CS5800 Algorithms

Module 1. Asymptotic Notations

Algorithms and Their Efficiency

- What is an "algorithm"? (CLRS 1.1)
  - A specific computational procedure for achieving the **input/output** relationship of a well-specified computational **problem**.
- Can there be multiple algorithms for a specific problem?
  - Of course, they should be ***correct***, first of all.
- What makes an algorithm (A) better (more *efficient*) than another (B)?
  - Does A run faster (*time*) than B on the same problem instance?
  - Does A use less memory (*space*) than B on the same problem instance?

Example of problem/algorithm: Sorting

- Sorting Problem:
  - In English: Sort a given sequence of numbers.
  - In mathematics:
    - Input: a sequence of numbers: $<a_1, a_2, \ldots, a_n>$
    - Output: a permutation of the sequence: $<a_1', a_2', \ldots, a_n'>$ such that $a_i' \leq a_j'$ if $i < j$ for all $1 \leq i < j \leq n$.
- Math Prerequisite Check:
  - Do you know the difference between Set and Sequence?
    - There is no order in set: $\{a_1, a_2, \ldots, a_n\}$ e.g. {1,2} = {2,1}
    - There is the order in sequence: $<a_1, a_2, \ldots, a_n>$ e.g. $< 1,2 > \neq < 2,1 >$
  - Do you know what permutation is? What is the difference between permutation and combination? How many permutations of k elements in a set of size n? How many combinations of k elements in a set of size n?
- Sorting Algorithm:
  - An algorithm that gives the correct sorted sequence for every input sequence.

Why should an algorithm be correct?

- Here is a very fast algorithm for any problem

```
Int VeryFastAlgorithm (object AnyInput)
{ return 0;}
```

- If an algorithm is not correct, why do we bother about efficiency?
- In fact, the definition of an algorithm says that it should generate correct output on **any given input**.
  - Thus, the expression "an incorrect algorithm" is an oxymoron.
- Study on efficiency of an algorithm assumes the algorithm is CORRECT.
- How are we sure that an algorithm is correct "all the time"?
  - Mathematical proof (Duh!)
  - Issue 1: Proving the correctness of an algorithm can be very difficult even for a very simple algorithm.
  - Issue 2: Prerequisite of input should be well-defined.
    - We may have a sorting algorithm that is correct for every integers but is not correct for real numbers.
- Correctness/Efficiency trade-off
  - A slow algorithm that is 100% correct vs a fast algorithm that is 99% correct.
  - What do you mean by 99% correct? It is either correct or wrong!
    - Yes. But, suppose that there is a fast sorting algorithm that is correct if a maximum number is a billion, but is not correct if some number is greater than a billion. And suppose that there is a slow sorting algorithm that is correct for all integers. If you want to sort numbers that is always less then a billion, you may want to use the first algorithm even though it is not a correct algorithm.

Does every problem have an algorithm?

- "Every problem has a solution." (?)
  - Wrong: There is a problem without a solution!
- What do you mean by there is a problem without a solution. You cannot say that there is no solution until you find it!
  - When we say there is a problem without a solution, it means there is a mathematical proof showing that the problem cannot have a solution.
  - E.g. "Halting problem is unsolvable."
- How can we prove there is no solution? Have you searched all universe?
  - Proof technique is a diagonalization (a generalization of the Pigeonhole principle).
- Fun fact: If there are 11 pigeons and 10 pigeonholes, then there must be at least one pigeonhole with at least 2 pigeons!
- What is this simple reasoning related with unsolvable problems?
  - Proof is out of scope of this course. (The scope is computability and complexity)
  - Proof Idea: All algorithms are enumerable (countable).
  - But, if we assume that there is an algorithm for Halting problem, you can find an algorithm that is not in the enumeration. (Similar reasoning that irrational numbers are not countable.)
  - Analogy:
    - Algorithm – Integer numbers (or Pigeonhole)
    - Problem – Real numbers (or Pigeons)
    - There is no 1:1 matching between Integer and Real numbers. Integers (also rational numbers) are countable, but real numbers are uncountable.
    - There are uncountable number of problems, but there are only countable number of algorithms.

How do we measure the efficiency of an algorithm?

- Let's measure the time of an algorithm.
  - How do we measure the time? Time taken for running an algorithm?
    - The same algorithm will run faster in faster CPU, faster memory, etc.
  - Can we measure the number of instructions taken for running algorithm?
    - A line of an instruction does not match with the real instruction in CPU.
    - One simple arithmetic instruction can be multiple microcontroller instructions.
    - The number of microcontroller instructions depend on CPU architecture.
  - Most algorithms will run faster for simple input but will run slower for complex input. Do we have to measure time for each input?
  - Average time? Worst case time?
- Let's measure the space of an algorithm.
  - How do we measure the space? Memory size required for running an algorithm?
  - Big input may require more memory.
  - Average space? Worst case?

Focusing on Temporal Efficiency

- Can comparing *absolute running times* of algorithms be a proper measurement?
  - There are too many factors affecting absolute execution time: Hardware speed, compiler optimization, variations in coding
- See example of comparing insertion sort and merge sort in CLRS 1.2
  - We see that ***constant factors*** in a formula for an algorithm's execution time doesn't really mean much for the algorithm's efficiency.
    - Thus, those constant factors are usually ignored in algorithm analysis.

We will measure the efficiency of an algorithm in terms of input size

- Input size: n
- Sorting algorithm: n numbers.
  - Sorting 2 numbers are much easier than sorting a billion numbers.
- Time and Space will grow (typically) as n becomes bigger.
- We are focusing on how time and space "grow", not on how much it takes exactly.
- Which grows faster? $x^{1.0000001}$ vs $1{,}000{,}000{,}000x$
  - $x^{1.0000001}$ eventually grows faster than $1{,}000{,}000{,}000x$.
- Why do you care for only big numbers? In a real life, we are only dealing with small numbers, aren't we?
  - No. We are dealing with bigger numbers as technology advances.
  - How many websites Google search engine deals with?
  - How many API calls Microsoft Azure receives?
  - How many transactions Amazon.com deals?
  - Example: Someone designed an API of algorithm which takes n^2 searches for n records. This is a GET REST call, which is supposed to return in 60 second time limit. The person thinks that n^2 is enough because, if there are only 100 records, then it only takes 10,000 searches, which takes 0.1 seconds in the server. But, the record grew from 100 to 10,000 records. Now it takes 100,000,000 searches, which takes 1,000 seconds (16.7 minutes), and it is always timed-out. If the person designed algorithm that takes $100n$ searches for n records, it also takes 0.1 seconds. But when the record grew to 10,000, it takes 10 seconds, so the API still works.

One More Abstraction We'd Like

- Compare the two running time functions:
  - $f(n) = n^2 + 1$
  - $g(n) = n \log_{10} n + 1000n + 9999$
  - Say, these are formulas for running times (in milliseconds) of algorithms F and G respectively on problem size n
- Now evaluate the functions for $n = 100$:
  - $f(100) = 10{,}001 \cong 10s$ vs $g(100) = 110{,}199 \cong 110s$
  - So can we say F is better than G?
- What about for $n = 10{,}000$?
  - $f(10{,}000) = 100{,}000{,}001 \cong 100{,}000s \cong 27.7h$
  - $g(10{,}000) = 10{,}049{,}999 \cong 10{,}000s \cong 2.7h$
- Focus on "dominating term" ("rate of growth" in text)

How Algorithms Can Make Differences To Any Problem

Multiple Ways of Computing Fibonacci Numbers

Fibonacci Numbers

- $Fib_0 = 0, Fib_1 = 1$
- $Fib_n = Fib_{n-1} + Fib_{n-2}$ for any integer $n \geq 2$
- 0, 1, __, __, __, __, __, __, __, __, …

- https://en.wikipedia.org/wiki/Fibonacci_number

What is Fib_{14} ?

Simple (Naive) Fibonacci Computation Algorithm

- How did you find Fib_{14} in the previous quiz problem?
- Will the following simple Fibonacci computation code be good enough for finding Fib_{50}?

```
# Simple Python code to compute/print Fib_50.
# (should be readable to everyone)
def fib(n):
    if n <= 1:
        return n
    return fib(n-1) + fib(n-2)

print fib(50)
```


http://pythontutor.com/

Why So Slow?

- How recursive calls are executed?
- Did you see how many times fib(3) was called?
- How about fib(2)? fib(1)? fib(0)?
- Edit code by changing 5 in fib(5) and check # steps.

Unnecessary Redundant Work

- When computing Fib_{50},
  - How many times fib(50) was called?
  - How many times fib(49) was called?
  - How many times fib(48) was called?
  - How many times fib(47) was called?
  - How many times fib(46) was called?
  - How many times fib(45) was called?
  - …
  - How many times fib(3) was called?
  - How many times fib(2) was called?

Analysis of Recursive Fibonacci

- $T(n)$: # execution steps when calling fib(n)
  - "Execution steps" include comparisons, arithmetic operations (e.g., +), assignments, return, … . It may not be the same as # lines of code.
- $T(0)$? $T(1)$?
- For $n \geq 2$, $T(n)$?

```
def fib(n):
    if n <= 1:
        return n
    return fib(n-1) + fib(n-2)
```

- An equation like above is called a *recurrence relation*.
- Closed form solution (non-recurrence):
  - solve $x^2 - x - 1 = 0$, say two roots: φ, ψ, then $T(n) = \frac{\varphi^n - \psi^n}{\varphi - \psi}$

Can We Do Any Better?

- How did you find Fib_{14} in the Checkpoint Quiz?
- Can you implement that idea in code?
- Check the interactable in the next slides
  - Click 'Visualize Execution' to visualize/trace the algorithm's execution.
  - Trace each execution by clicking Forward. Make sure to think about what the next step does before clicking Forward.
  - Compare the two different numbers of steps for the same n.
    - Click 'Edit Code', replace 5 in 'print fib(5)' with another number, and click 'Visualize Execution' again to see the new number of steps for the different n.

Recursive vs. Iterative Fibonacci

```
def fib(n):
    if n <= 1:
        return n
    return fib(n-1) + fib(n-2)

print fib(10)
```

```
def fib(n):
    if n <= 1:
        return n
    i = 2
    fib_i_2 = 0
    fib_i_1 = 1
    fib_i = 1
    while i < n:
        i += 1
        fib_i_2 = fib_i_1
        fib_i_1 = fib_i
        fib_i = fib_i_1 + fib_i_2
    return fib_i
print fib(10)
```


Analysis of Iterative Fibonacci

- $T(0), T(1)$ are the same as recursive.
- For $n \geq 2$, the while loop iterates exactly $n - 2$ times
  - $T(n) =$
- Asymptotic notation for $T(n)$

```
def fib_iterative(n):
    if n <= 1:
        return n

    fib_i_2 = 0   # fib(i-2)
    fib_i_1 = 1   # fib(i-1)
    fib_i = 1     # fib(i)
    i = 2
    while i < n:
        i += 1
        fib_i_2 = fib_i_1
        fib_i_1 = fib_i
        fib_i = fib_i_1 + fib_i_2

    return fib_i
```

Comparing Growths of Linear Time and Exponential Time Complexities

- Assuming each execution step takes $1\mu s$ (microsecond = 10^{-6}s, frequently typed as 'us'), $T_{rec}(n) = 2^n$ and $T_{iter}(n) = 10000n$ (some big coeff),

n	5	10	20	30	40	50	100	1000	10000
Iter. time									
Recur. time									

- See for yourself by computing/printing first 50 Fibonacci numbers using fib_iterative(n)
- Which algorithm would you use? Can we do better?

Are All Recursive Algorithms Bad?

- Fibonacci is an extreme case
- Usually same time complexity
  - But still bigger coefficient
- Higher space complexity
  - Iterative is usually $O(1)$, whereas recursive is usually $O(n)$.
  - If memory is limited, can't use recursive.
- On the other hand, recursive algorithm can be:
  - Easier to understand
  - Smaller code: Iterative code can be extremely complicated in many cases
  - Easier to analyze time complexity

Fast Recursive Fibonacci Algorithm

```
def fib(n, a, b):
    if n == 0:
        return a
    if n == 1:
        return b
    return fib(n-1, b, a+b)

print fib(10, 0, 1)
```

This is called Tail Recursion.

Asymptotic Notations and Algorithm Analysis

How Asymptotic Notations Are Used In Algorithm Analysis

Motivation

- A mathematical tool (framework) used for describing algorithm's efficiency, considering the two abstractions we saw:
  - **Constant factors are not important**.
  - **The most dominating term matters** (growth rate)
- Fairly theoretical, mostly for understanding/gaining insights in analysis of algorithms
  - Formal definitions (big-O, big-Omega, ...)
- Note, in real world, constant factors and non-dominating terms could matter (sometimes seriously)
- Let's begin theoretical journey! (CLRS 3.1)

O-Notation (Big-O)

- We say (define):
  - A function $f(n)$ is in big-O of $g(n)$ (denoted $O(g(n))$) iff (if and only if) there exist positive constants c and n_0 such that $0 \leq f(n) \leq cg(n)$ for all n that's greater than or equal to n_0.
- In other words,
  - $O\big(g(n)\big)$ is a set of all functions (call any such function $x(n)$, to avoid confusion with $f(n)$) where there exist positive constants c and n_0 such that $0 \leq x(n) \leq cg(n)$ for all n that's greater than or equal to n_0.
  - And $f(n)$ is a member of the set $O(g(n))$ (is in the set).
  - Proper notation would be: $f(n) \in O(g(n))$, but we abuse $=$, and write $f(n) = O(g(n))$ most of the time.
- What does all this mean?
  - Study the worked examples in the following slides

Big-O Notation Proof Examples

Time To Prove Big-O Notations Formally

Is $\frac{1}{2}n^2 - 3\mathrm{n} = O(n^2)$?

A function $f(n)$ is in big-O of $g(n)$ (denoted $O(g(n))$) iff (if and only if) there exist positive constants c and n_0 such that $0 \leq f(n) \leq cg(n)$ for all $n \geq n_0$.

- If you think it is, and need to prove it, you must find c and n_0 such that

$$0 \leq \frac{1}{2}n^2 - 3\mathrm{n} \leq \boldsymbol{cn^2}$$

for all $n \geq \boldsymbol{n_0}$

- This is an easy case, after trying a few possible c and n_0 with some intuitions:
  - As long as c is at least $\frac{1}{2}$, the inequality holds for any non-negative n!
  - Thus, we can simply let $c = \frac{1}{2}, n_0 = 6$, which satisfies the definition (the inequality above) of big-Oh.
  - In fact, there are infinitely many valid c and n_0 that can be used for the proof.

Is $100n^2 + 123\text{n} = O(n^2)$?

A function $f(n)$ is in big-O of $g(n)$ (denoted $O(g(n))$) iff (if and only if) there exist positive constants c and n_0 such that $0 \leq f(n) \leq cg(n)$ for all $n \geq n_0$.

- If you think it is, and need to prove it, you must find c and n_0 such that

$$0 \leq 100n^2 + 123n \leq cn^2$$

for all $n \geq n_0$

- This is another easy case, after trying a few possible c and n_0 with some intuitions:
  - You soon realize that c must be greater than 100. Let $c = 101$.
  - Then for what values of n does $100n^2 + 123n \leq 101n^2$?
    - Just solve the inequality, and you get $n \geq 123$, which gives us 123 for n_0.

Is $100n^2 + 123\text{n} = O(n^3)$?

- If you think it is, and need to prove it, you must find c and n_0 such that

$$0 \leq 100n^2 + 123n \leq cn^3$$

for all $n \geq n_0$

- This is another easy case, after trying a few possible c and n_0 with some intuitions:
  - c doesn't have to be very big here, because we have n^3 that grows fast. Let $c = 1$.
  - Then for what values of n is $100n^2 + 123n \leq n^3$?
    - No need to solve the inequality precisely. Just divide both sides by n^2, which gives us $n \geq 100 + \frac{123}{n}$, where $100 + \frac{123}{n} \leq 223$, and this (223) is our n_0.

Is $\frac{1}{200}n^2 = O(n)$?

- If you think it is, and need to prove it, you must find c and n_0 such that
$$0 \leq \frac{1}{200}n^2 \leq cn$$
for all $n \geq n_0$
- However, you soon realize that no matter what values of c and n_0 you choose, there'll be always some n that makes $\frac{1}{200}n^2 > cn$
  - Easy, just pick $n = 200c + 1$ for any c. Doesn't matter what n_0 is (in this case). This is called counterexample.
- Therefore, $\frac{1}{200}n^2 = O(n)$ is not true!
- See it! => https://www.mathway.com/Graph

Lesson From Visualization

Rate of growth of a higher order term (n^2) can't be overcome by a constant factor c multiplied to a lower order term (n), no matter how big c can get.

More Intuitions of Big-O

- Lower order term (e.g, $123n$ in $100n^2 + 123n$) can be always ignored by setting n_0 sufficiently large.
  - Highest order term is the "dominating" term.
- Rule of thumb: Pick the highest order term only, drop the constant factor, and that's your best big-Oh notation for a given function. E.g.,
  - $\frac{1}{2}n^2 - 3\mathrm{n}$: Pick the highest order term only ($\frac{1}{2}n^2$) and drop the constant factor ($\frac{1}{2}$), which gives $O(n^2)$.
  - $100n^2 + 123n$: Same, pick $100n^2$, drop 100, giving $O(n^2)$.

Q: $123n + \frac{1}{10}n^2 + 456 = O(n^2)$?

Q: $n^3 + 4n^2 + 5 = O(n^3)$?

35

Q: $4n^6 + 29n^3 + 100 = O(n^5)$?

36

Q: $99999n^2 + 8888888n + 7777777 = O(\frac{n^2}{99999999} + 777n + 55555)$?

37

Q: $98765n^4 + 4321n^3 + 234n + 567 = O(\frac{n^5}{99999})$?

38

Q: $2n \log_2 n + 3n + 5 = O(n)$?

Bounding Properties of Asymptotic Notations

Useful Intuitions On Asymptotic Notations

Big-O Is Upper Bound

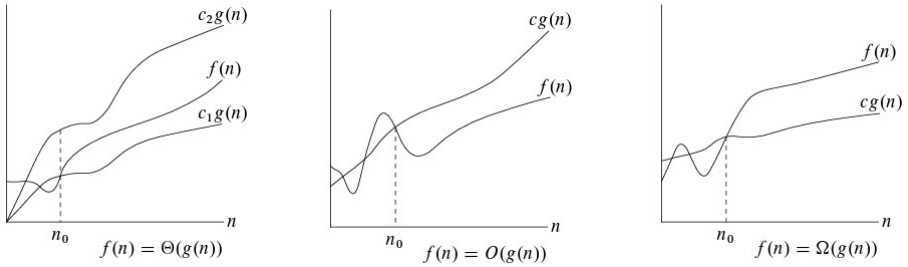
- If $f(n)$ is in $O(n^2)$, then $f(n)$ is also in $O(n^3)$, $O(n^4)$, … .
  - E.g., $f(n) = \frac{1}{200}n^2 + 123n$.
- But, $f(n)$ is not in $O(n)$ for the above example!
- The function in $O(\ )$ for a given function $f(n)$ is an "asymptotic" upper bound.
- There are many upper bounds, and we'd prefer to find the "tight" upper bound.
- In the above example, it's $O(n^2)$.
  - Because it's tighter than all other $O(n^k)$ for any $k > 2$.
  - And also because $f(n)$ cannot be in $O(n)$.

Big-Omega Notation: Asymptotic Lower Bound

- Not as frequently mentioned as big-O, but still important notation for asymptotic lower bound
  - E.g., algorithm A takes at least $\Omega(n)$ time for input size n
- Exactly the same patterned definition, just the different inequality:
  - A function $f(n)$ is in $\Omega(g(n))$ iff there exist positive constants c and n_0 such that $0 \leq \mathrm{cg(n)} \leq f(n)$ for all n that's greater than or equal to n_0.
- Exercise: Prove that $n^4 + 12n^3 - 34n^2 + 56n + 78$
  - Is in $\Omega(n^4)$, in $\Omega(n^3)$, in $\Omega(n^2)$, in $\Omega(n^1)$, and in $\Omega(1)$.
  - But not in $\Omega(n^5)$.

Theta Notation: Asymptotic Tight Bound

- If $f(n) = \mathrm{O}(\mathrm{g}(\mathrm{n}))$ and also $f(n) = \Omega(g(n))$, we say $f(n) = \Theta(g(n))$. In other words,
  - A function $f(n)$ is in $\Theta(g(n))$ iff there exist positive constants c_1, c_2 and n_0 such that $0 \leq \mathrm{c}_1\mathrm{g}(\mathrm{n}) \leq f(n) \leq c_2 g(n)$ for all n that's greater than or equal to n_0.
- Graphic examples of Θ, O, and Ω (CLRS Fig. 3.1)



Misc: Small-o/omega Notations: Non-tight Bounds

- $2n^2 = O(n^2)$ is asymptotically tight, but $2n = O(n^2)$ is not tight.
- Use o-notation to denote an upper bound that is not asymptotically tight. Define:
  - A function $f(n)$ is in $o(g(n))$ iff for any positive constant $c > 0$, there exists a constant $n_0 > 0$ such that $0 \leq f(\mathrm{n}) < \mathrm{cg}(n)$ for all n that's greater than or equal to n_0.
- E.g.: $2n = o(n^2)$, but $2n^2 \neq o(n^2)$.
- Similar definition for $\mathrm{f}(\mathrm{n}) = \omega(g(n))$.

Q: What is the smallest integer k that makes the following statement true?

$$100n^3 + 345n^2 + 678n + 543 = O(n^k)$$

45

Q: What is the smallest integer k that makes the following statement true?

$$2n \log_2 n + 3456789n + 999999 = O(n^k)$$

46

Q: What is the smallest integer k that makes the following statement true?

$$4567n^2 \log_2 n + 89n^3 + 45n + 12345 = O(n^k)$$

Useful Relationships of Asymptotic Notations

Different Asymptotic Notations Are Related

Properties of Asymptotic Notations

- $f(n) = o(g(n))$ iff $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$.
- $f(n) = \omega(g(n))$ iff $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$.
- Transitivity
  - $f(n) = X(g(n))$ and $g(n) = X(h(n))$ implies $f(n) = X(h(n))$. (X can be any of $O, \Omega, \Theta, o, \omega$.
- Reflexivity: $f(n) = X(f(n))$ for $X = O, \Omega, \Theta$.
- Symmetry: $f(n) = \Theta(g(n))$ iff $g(n) = \Theta(f(n))$.
- Transpose symmetry:
  - $f(n) = O(g(n))$ iff $g(n) = \Omega(f(n))$.
  - $f(n) = o(g(n))$ iff $g(n) = \omega(f(n))$.
- Can you prove these properties?

Analogy with Numeric Inequalities

- $f(n) = O(g(n))$ is like $a \leq b$.
- $f(n) = \Omega(g(n))$ is like $a \geq b$.
- $f(n) = \Theta(g(n))$ is like $a = b$.
- $f(n) = o(g(n))$ is like $a < b$.
- $f(n) = \omega(g(n))$ is like $a > b$.

Big O Hierarchy

- $O(n!)$
- $O(3^n)$
- $O(2^n)$
- $O(n^3)$
- $O(n^2)$
- $O(n \log n)$
- $O(n)$
- $O(\sqrt{n})$
- $O(\log n)$
- $O(\log \log n)$
- $O(1)$
- What about $O(n^{1.01})$ vs $O(n \log n)$?
- Polynomial vs Exponential
- Exponential is bad. Polynomial is okay. Linear is good. Logarithmic is great. Constant is the best.
- But, there are many problems that cannot be solved in linear time or even in polynomial time!