CSCI 3230 Data Structures

Algorithm analysis

Weitian Tong, Ph.D.

Department of Computer Science
Georgia Southern University
Website: www.weitiantong.com
Email:wtong@georgiasouthern.edu

Table of contents

Compare two algorithms

Compare two algorithms via experiments

- Write a program implementing the algorithm
- Run the program with inputs of varying size and composition
- Use a method like System.currentTimeMillis() to get an accurate measure of the actual running time
- Plot the results

Compare two algorithms via experiments

- Write a program implementing the algorithm
- Run the program with inputs of varying size and composition
- Use a method like System.currentTimeMillis() to get an accurate measure of the actual running time
- Plot the results

Limitations:

- May be difficult to implement the algorithm
- Too many inputs
- In order to compare two algorithms, same hardware and software environments must be used

Compare two algorithms via theorectical model

- Uses a high-level description (pseudo-code) of the algorithm instead of an implementation
- Characterizes running time as a function of the input size, n.
- Takes into account all possible inputs
- Allows us to evaluate the speed of an algorithm independent of the hardware/software environment

Compare two algorithms via theorectical model

- Uses a high-level description (pseudo-code) of the algorithm instead of an implementation
- Characterizes running time as a function of the input size, n.
- Takes into account all possible inputs
- Allows us to evaluate the speed of an algorithm independent of the hardware/software environment

Primitive Operation: Basic computation operations

Examples:

- Evaluating an expression
- Assigning a value to a variable
- Indexing into an array
- Calling a method
- Returning from a method

Counting Primitive Operation

Example: find max from an array

```
1  Algorithm arrayMax(A, n)
2    currentMax = A[0]   // ≈ 2
3    for i = 1 to n - 1 do
4      if A[i] > currentMax then
5        currentMax = A[i]    // ≈ 4(n − 1)
6    return currentMax   // ≈ 1
```

In total, $4n - 1$

- a = Time taken by the fastest primitive operation
- b = Time taken by the slowest primitive operation

Let $T(n)$ be worst-case time of arrayMax. Then

$$a(4n - 1) \leq T(n) \leq b(4n - 1)$$

Changing the hardware/ software environment **only affects** $T(n)$ **by a constant factor**

Example: Selection Sort

```
1  public static int[] selectionSort(int[] arr) {
2    for (i = 0 to n - 1) {
3      int indexOfCurrentMin = i; // ≈ 1 × n
4      for (int j = i + 1 to n ) {
5        if (arr[j] < arr[indexOfCurrentMin]) {
6          indexOfCurrentMin = j;
7        } // ≈ 4 × (n − i) for each i
8      }
9      int currentMin = arr[indexOfCurrentMin]; // ≈ 2 × n
10     arr[indexOfCurrentMin] = arr[i]; // ≈ 3 × n
11     arr[i] = currentMin; // ≈ 2 × n
12   }
13   return arr; // ≈ 1
14 }
```

In total,

$$1 + 8n + 4 \times \sum_{i=0}^{n-1} (n - i)$$

How to compare two functions

Growth factor

Given two algorithms

- $\mathcal{A}_1$: running time / # of primitive operations $f(n) = n + 100$
- $\mathcal{A}_2$: running time / # of primitive operations $g(n) = n^2$

Which algorithm is faster?

Growth factor

Given two algorithms

- $\mathcal{A}_1$: running time / # of primitive operations $f(n) = n + 100$
- $\mathcal{A}_2$: running time / # of primitive operations $g(n) = n^2$

Which algorithm is faster?

Growth Rate: how fast a function becomes larger as n increases?

To compare two algorithms, we prefer the algorithm, whose running time function has smaller growth rate.

How to compare two functions

Classifying Running Time

Classifying Running Time

Name	$T(n)$	$n = 10$	$n = 100$	$n = 1000$	$n = 10000$
Constant	1	1	1	1	1
Logarithmic	$\log n$	3	6	9	13
Square Root	$n^{1/2}$	3	10	31	100
Linear	n	10	100	1000	10000
N-Log-N	$n \log n$	30	600	9000	130000
Quadratic	n^2	100	10000	10^6	10^8
Cubic	n^3	1000	10^6	10^9	10^{12}
Exponential	2^n	1024	10^{30}	10^{300}	10^{3000}

Classifying Running Time

Which are more alike?

$$n^{1000}, \qquad n^2, \qquad 2^n$$

Classifying Running Time

Which are more alike?

$$n^{1000}, \qquad n^2, \qquad 2^n$$

first and second

Classifying Running Time

Which are more alike?

$$n^{1000}, \qquad n^2, \qquad 2^n$$

first and second

How about

$$1000 \cdot n^2, \qquad 3 \cdot n^2, \qquad 2 \cdot n^3$$

Classifying Running Time

Which are more alike?

$$n^{1000}, \qquad n^2, \qquad 2^n$$

first and second

How about

$$1000 \cdot n^2, \qquad 3 \cdot n^2, \qquad 2 \cdot n^3$$

first and second

Classifying Running Time

Which are more alike?

$$n^{1000}, \qquad n^2, \qquad 2^n$$

first and second

How about

$$1000 \cdot n^2, \qquad 3 \cdot n^2, \qquad 2 \cdot n^3$$

first and second

Which algorithm is better?

$$1000 \cdot n^2 + 10^8, \qquad n^3$$

Classifying Running Time

Which are more alike?

$$n^{1000}, \qquad n^2, \qquad 2^n$$

first and second

How about

$$1000 \cdot n^2, \qquad 3 \cdot n^2, \qquad 2 \cdot n^3$$

first and second

Which algorithm is better?

$$1000 \cdot n^2 + 10^8, \qquad n^3$$

first

How to compare two functions

Some Math to Review

Basic math

- Summations: Σ (what is $\sum_{i=1}^{n} i$?)
- Logarithms and Exponents
  - properties of logarithms:
    - $\log_b(xy) = \log_b x + \log_b y$
    - $\log_b(x/y) = \log_b x - \log_b y$
    - $\log_b x^a = a \log_b x$
    - $\log_b a = \log_x a / \log_x b$
  - properties of exponentials:
    - $a^{(b+c)} = a^b a^c$
    - $a^{bc} = (a^b)^c$
    - $a^b / a^c = a^{(b-c)}$
    - $b = a^{\log_a b}$ (why?)
    - $b^c = a^{c \cdot \log_a b}$ (why?)
- Existential ($\exists$) and universal ($\forall$) operators
- Proof techniques (Proof by contradiction, mathematical induction, ...)

How to compare two functions

Asymptotic Notation & Proving Bounds

Asymptotic notations: O, Ω, Θ

How to compare two numerical numbers?

$$\leq, \geq, =, <, >$$

How to compare the growth rates of two functions?

$$O, \Omega, \Theta$$

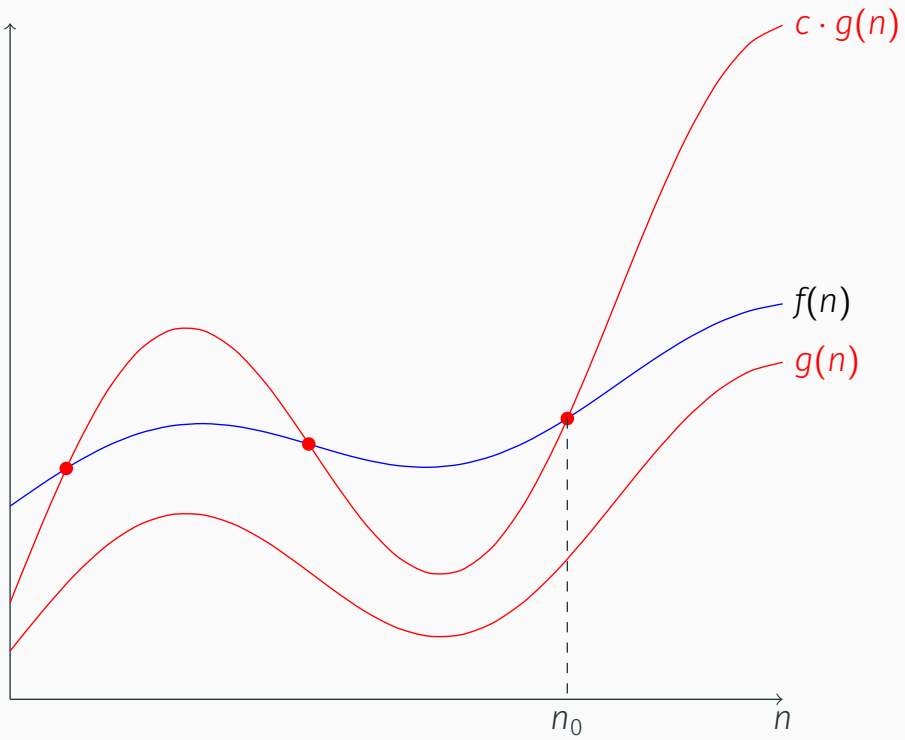
Roughly, $O \approx \leq$, $\Omega \approx \geq$, $\Theta \approx =$

Given two functions $f(n) = n^2$, $g(n) = n^3$, we say $f(n) = O(g(n))$ or $g(n) = \Omega(f(n))$.

Big-Oh notation

Given functions $f(n)$ and $g(n)$, we say that $f(n)$ is $O(g(n))$ if there are positive constants c and n_0 such that

$$f(n) \leq c \cdot g(n), \forall \ \ n > n_0$$



Big-Oh notation

Given functions $f(n)$ and $g(n)$, we say that $f(n)$ is $O(g(n))$ if there are positive constants c and n_0 such that

$$f(n) \leq c \cdot g(n), \forall \ \ n > n_0$$

Example 1: $2n + 10$ is $O(n)$

Big-Oh notation

Given functions $f(n)$ and $g(n)$, we say that $f(n)$ is $O(g(n))$ if there are positive constants c and n_0 such that

$$f(n) \leq c \cdot g(n), \forall \ \ n > n_0$$

Example 1: $2n + 10$ is $O(n)$

$$\begin{aligned} 2n + 10 &\leq cn \\ (c-2)n &\geq 10 \\ n &\geq 10/(c-2) \end{aligned}$$

Pick $c = 3$ and $n_0 = 10$

Big-Oh notation

Given functions $f(n)$ and $g(n)$, we say that $f(n)$ is $O(g(n))$ if there are positive constants c and n_0 such that

$$f(n) \leq c \cdot g(n), \forall \ \ n > n_0$$

Example 1: $2n + 10$ is $O(n)$

$$\begin{aligned} 2n + 10 &\leq cn \\ (c-2)n &\geq 10 \\ n &\geq 10/(c-2) \end{aligned}$$

Pick $c = 3$ and $n_0 = 10$

Practice:

- $2n + 3 \log n + 100$ is $O(n)$
- $3n^3 + 20n^2 + 5$ is $O(n^3)$
- $3 \log n + 1000$ is $O(\log n)$

Big-Oh rules

We generally specify the tightest and simplest bound.

- Only keep the term with the highest growth rate
- Drop constant coefficient

Big-Oh rules

We generally specify the tightest and simplest bound.

- Only keep the term with the highest growth rate
- Drop constant coefficient
- Say "$2n$ is $O(n)$" instead of "$2n$ is $O(n^2)$"
- Say "$3n + 5$ is $O(n)$" instead of "$3n + 5$ is $O(3n)$"

Big-Oh rules

We generally specify the tightest and simplest bound.

- Only keep the term with the highest growth rate
- Drop constant coefficient
- Say "$2n$ is $O(n)$" instead of "$2n$ is $O(n^2)$"
- Say "$3n + 5$ is $O(n)$" instead of "$3n + 5$ is $O(3n)$"

What are the bounds for the following functions under big-Oh?

- $100n^{29} + 2n^{17} + 10^{27}$
- $10^5 n^{100} + 3 \cdot 2^n + 2 \log n$
- $16 + n \log n + 2n^2$

Big-Oh rules

We generally specify the tightest and simplest bound.

- Only keep the term with the highest growth rate
- Drop constant coefficient
- Say "$2n$ is $O(n)$" instead of "$2n$ is $O(n^2)$"
- Say "$3n + 5$ is $O(n)$" instead of "$3n + 5$ is $O(3n)$"

What are the bounds for the following functions under big-Oh?

- $100n^{29} + 2n^{17} + 10^{27}$
- $10^5 n^{100} + 3 \cdot 2^n + 2 \log n$
- $16 + n \log n + 2n^2$

Answer: $O(n^{29})$, $O(2^n)$, $O(n^2)$

Asymptotic Algorithm Analysis

The asymptotic analysis of an algorithm determines the running time in big-Oh notation

To perform the asymptotic analysis,

- count worst-case number of primitive operations executed as a function of the input size
- express this function with big-Oh notation

Example: arrayMax

- arrayMax executes at most $constant \times (4n - 1)$ primitive operations
- arrayMax runs in $O(n)$ time

Trick: Since constant factors and lower-order terms are eventually dropped anyhow, we can disregard them when counting primitive operations

Example: Computing Prefix Averages

The *i*-th prefix average of an array X is the average of the first $(i+1)$ elements of X:

$$A[i] = (X[0] + X[1] + \ldots + X[i])/(i+1)$$

How to computing the array A of prefix averages of another array X?

Example: Computing Prefix Averages

The *i*-th prefix average of an array X is the average of the first $(i+1)$ elements of X:

$$A[i] = (X[0] + X[1] + \ldots + X[i])/(i+1)$$

How to computing the array A of prefix averages of another array X?

```
1  Algorithm prefixAverages1(X, n)
2
3  A = new array of n integers
4  for i = 0 to n - 1 do
5      s = X[0]
6      for j = 1 to i do
7          s = s + X[j]
8      A[i] = s / (i + 1)
9  return A
```

Example: Computing Prefix Averages

The *i*-th prefix average of an array X is the average of the first $(i+1)$ elements of X:

$$A[i] = (X[0] + X[1] + \ldots + X[i])/(i+1)$$

How to computing the array A of prefix averages of another array X?

```
1  Algorithm prefixAverages1(X, n)
2
3  A = new array of n integers
4  for i = 0 to n - 1 do
5      s = X[0]
6      for j = 1 to i do
7          s = s + X[j]
8      A[i] = s / (i + 1)
9  return A
```

Algorithm prefixAverages1 runs in $O(n^2)$ time

Example: Computing Prefix Averages

The *i*-th prefix average of an array X is the average of the first $(i+1)$ elements of X:

$$A[i] = (X[0] + X[1] + \ldots + X[i])/(i+1)$$

How to computing the array A of prefix averages of another array X?

Example: Computing Prefix Averages

The *i*-th prefix average of an array X is the average of the first $(i+1)$ elements of X:

$$A[i] = (X[0] + X[1] + \ldots + X[i])/(i+1)$$

How to computing the array A of prefix averages of another array X?

```
1  // Algorithm prefixAverages2(X, n)
2  A = new array of n integers
3  s = 0
4  for i = 0 to n - 1 do
5      s = s + X[i]
6      A[i] = s / (i + 1)
7  return A
```

Example: Computing Prefix Averages

The *i*-th prefix average of an array X is the average of the first $(i+1)$ elements of X:

$$A[i] = (X[0] + X[1] + \ldots + X[i])/(i+1)$$

How to computing the array A of prefix averages of another array X?

```
1  // Algorithm prefixAverages2(X, n)
2  A = new array of n integers
3  s = 0
4  for i = 0 to n - 1 do
5      s = s + X[i]
6      A[i] = s / (i + 1)
7  return A
```

Algorithm prefixAverages2 runs in $O(n)$ time

Algorithm Complexity vs Problem Complexity

Time Complexity of an Algorithm

Consider the height of tallest person in our class.

Target to give a YES to the following question:

- Is he/she taller than 1.8 m?
- Is he/she shorter than 1.8 m?

Time Complexity of an Algorithm

Consider the height of tallest person in our class.

Target to give a YES to the following question:

- Is he/she taller than 1.8 m?
- Is he/she shorter than 1.8 m?

The time complexity of an algorithm is the largest time required on any input of size n. (**Worst case analysis**.)

- $O(n^2)$: For any input size $n \geq n_0$, the algorithm takes no more than cn^2 time on every input.
- $\Omega(n^2)$: For any input size $n \geq n_0$, the algorithm takes at least cn^2 time on at least one input.
- $\Theta(n^2)$: Do both.

Time Complexity of a Problem

The time complexity of a problem is the time complexity of the fastest algorithm that solves the problem.

- $O(n^2)$: Provide an algorithm that solves the problem in no more than this time.
- $\Omega(n^2)$: Prove that no algorithm can solve it faster
- $\Theta(n^2)$: Do both.

Thank you!

Questions?