Introduction & Asymptotic Run Time

CS325

Class overview

- Asymptotic runtime analysis of algorithms
  - Big-Oh notation
  - Analyzing iterative and recursive algorithms (recurrence relation)
  - Solving recurrence relations
- Prove the correctness of algorithms
  - Some basic proof techniques: proof by induction, proof by contradiction
- Design efficient algorithms
  - Divide and conquer
  - Dynamic programming
  - Greedy algorithms
  - Linear programming
- Limits of computation:
  - Concept of reduction
  - P vs. NP

What is algorithm?

- Algorithm
  - A term coined to honor Al Khwarizmi, a Persian mathematician who wrote the first foundational book on algebra
  - In his book, he moved from solving specific problems to a more general way of solving problems
  - Introduced **precise, unambiguous and correct procedures for solving general problems** – algorithms

A simple example: Insertion sort

Algo **InsertionSort** (A)

for $i \leftarrow 0$ to $length(A) - 1$
  $x \leftarrow A[i]$
  $j \leftarrow i - 1$
  while $j \geq 0$ and $A[j] > x$
    $A[j+1] \leftarrow A[j]$
    $j \leftarrow j - 1$
  end while
  $A[j+1] \leftarrow x$
end for
Return A

- Given an input array A of n numbers, build a sorted array one element at a time
- Each time take one element $A[i]$ and insert into $A[0, \ldots, i-1]$ at the proper spot

6 5 3 1 8 7 2 4

Precise, unambiguous and correct procedures for solving the general problem of **sorting an array of arbitrary size**

Why studying algorithms

- Important for all branches of computer science (and other as well)
  - Computer Networking heavily rely on graph algorithms
  - Bioinformatics builds on dynamic programming algorithms
  - Cryptography – number theoretic algorithms
  - ....
- And it is extremely fun, challenging but fun! And it will help you for your job interview!!!
- Two fundamental and vital questions
  - Is the algorithm correct?
  - Is the algorithm efficient? Or, can we do better?

Efficiency: runtime analysis

What does it mean to perform run-time analysis on an algorithm?

Algo **InsertionSort** (A)

for $i \leftarrow 1$ *to length*(A)
    $x \leftarrow A[i]$
    $j \leftarrow i - 1$
    while $j > 0$ and $A[j] > x$
        $A[j+1] \leftarrow A[j]$
        $j \leftarrow j - 1$
    end while
    $A[j+1] \leftarrow x$
end for
Return A

- Take the insertion sort algorithm as an example
- The practical run time will depend on
  - Implementation of the algorithm
  - The programming language used
  - The processor
  - The input A
  - ….

Run-time analysis

Abstraction #1:

We don't care about the exact time each step takes, we only care about the number of <u>basic operations</u>(comparison, $\times, -, +$ etc.)

Runn time is expressed by counting the number of basic computer steps, as a function of <u>the size of the input</u>.

Algo **Sum** (A)

1. $s = 0$
2. for $i \leftarrow 1$ $to\ length(A)$
3. $\quad s = s + A(i)$
4. end for
5. Return s

- The size of the input is n, the length of A.
- The number of basic operations in this simple algorithm is:
  - Outside loop: 2 steps
  - Inside loop:
    - n iterations
    - 2 steps per iteration
- Total: 2n + 2

Return to Insertion Sort

Algo **InsertionSort** (A)

for $i \leftarrow 1$ $to\ length(A)$
    $x \leftarrow A[i]$
    $j \leftarrow i - 1$
    while $j > 0$ and $A[j] > x$
        $A[j+1] \leftarrow A[j]$
        $j \leftarrow j - 1$
    end while
    $A[j+1] \leftarrow x$
end for
Return A

Outer for loop:

- Run n iterations
- each iteration takes 3 actions

Inner while loop:

- Each iteration takes 2 steps
- How many iterations? It depends
  - Best case (already sorted): 1

    1, 2, 4, 5, 10, 20
  - Worst case (reverse order): $i - 1$

    20, 10, 5, 4, 2, 1

The specific run time depends on the input

Runtime analysis

Abstraction #2: Focus on worst-case.

The run time of an algorithm provides a bound on the run time that holds for every possible input.

- E.g., If an algorithm runs in n^2 time for one input and n time for all other inputs, the runtime is n^2

- Why? Because we want general purpose analysis
- What about "Average-case" analysis ? Much harder to perform this type of analysis
  - Requires heavier machinery about probabilities
  - Requires a good understanding of the domain – what would average input look like?
- "Best-case" analysis? Sorry, that is just wishful thinking …

Worst-case Runtime of Insertion Sort

Algo **InsertionSort** (A)

for $i \leftarrow 1\ to\ length(A)$
    $x \leftarrow A[i]$
    $j \leftarrow i - 1$
    while $j > 0$ and $A[j] > x$
        $A[j+1] \leftarrow A[j]$
        $j \leftarrow j - 1$
    end while
    $A[j+1] \leftarrow x$
end for
Return A

Outer loop

- n iterations
- Each iteration 3 operations

Inner loop

- $(i - 1)$ iterations
- Each iterations 2 operations

Putting it together:

$$3n + 2(0 + 1 + 2 + \cdots + n - 1)$$
$$= 3n + 2\frac{n(n-1)}{2} = 3n + n^2 - n$$
$$= n^2 + 2n$$

Runtime analysis

Abstraction #3:
When consider the runtime, we do not care about multiplicative constants and lower order terms

- Drop the constant scaling factors

$2n$, $3n$ and $3975n$ are considered equivalent

- Drop the lower order terms

$2n^2 + 2n + 493485$ is equivalent to n^2

This greatly simply the analysis process, no need to do exact counting of the number of steps per iterations in our previous analysis

Worst-case Runtime of Insertion Sort

Algo **InsertionSort** (A)

for $i \leftarrow 1\ to\ length(A)$
  $x \leftarrow A[i]$
  $j \leftarrow i - 1$
  while $j > 0$ and $A[j] > x$
    $A[j+1] \leftarrow A[j]$
    $j \leftarrow j - 1$
  end while
  $A[j+1] \leftarrow x$
end for
Return A

Outer loop

- n iterations
- Each iteration constant # of operations

Inner loop

- $(i - 1)$ iterations
- Each iterations constant # of operations

Putting it together:

$$cn + c(0 + 1 + 2 + \cdots + n - 1)$$
$$= cn + c\frac{n(n-1)}{2} = \frac{c}{2}n^2 + \frac{c}{2}n$$
$$\approx n^2$$

More rigorously, this is achieved by **asymptotic analysis of run time**

Asymptotic run time analysis

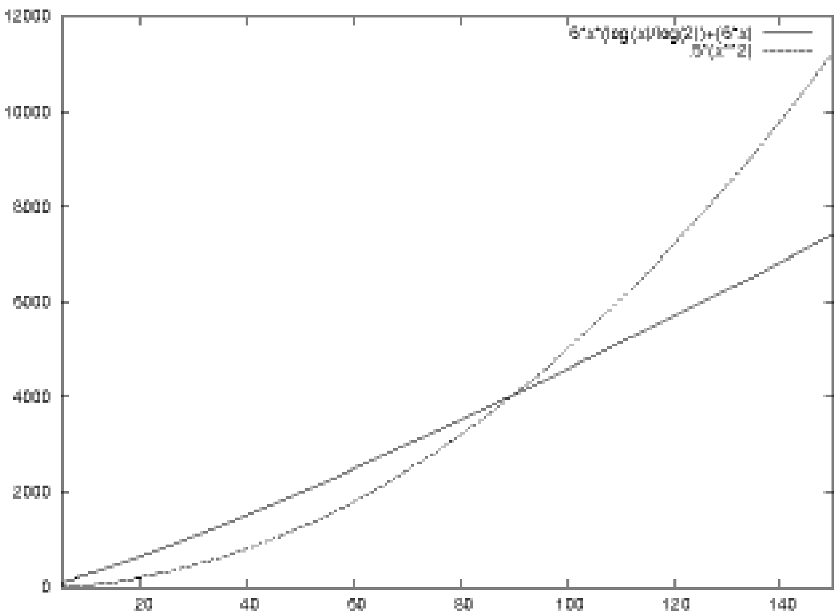
- Focuses on how the run time of an algorithm scales as the input becomes very large
  - It Ignores constant factors and lower-order terms, which become insignificant for large input sizes.
  - It allows us to compare algorithms based on their growth rates, independent of hardware or implementation details.
- Why do we care?
  - Asymptotic analysis helps in understanding the **scalability** of an algorithm

Asymptotic growth

- Example:

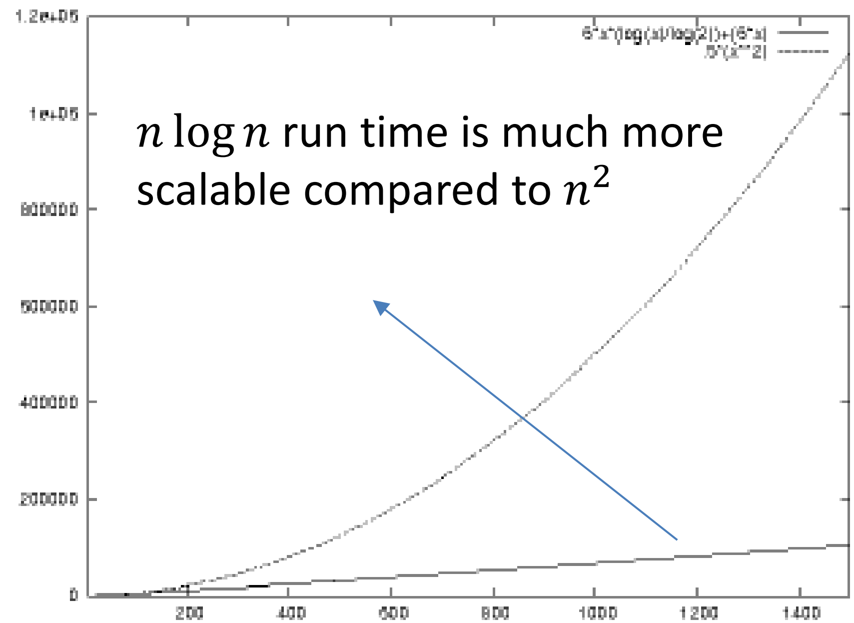
$$6n\log_2 n \; + 6n \quad \text{versus} \qquad \frac{1}{2}n^2$$

Merge sort



Small n(1-150)

Insertion sort



Larger n (200-1400)

Asymptotic Notations: overview

- Big O (O): describes an upper bound on the growth rate
- Little o (o): describes a strict upper bound on the growth rate
- Big Omega (Ω): describes a lower bound on the growth rate
- Theta(Θ): describe a tight bound on the growth rate

In practice, big O is most frequently used in runtime analysis. For example, it's common (and correct) to say the runtime of ***Sum*** is $O(n)$, even though $\Theta(n)$ is also correct, and more precise here.

Asymptotic Notations: Definition

Let $f(n)$ represent the runtime of an algorithm as a function of the input size n; $g(n)$ represents a (reference) function (usually simplified, no constants/lower order terms, e.g., $\log(n)$, n^2 etc.)

Big-O Definition: We say $f(n) = O\big(g(n)\big)$ if and only if there exist some positive constants c and n_0 such that

$$f(n) \leq cg(n)$$

for all $n \geq n_0$

Explanation:

- c is a constant that scales the reference function
- n_0: the threshold input size where the inequality starts to hold

$f(n) = O(g(n))$ indicates that for large enough n, the growth of $f(n)$ does not exceed the growth of $g(n)$

- Intuitively analogous to $f(n) \leq_A g(n)$

Go back to Insertion Sort

Algo **InsertionSort** (A)

```
for i ← 1 to length(A)
    x ← A[i]
    j ← i − 1
    while j > 0 and A[j] > x
        A[j + 1] ← A[j]
        j ← j − 1
    end while
    A[j + 1] ← x
end for
Return A
```

- Runtime:

$$f(n) = n^2 + 2n$$

- Claim:

$$f(n) = O(n^2)$$

Proof:

For any $n \geq \underbrace{2}_{n_0}$, we have:

$$\underbrace{n^2 + 2n}_{f(n)} \leq \underbrace{2}_{c}\underbrace{n^2}_{g(n)}$$

Little o: Definition

We say that $f(n) = o\big(g(n)\big)$ if

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

This means that as n becomes large, $f(n)$ grows strictly slower, and becomes neglectable compared to $g(n)$

Little o: Example

- $f(n) = 3900n + 57800$

$f(n) = o(n^2)$?

$f(n) = o(n)$?

Big-Omega (Ω): Definition

$f(n) = \Omega\big(g(n)\big)$ if and only if there exist constants c, n_0 such that

$$f(n) \geq cg(n)$$

for all $n \geq n_0$

Intuitive Meaning: $f(n)$ grows no slower than $g(n)$

Analogous to $f(n) \geq_A g(n)$

Big Omega: Example

- $f(n) = 0.5n^3 + 0.1\,n^2$

$f(n) = \Omega(n^2)$?

$f(n) = \Omega(n^3)$?

$f(n) = \Omega(n)$?

$f(n) = \Omega(n^4)$?

Theta (θ): Definition

$f(n) = \theta\big(g(n)\big)$ if and only if

there exist constants c_1, c_2, n_0 such that

$$c_1 g(n) \leq f(n) \leq c_2 g(n)$$

for all $n \geq n_0$

Or equivalently $f(n) = O\big(g(n)\big)$ and $f(n) = \Omega\big(g(n)\big)$

Intuitive Meaning: $f(n)$ grows at the same rate (asymptotically equivalent) as $g(n)$

Analogous to $f(n) =_A g(n)$

θ: Example

- $f(n) = a_k n^k + a_{k-1} n^{k-1} + \cdots + a_0$

$f(n) = \theta\left(n^k\right)$?

Relation between O, o, Ω, θ

- $f = \theta(g) \Leftrightarrow$
  $$f = O(g), f = \Omega(g)$$
- $f = o(g) \Rightarrow$
  $$f = O(g)$$
- $f = O(g) \Leftrightarrow$
  $$g = \Omega(f)$$
- $f = \theta(g) \Leftrightarrow$
  $$g = \theta(f)$$

Limits are useful

Consider the value of $\lim_{n \to \infty} \frac{f(n)}{g(n)}$

∞	Nonzero constant c	0
Meaning: with large n $g(n)$ is neglectable compared to $f(n)$	Meaning: with large n $g(n)$ and $f(n)$ have similar growth rate	Meaning: with large n $f(n)$ is neglectable compared to $g(n)$
$f(n) = \Omega(g(n))$	$f(n) = \theta(g(n))$	$f(n) = o(g(n))$

Examples

$$f(n) = a_k n^k + a_{k-1} n^{k-1} + \cdots + a_0$$

$$g_1(n) = n^k, \qquad g_2(n) = n^{k-1}$$

$$\lim_{n\to\infty} \frac{f(n)}{g_1(n)} = \lim_{n\to\infty} \left(a_k + \frac{a_{k-1}}{n} + \cdots + \frac{a_0}{n^k} \right) = a_k$$

$$f = O(g_1)? \quad f = \theta(g_1)? \quad f = \Omega(g_1)?$$

$$\lim_{n\to\infty} \frac{f(n)}{g_2(n)} = \lim_{n\to\infty} \left(a_k n + a_{k-1} + \cdots + \frac{a_0}{n^{k-1}} \right) = \infty$$

$$f = O(g_2)? \quad f = \theta(g_2)? \quad f = \Omega(g_2)?$$

Caveats about constants

- We said that in asymptotic analysis we don't care about **(multiplicative)** constants, but some constants are important
- For example: n^2 vs. n^3, 2^n vs. 3^n

- Which constants in the following expressions do we care about in asymptotic run time?

$$2(n+1)^3, \log_4 n, \log(n^5), (\log \mathrm{n})^6, 8^{7(\log_9 n)}$$

Useful facts about logs and exponentials

$$a^{x+y} = a^x \cdot a^y$$

$$a^{2x} = (a^x)^2$$

$$\log_2 a^x = x \log_2 a$$

$$\log(a \cdot b) = \log a + \log b$$

$$\log \frac{a}{b} = \log a - \log b$$

$$\log_a x = \log_a b \log_b x$$

Common Efficiency Class

Class	Name	
1	constant	No reasonable examples, most cases infinite input size requires infinite run time
$\log n$	Logarithmic	Each operation reduces the problem size by half, Must not look at the whole input, or a fraction of the input, otherwise will be linear
n	linear	Algorithms that scans a list of n items (sequential search)
$n \log n$	linearithmic	Many D&C algorithms e.g., merge sort
n^2	quadratic	Double embedded loops, insertion sort
n^3	cubic	Three embedded loops, some linear algebra algo.
2^n	exponential	Typical for algo that generates all subsets of a n element set.
$n!$	factorial	Typical for algo that generates all permutations of a n-element set

Summary

- Run time of an algorithm measures the number of basic operations as a function of the input size
  - Input size : roughly viewed as the number of bits for representing the input. Sorting: n = the size of the array
  - A more nuanced example: addition of arbitrarily large numbers
    - Input size = the number of bits for representing the numbers
- Run time may depend on the input: Best-case, worst-case and average-case
  - worst case is what we care about in general
- Abstraction to asymptotic behavior: ignore the multiplicative constants, and lower order terms
- Asymptotic notations:
  - O – upper bound. E.g., insertion sort: $O(n^2), O(n^3)$ …
  - o – strict upper bound. E.g., insertion sort: $o(n^3)$
  - Ω – lower bound. E.g., insertion sort: $\Omega(n^2), \Omega(n), \Omega(\log n)$…
  - Θ – tight bound. E.g., insertion sort: $\theta(n^2)$, $\theta(n^2 + 4n)$ …

Quick in-class exercise

$T(n) = 2n^2 + 3n$, which of the following statements are true? (check all that apply)

A. $T(n) = O(n)$

B. $T(n) = O(n^3)$

C. $T(n) = \Omega(n)$

D. $T(n) = \theta(n^2)$

Quick in-class practice

$f(n) = 2^{n+10} \quad g(n) = 2^n$

Which of the followings are correct?

A. $f(n) = O(g(n))$

B. $f(n) = \theta(g(n))$

C. $f(n) = \Omega(g(n))$

D. $f(n) = o(g(n))$

Next lecture

- We will use MergeSort as an example to introduce
  - Run time analysis of recursive algorithms
  - Proof of correctness for recursive algorithms using proof by induction
- Please watch the posted prep videos before class