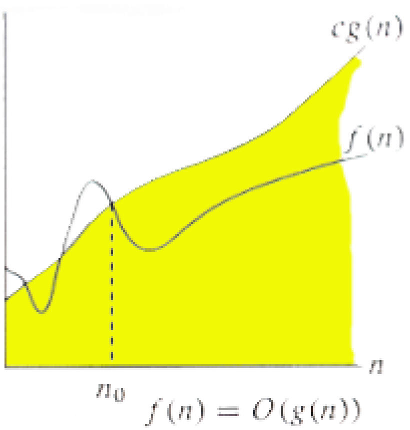
$O(g) = \{f\colon$ there exists constant $c > 0$ and $n_0 > 0$ such that for all $n \geq n_0, 0 \leq f(n) \leq c \cdot g(n)\}$

We sometimes also write $f = O(g)$, though not 100% correct

We frequently write $f(n) = O(g(n))$, though technically, n should not have been used (there can be more than one parameter)

Similarly for other asymptotic notations; PS: we **accept** all versions

Pictorial interpretation of Big O notation



Big O notation is an upper bound notation

So, saying $f(n)$ is <u>at least</u> $O(g(n))$ is not correct

Big O (upper bound)

Example: $100n + 1000 \in O(n^2)$

Big O (upper bound)

Example: $100n + 1000 \in O(n^2)$

- $0 \leq 100n + 1000$ (for any positive n)

Big O (upper bound)

Example: $100n + 1000 \in O(n^2)$

- $0 \leq 100n + 1000$ (for any positive n)
- $0 \leq 100n + 1000 \leq 101n$ (for $n \geq 1000$)

Big O (upper bound)

Example: $100n + 1000 \in O(n^2)$

- $0 \leq 100n + 1000$ (for any positive n)
- $0 \leq 100n + 1000 \leq 101n$ (for $n \geq 1000$)
- $0 \leq 100n + 1000 \leq 101n \leq 101n^2$ (for $n \geq 1000$)
  i.e., we can set $c = 101$ and $n_0 = 1000$

Hence, $100n + 1000 \in O(n^2)$

Big O (upper bound)

Example: $100n + 1000 \in O(n^2)$

- $0 \leq 100n + 1000$ (for any positive n)
- $0 \leq 100n + 1000 \leq 101n$ (for $n \geq 1000$)
- $0 \leq 100n + 1000 \leq 101n \leq 101n^2$ (for $n \geq 1000$)
  i.e., we can set $c = 101$ and $n_0 = 1000$

Hence, $100n + 1000 \in O(n^2)$

But is this upper bound tight?

Big O (upper bound)

Example: $100n + 1000 \in O(n^2)$

- $0 \leq 100n + 1000$ (for any positive n)
- $0 \leq 100n + 1000 \leq 101n$ (for $n \geq 1000$)
- $0 \leq 100n + 1000 \leq 101n \leq 101n^2$ (for $n \geq 1000$)
  i.e., we can set $c = 101$ and $n_0 = 1000$

Hence, $100n + 1000 \in O(n^2)$

But is this upper bound tight?
No, we can also show that $100n + 1000 \in O(n)$ using the same $c = 101$ and $n_0 = 1000$

Is this the only c and n_0 to show that $100n + 1000 \in O(n^2)$?

Big O (upper bound)

Example: $100n + 1000 \in O(n^2)$

- $0 \leq 100n + 1000$ (for any positive n)
- $0 \leq 100n + 1000 \leq 101n$ (for $n \geq 1000$)
- $0 \leq 100n + 1000 \leq 101n \leq 101n^2$ (for $n \geq 1000$)
  i.e., we can set $c = 101$ and $n_0 = 1000$

Hence, $100n + 1000 \in O(n^2)$

But is this upper bound tight?
No, we can also show that $100n + 1000 \in O(n)$ using the same $c = 101$ and $n_0 = 1000$

Is this the only c and n_0 to show that $100n + 1000 \in O(n^2)$?
No, we can also show that $100n + 1000 \in O(n^2)$ with:
$c = 101$ and $n_0 = 1001$ (or any larger n_0),
$c = 1100$ (or any larger c) and $n_0 = 1$, etc.

Question

Let $f(n) = 10n^3 + 5n + 15$ and $g(n) = n^4$

We want to prove that $f(n) \in O(g(n))$ by showing that $0 \leq f(n) \leq c \cdot g(n)$ for all $n \geq n_0$

What should be the appropriate c and n_0? (there are > 1 answers)

A). $c = 2,\ n_0 = 10$

B). $c = 1,\ n_0 = 11$

C). $c = 5,\ n_0 = 1$

D). $c = 1,\ n_0 = 10$

Solution

Reminder: $f(n) = 10n^3 + 5n + 15$ and $g(n) = n^4$
We want to show that $0 \leq f(n) \leq c \cdot g(n)$ for all $n \geq n_0$

Option C). $c = 5$, $n_0 = 1$ **is incorrect**, e.g., for $n = n_0 = 1$:
$f(1) = 30$; $5 \cdot g(1) = 5 \cdot 1 = 5$; so $f(n) > c \cdot g(n)$

Solution

Reminder: $f(n) = 10n^3 + 5n + 15$ and $g(n) = n^4$
We want to show that $0 \leq f(n) \leq c \cdot g(n)$ for all $n \geq n_0$

Option C). $c = 5$, $n_0 = 1$ **is incorrect**, e.g., for $n = n_0 = 1$:
$f(1) = 30$; $5 \cdot g(1) = 5 \cdot 1 = 5$; so $f(n) > c \cdot g(n)$

Option D). $c = 1$, $n_0 = 10$ **is incorrect**, e.g., for $n = n_0 = 10$:
$f(10) = 10\,065$; $1 \cdot g(10) = 1 \cdot 10\,000 = 10\,000$; so $f(n) > c \cdot g(n)$

Solution

Reminder: $f(n) = 10n^3 + 5n + 15$ and $g(n) = n^4$
We want to show that $0 \le f(n) \le c \cdot g(n)$ for all $n \ge n_0$

Option C). $c = 5$, $n_0 = 1$ **is incorrect**, e.g., for $n = n_0 = 1$:
$f(1) = 30$; $5 \cdot g(1) = 5 \cdot 1 = 5$; so $f(n) > c \cdot g(n)$

Option D). $c = 1$, $n_0 = 10$ **is incorrect**, e.g., for $n = n_0 = 10$:
$f(10) = 10\,065$; $1 \cdot g(10) = 1 \cdot 10\,000 = 10\,000$; so $f(n) > c \cdot g(n)$

Option A). $c = 2$, $n_0 = 10$ **is correct**, i.e., for $n \ge 10$, we have:
$10n^3 + (5n + 15) \le 10n^3 + (20n) \le 10n^3 + (10n^3) \le 2n^4$

Solution

Reminder: $f(n) = 10n^3 + 5n + 15$ and $g(n) = n^4$
We want to show that $0 \leq f(n) \leq c \cdot g(n)$ for all $n \geq n_0$

Option C). $c = 5$, $n_0 = 1$ **is incorrect**, e.g., for $n = n_0 = 1$:
$f(1) = 30$; $5 \cdot g(1) = 5 \cdot 1 = 5$; so $f(n) > c \cdot g(n)$

Option D). $c = 1$, $n_0 = 10$ **is incorrect**, e.g., for $n = n_0 = 10$:
$f(10) = 10\,065$; $1 \cdot g(10) = 1 \cdot 10\,000 = 10\,000$; so $f(n) > c \cdot g(n)$

Option A). $c = 2$, $n_0 = 10$ **is correct**, i.e., for $n \geq 10$, we have:
$10n^3 + (5n + 15) \leq 10n^3 + (20n) \leq 10n^3 + (10n^3) \leq 2n^4$

Option B). $c = 1$, $n_0 = 11$ **is correct**, i.e., for $n \geq 11$, we have:
$10 \cdot 11^3 + 5 \cdot 11 + 15 \leq 11 \cdot 11^3$
$5 \cdot 11 + 15 \leq 11^3$ (the gap will grow with larger $n \geq 10.0641$)
Tips: set $c = 1$ and n_0 to be a large value; see if the gap grows

NUS National University of Singapore | Computing

New notation Ω (lower bound)

$f \in \Omega(g)$ if there exists constant $c > 0$ and $n_0 > 0$ such that for all $n \geq n_0 : 0 \leq c \cdot g(n) \leq f(n)$

Interpretation: g is a lower bound on f

Ω (lower bound)

Example: $n^2 \in \Omega(100n + 1000)$

We swap $f(n)$ and $g(n)$ from the earlier Big O example

This is the complementary property of asymptotic notations

Ω (lower bound)

Example: $n^2 \in \Omega(100n + 1000)$
We swap $f(n)$ and $g(n)$ from the earlier Big O example
This is the complementary property of asymptotic notations

- $0 \leq \frac{1}{101} \cdot (100n + 1000) \leq n^2$ for $n \geq 1000$
  i.e., we can set $c = \frac{1}{101}$ and $n_0 = 1000$
  just set this c to be the reciprocal of the c in Big O analysis

Ω (lower bound)

Example: $n^2 \in \Omega(100n + 1000)$
We swap $f(n)$ and $g(n)$ from the earlier Big O example
This is the complementary property of asymptotic notations

- $0 \leq \frac{1}{101} \cdot (100n + 1000) \leq n^2$ for $n \geq 1000$
  i.e., we can set $c = \frac{1}{101}$ and $n_0 = 1000$
  just set this c to be the reciprocal of the c in Big O analysis

Again, there are many other possible c and n_0

Ω (lower bound)

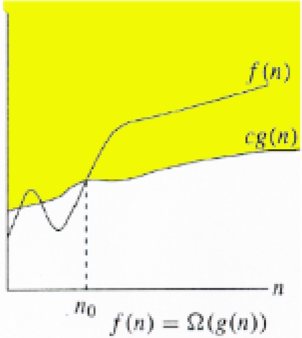
Example: $n^2 \in \Omega(100n + 1000)$
We swap $f(n)$ and $g(n)$ from the earlier Big O example
This is the complementary property of asymptotic notations

- $0 \leq \frac{1}{101} \cdot (100n + 1000) \leq n^2$ for $n \geq 1000$
  i.e., we can set $c = \frac{1}{101}$ and $n_0 = 1000$
  just set this c to be the reciprocal of the c in Big O analysis

Again, there are many other possible c and n_0

PS: We usually have $f(n)$ as the more complex function and $g(n)$ to be the simpler one, i.e., $7n^2 + 5n + 77 \in \Omega(n^2)$

Pictorial interpretation of Ω-notation



New notation Θ (tight bound)

$f \in \Theta(g)$ if there exists constants $c_1, c_2 > 0$ and $n_0 > 0$ such that for all $n \geq n_0 : 0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$

Interpretation: g is a tight bound on f

We will frequently do Θ analysis in CS3230

Θ-notation (tight bound)

Example: $10n^2 + n \in \Theta(n^2)$

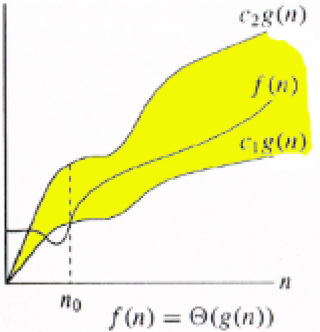
Θ-notation (tight bound)

Example: $10n^2 + n \in \Theta(n^2)$

- $0 \leq \frac{1}{2}n^2 \leq (10n^2 + n) \leq 11n^2$ for $n \geq 2$
  i.e., $c_1 = \frac{1}{2}$, $c_2 = 11$, and $n_0 = 2$
  again, these are not the only valid constants c_1, c_2, and n_0

Hence, $10n^2 + n \in \Theta(n^2)$

Pictorial interpretation of Θ-notation



O, Ω, and Θ

$$\Theta(g) = O(g) \cap \Omega(g)$$

Little-o (strict upper bound)

$f \in o(g)$ if **for any constant** $c > 0$, there exists $n_0 > 0$ such that for all $n \geq n_0 : 0 \leq f(n) < c \cdot g(n)$ (notice **for any constant** $c > 0$ instead of **there exists constant** $c > 0$, and $<$ instead of $\leq$)

PS: some textbooks define Little-o using $\leq c \cdot g(n)$ instead of $< c \cdot g(n)$.
This will only change the chosen c and/or n_0

Little-o (strict upper bound)

$f \in o(g)$ if **for any constant** $c > 0$, there exists $n_0 > 0$ such that for all $n \geq n_0 : 0 \leq f(n) < c \cdot g(n)$ (notice **for any constant** $c > 0$ instead of **there exists constant** $c > 0$, and $<$ instead of $\leq$)

PS: some textbooks define Little-o using $\leq c \cdot g(n)$ instead of $< c \cdot g(n)$.
This will only change the chosen c and/or n_0

Example: $n \in o(n^2)$
For any constant $c > 0$, let $n_0 = 1 + \frac{1}{c}$ (for $0 < c < 1$, setting, $n_0 = 1 + \lceil \frac{1}{c} \rceil$, and for $c \geq 1$, setting $n_0 = 2$ is also ok)
Then, for $n \geq n_0$, $n < c \cdot n^2$

Little-o (strict upper bound)

$f \in o(g)$ if **for any constant** $c > 0$, there exists $n_0 > 0$ such that for all $n \geq n_0 : 0 \leq f(n) < c \cdot g(n)$ (notice **for any constant** $c > 0$ instead of **there exists constant** $c > 0$, and $<$ instead of $\leq$)

PS: some textbooks define Little-o using $\leq c \cdot g(n)$ instead of $< c \cdot g(n)$.
This will only change the chosen c and/or n_0

Example: $n \in o(n^2)$
For any constant $c > 0$, let $n_0 = 1 + \frac{1}{c}$ (for $0 < c < 1$, setting, $n_0 = 1 + \lceil \frac{1}{c} \rceil$, and for $c \geq 1$, setting $n_0 = 2$ is also ok)
Then, for $n \geq n_0$, $n < c \cdot n^2$

But $n^2 - n \notin o(n^2)$
Let's say we pick $c = \frac{1}{2}$ (just need to show one counterexample), for any n_0 and large enough n (that is, $n > max(2, n_0)$) we have:
$n^2 - n > \frac{1}{2}n^2$, because
$\frac{1}{2}n^2 > n$, that is
$n^2 > 2n$

ω (strict lower bound)

$f \in \omega(g)$ if **for any constant** $c > 0$, there exists $n_0 > 0$ such that for all $n \geq n_0 : 0 \leq c \cdot g(n) < f(n)$

Example: $n^2 - 36 \in \omega(n)$
For any constant $c > 0$, let $n_0 > \sqrt{36} + c$,
Then, for $n \geq n_0$, $\quad 0 \leq c \cdot n < n^2 - 36$

Asymptotic Notation: Taking Limits

Assume $f(n), g(n) > 0$, we have:

- $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \Rightarrow f(n) \in o(g(n))$
- $\lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty \Rightarrow f(n) \in O(g(n))$
- $0 < \lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty \Rightarrow f(n) \in \Theta(g(n))$
- $\lim_{n\to\infty} \frac{f(n)}{g(n)} > 0 \Rightarrow f(n) \in \Omega(g(n))$
- $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty \Rightarrow f(n) \in \omega(g(n))$

It is easier to show o, Θ, ω using limits

$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \Rightarrow f(n) \in o(g(n))$

Proof:

By definition of limit, $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$, means
$\forall \epsilon > 0, \exists n_0 > 0$, such that $\forall n \geq n_0$,
$\frac{f(n)}{g(n)} < \epsilon$

$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \Rightarrow f(n) \in o(g(n))$

Proof:

By definition of limit, $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$, means
$\forall \epsilon > 0, \exists n_0 > 0$, such that $\forall n \geq n_0$,
$\frac{f(n)}{g(n)} < \epsilon$

Hence, for any constant $c > 0$ (i.e., we can set $c = \epsilon$), $\exists n_0 > 0$,
such that $\forall n \geq n_0$,
$f(n) < \epsilon \cdot g(n)$, i.e.,
$f(n) < c \cdot g(n)$,
$f(n) \in o(g(n))$

$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \Rightarrow f(n) \in o(g(n))$

Proof:

By definition of limit, $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$, means
$\forall \epsilon > 0, \exists n_0 > 0$, such that $\forall n \geq n_0$,
$\frac{f(n)}{g(n)} < \epsilon$

Hence, for any constant $c > 0$ (i.e., we can set $c = \epsilon$), $\exists n_0 > 0$,
such that $\forall n \geq n_0$,
$f(n) < \epsilon \cdot g(n)$, i.e.,
$f(n) < c \cdot g(n)$,
$f(n) \in o(g(n))$

We will prove at least one other during Tut01

Example

By limit, show that $n^6 + 233n^2 \in \omega(n^2)$

$\lim_{n\to\infty} \frac{n^6+233n^2}{n^2} = \lim_{n\to\infty} \frac{n^4+233}{1} = \infty \Rightarrow f(n) \in \omega(g(n))$

Asymptotic Notation: Some Properties

- Reflexivity: For O, Ω, and Θ,
  $f(n) \in O(f(n))$, similarly for Ω and Θ
- Transitivity: For all five: O, Ω, Θ, o, and ω
  $f(n) \in O(g(n))$ and $g(n) \in O(h(n))$ implies $f(n) \in O(h(n))$
- Symmetry:
  $f(n) \in \Theta(g(n))$ iff $g(n) \in \Theta(f(n))$
- Complementary:
  $f(n) \in O(g(n))$ iff $g(n) \in \Omega(f(n))$
  $f(n) \in o(g(n))$ iff $g(n) \in \omega(f(n))$

We will prove some of these during Tut01

See Asymptotic_Analysis-Useful_Facts.pdf for math refresher

Acknowledgement

The slides are modified from previous editions of this course and similar course elsewhere.

List of credits: Surender Baswana, Arnab Bhattacharyya, Diptarka Chakraborty, Erik D. Demaine, Steven Halim, Sanjay Jain, Lee Wee Sun, Charles E. Leiserson, Leong Hon Wai, Warut Suksompong, Ken Sung, Chang Yi-Jun.