

CS121 Data Structures

Asymptotic Analysis

Monika Stepanyan
mstepanyan@aua.am



Spring 2024

Important Data and Statistics

- ▶ TA offline OH on Wednesday, 13:00–14:00, in 415W.
- ▶ TA offline PSS on Saturday, 14:00–16:00, in 306E.

Data Structure and Algorithm Analysis

A *data structure* is a systematic way of organizing and accessing data.

An *algorithm* is a step-by-step procedure for performing some task in a finite amount of time.

*But when can we call a data structure or an algorithm 'good'?
How can we classify them?*

Analysis Tools

1. running time
2. space usage

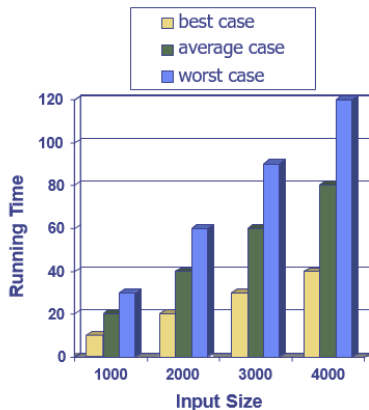
Rationale: Computer solutions should run as fast as possible.

Running time:

- ▶ increases with the input size
- ▶ is affected by the hardware environment and software environment

Running Time

- ▶ Most algorithms transform input objects into output values.
- ▶ The running time of an algorithm typically grows with the input size.
- ▶ Average case time is often difficult to determine.
- ▶ We focus on the worst case running time.
 - ▶ Easier to analyze
 - ▶ Crucial to applications such as games, finance and robotics

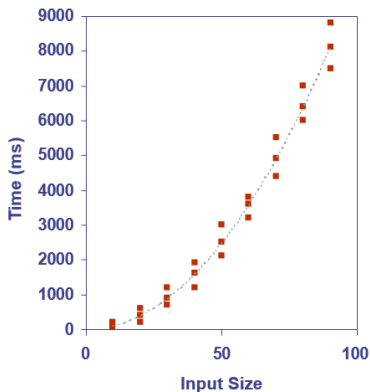


Experimental Studies

- ▶ Write a program implementing the algorithm
- ▶ Run the program with inputs of different size and composition, noting the time needed:
- ▶ Plot the results

```
1 long startTime = System.currentTimeMillis();  
2 /* (run the algorithm) */  
3 long endTime = System.currentTimeMillis();  
4 long elapsed = endTime - startTime;
```

```
// record the starting time  
  
// record the ending time  
// compute the elapsed time
```



Challenges of Experimental Analysis

Three major limitations:

- ▶ It is necessary to implement the algorithm, which may be difficult
- ▶ Results may not be indicative of the running time on other inputs not included in the experiment.
- ▶ In order to compare two algorithms, the same hardware and software environments must be used

Theoretical Analysis

- ▶ Uses a high-level description 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

Pseudocode

- ▶ A high-level description of an algorithm
- ▶ More structured than English prose
- ▶ Less detailed than a program
- ▶ Preferred notation for describing algorithms
- ▶ Hides program design issues

The Random Access Machine (RAM) Model

A **RAM** model consists of

- ▶ A **CPU**
- ▶ A potentially unbounded bank of memory cells, each of which can hold an arbitrary number or character

Memory cells are numbered and accessing any cell in memory takes unit time

Primitive Operations

Primitive operations take no more than *constant* time.

- ▶ Basic computations performed by an algorithm
- ▶ Identifiable in pseudocode
- ▶ Largely independent from the programming language

Examples

- ▶ Performing an arithmetic operation
- ▶ Following an object reference
- ▶ Assigning a value to a variable
- ▶ Comparing two numbers
- ▶ Indexing into an array
- ▶ Calling a method
- ▶ Returning from a method

Counting Primitive Operations

By inspecting the (pseudo)code, we can determine the maximum number of primitive operations executed by an algorithm, as a function of the input size

```
1  /** Returns the maximum value of a nonempty array of numbers. */
2  public static double arrayMax(double[] data) {
3      int n = data.length;
4      double currentMax = data[0];           // assume first entry is biggest (for now)
5      for (int j=1; j < n; j++)              // consider all other entries
6          if (data[j] > currentMax)          // if data[j] is biggest thus far...
7              currentMax = data[j];         // record it as the current max
8      return currentMax;
9  }
```

Counting Primitive Operations

By inspecting the (pseudo)code, we can determine the maximum number of primitive operations executed by an algorithm, as a function of the input size

```
1  /** Returns the maximum value of a nonempty array of numbers. */
2  public static double arrayMax(double[] data) {
3      int n = data.length;
4      double currentMax = data[0];           // assume first entry is biggest (for now)
5      for (int j=1; j < n; j++)              // consider all other entries
6          if (data[j] > currentMax)          // if data[j] is biggest thus far...
7              currentMax = data[j];         // record it as the current max
8      return currentMax;
9  }
```

Steps 3 & 4: 2 ops each

Counting Primitive Operations

By inspecting the (pseudo)code, we can determine the maximum number of primitive operations executed by an algorithm, as a function of the input size

```
1  /** Returns the maximum value of a nonempty array of numbers. */
2  public static double arrayMax(double[] data) {
3      int n = data.length;
4      double currentMax = data[0];           // assume first entry is biggest (for now)
5      for (int j=1; j < n; j++)              // consider all other entries
6          if (data[j] > currentMax)          // if data[j] is biggest thus far...
7              currentMax = data[j];         // record it as the current max
8      return currentMax;
9  }
```

Steps 3 & 4: 2 ops each

Steps 5 & 6: almost $2n$ ops each

Counting Primitive Operations

By inspecting the (pseudo)code, we can determine the maximum number of primitive operations executed by an algorithm, as a function of the input size

```
1  /** Returns the maximum value of a nonempty array of numbers. */
2  public static double arrayMax(double[] data) {
3      int n = data.length;
4      double currentMax = data[0];           // assume first entry is biggest (for now)
5      for (int j=1; j < n; j++)              // consider all other entries
6          if (data[j] > currentMax)          // if data[j] is biggest thus far...
7              currentMax = data[j];         // record it as the current max
8      return currentMax;
9  }
```

Steps 3 & 4: 2 ops each

Steps 5 & 6: almost $2n$ ops each

Step 7: 0 to almost $2n$ ops

Counting Primitive Operations

By inspecting the (pseudo)code, we can determine the maximum number of primitive operations executed by an algorithm, as a function of the input size

```
1  /** Returns the maximum value of a nonempty array of numbers. */
2  public static double arrayMax(double[] data) {
3      int n = data.length;
4      double currentMax = data[0];           // assume first entry is biggest (for now)
5      for (int j=1; j < n; j++)              // consider all other entries
6          if (data[j] > currentMax)          // if data[j] is biggest thus far...
7              currentMax = data[j];         // record it as the current max
8      return currentMax;
9  }
```

Steps 3 & 4: 2 ops each

Steps 5 & 6: almost $2n$ ops each

Step 7: 0 to almost $2n$ ops

Step 8: 1 op

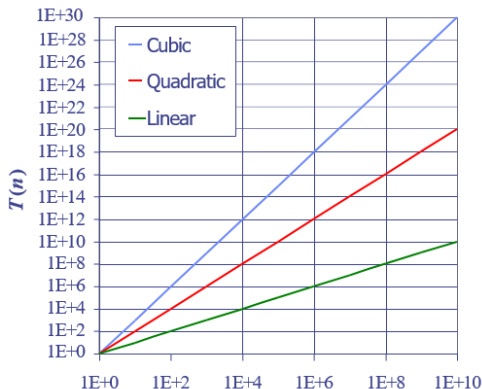
Estimating Running Time

- ▶ Algorithm *arrayMax* executes $6n + 5$ primitive operations in the worst case, $4n + 5$ in the best case. Define:
 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
 $a(4n + 5) \leq T(n) \leq b(6n + 5)$
- ▶ Hence, the running time $T(n)$ is bounded by two linear functions

Seven Important Functions

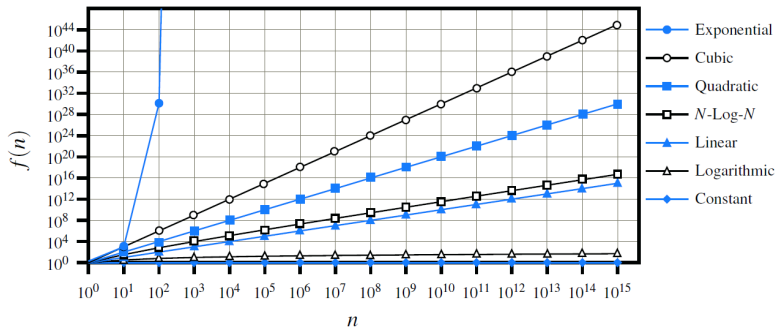
Seven functions that often appear in algorithm analysis:

- ▶ Constant ≈ 1
- ▶ Logarithmic $\approx \log n$
- ▶ Linear $\approx n$
- ▶ N-Log-N $\approx n \log n$
- ▶ Quadratic $\approx n^2$
- ▶ Cubic $\approx n^3$
- ▶ Exponential $\approx 2^n$

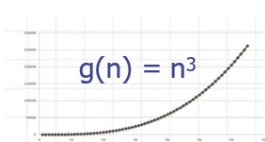
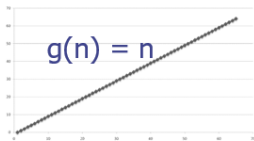
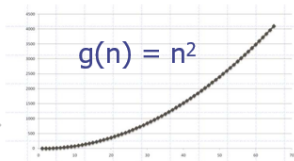
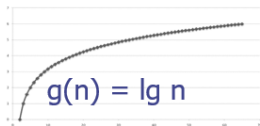
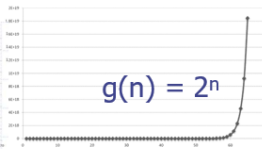
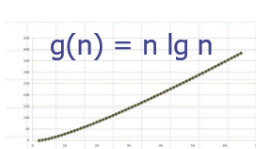
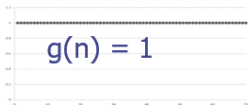


In a log-log plot, the slope of the line corresponds to the growth rate

All Seven Functions



Functions Graphed Using “Normal” Scale




Growth Rate of Running Time

- ▶ Changing the hardware/ software environment
 - ▶ Affects $T(n)$ by a constant factor, but
 - ▶ Does not alter the growth rate of $T(n)$
- ▶ The linear growth rate of the running time $T(n)$ is an intrinsic property of algorithm *arrayMax*

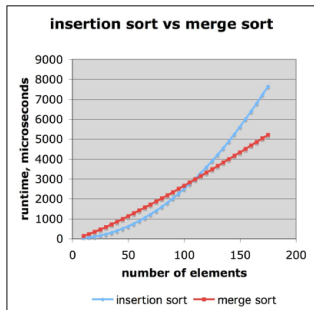
Why Growth Rate Matters

if runtime is...	time for $n + 1$	time for $2n$	time for $4n$
$c \lg n$	$c \lg (n + 1)$	$c (\lg n + 1)$	$c (\lg n + 2)$
$c n$	$c (n + 1)$	$2c n$	$4c n$
$c n \lg n$	$\sim c n \lg n + c n$	$2c n \lg n + 2c n$	$4c n \lg n + 8c n$
$c n^2$	$\sim c n^2 + 2c n$	$4c n^2$	$16c n^2$
$c n^3$	$\sim c n^3 + 3c n^2$	$8c n^3$	$64c n^3$
$c 2^n$	$c 2^{n+1}$	$c 2^{2n}$	$c 2^{4n}$

runtime
quadruples
when
problem
size doubles



Comparison of Two Algorithms



insertion sort is

$$\approx n^2$$

merge sort is

$$\approx n \log n$$

Sort a million items?

roughly 70 hours versus **roughly 40 seconds**

This is a slow machine, but if 100 x as fast then it's 40 minutes versus less than 0.5 seconds

Constant Factors

The growth rate is not affected by

- ▶ constant factors or
- ▶ lower-order terms

Examples

- ▶ $10^2n + 10^5$ is a

Constant Factors

The growth rate is not affected by

- ▶ constant factors or
- ▶ lower-order terms

Examples

- ▶ $10^2 n + 10^5$ is a linear function
- ▶ $10^5 n^2 + 10^8 n$ is a

Constant Factors

The growth rate is not affected by

- ▶ constant factors or
- ▶ lower-order terms

Examples

- ▶ $10^2 n + 10^5$ is a linear function
- ▶ $10^5 n^2 + 10^8 n$ is a quadratic function

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

$$f(n) \leq cg(n) \quad \forall n \geq n_0$$

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

► $2n + 10 \leq cn$

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

$$f(n) \leq cg(n) \quad \forall n \geq n_0$$

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

- ▶ $2n + 10 \leq cn$
- ▶ $(c - 2)n \geq 10$

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

$$f(n) \leq cg(n) \quad \forall n \geq n_0$$

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

- ▶ $2n + 10 \leq cn$
- ▶ $(c - 2)n \geq 10$
- ▶ $n \geq \frac{10}{c-2}$

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

$$f(n) \leq cg(n) \quad \forall n \geq n_0$$

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

- ▶ $2n + 10 \leq cn$
- ▶ $(c - 2)n \geq 10$
- ▶ $n \geq \frac{10}{c-2}$
- ▶ Pick $c = 3$ and $n_0 = 10$

Big-Oh Example

Example: the function n^2 is not $O(n)$

► $n^2 \leq cn$

Big-Oh Example

Example: the function n^2 is not $O(n)$

► $n^2 \leq cn$

► $n \leq c$

Big-Oh Example

Example: the function n^2 is not $O(n)$

- ▶ $n^2 \leq cn$
- ▶ $n \leq c$
- ▶ The above inequality cannot be satisfied since c must be a constant

Big-Oh and Growth Rate

- ▶ The big-Oh notation gives an upper bound on the growth rate of a function
- ▶ The statement “ $f(n)$ is $O(g(n))$ ” means that the growth rate of $f(n)$ is no more than the growth rate of $g(n)$
- ▶ We can use the big-Oh notation to rank functions according to their growth rate

	$f(n)$ is $O(g(n))$	$g(n)$ is $O(f(n))$
$g(n)$ grows more	Yes	No
$f(n)$ grows more	No	Yes
Same growth	Yes	Yes

Big-Oh Rules

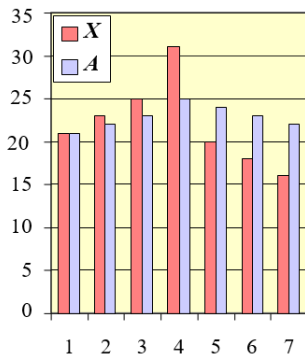
- ▶ If $f(n)$ is a polynomial of degree d , then $f(n)$ is $O(n^d)$, i.e.,
 - ▶ Drop lower-order terms
 - ▶ Drop constant factors
- ▶ Use the smallest possible class of functions
 - ▶ Say “ $2n$ is $O(n)$ ” instead of “ $2n$ is $O(n^2)$ ”
- ▶ Use the simplest expression of the class
 - ▶ Say “ $3n + 5$ is $O(n)$ ” instead of “ $3n + 5$ is $O(3n)$ ”

Asymptotic Algorithm Analysis

- ▶ The asymptotic analysis of an algorithm determines the running time in big-Oh notation
- ▶ To perform the asymptotic analysis
 - ▶ We find the worst-case number of primitive operations executed as a function of the input size
 - ▶ We express this function with big-Oh notation
- ▶ Example:
 - ▶ We say that algorithm *arrayMax* “runs in $O(n)$ time”
- ▶ Since constant factors and lower-order terms are eventually dropped anyhow, we can disregard them when counting primitive operations

Computing Prefix Averages

- ▶ We further illustrate asymptotic analysis with two algorithms for prefix averages
- ▶ The i -th prefix average of an array X is average of the first $(i + 1)$ elements of X :
$$A[i] = \frac{(X[0] + X[1] + \dots + X[i])}{(i+1)}$$
- ▶ Computing the array A of prefix averages of another array X has applications to financial analysis



Example: $X = [2, 4, 6, 8]$, $A = [2, 3, 4, 5]$

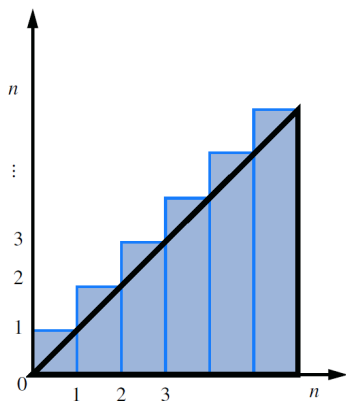
Prefix Averages (Quadratic)

The following algorithm computes prefix averages in quadratic time by applying the definition

```
1  /** Returns an array a such that, for all j, a[j] equals the average of x[0], ..., x[j]. */
2  public static double[] prefixAverage1(double[] x) {
3      int n = x.length;
4      double[] a = new double[n];           // filled with zeros by default
5      for (int j=0; j < n; j++) {
6          double total = 0;                 // begin computing x[0] + ... + x[j]
7          for (int i=0; i <= j; i++)
8              total += x[i];
9          a[j] = total / (j+1);             // record the average
10     }
11     return a;
12 }
```

Arithmetic Progression

- ▶ The running time of *prefixAverage1* is $O(1 + 2 + \dots + n)$
- ▶ The sum of the first n integers is $\frac{n(n+1)}{2}$
- ▶ Thus, algorithm *prefixAverage1* runs in $O(n^2)$ time



Prefix Averages 2 (Linear)

The following algorithm uses a running summation to improve the efficiency

```
1  /** Returns an array a such that, for all j, a[j] equals the average of x[0], ..., x[j]. */
2  public static double[] prefixAverage2(double[] x) {
3      int n = x.length;
4      double[] a = new double[n];           // filled with zeros by default
5      double total = 0;                     // compute prefix sum as x[0] + x[1] + ...
6      for (int j=0; j < n; j++) {
7          total += x[j];                    // update prefix sum to include x[j]
8          a[j] = total / (j+1);             // compute average based on current sum
9      }
10     return a;
11 }
```

Algorithm *prefixAverage2* runs in $O(n)$ time!

Relatives of Big-Oh

big-Omega

- ▶ $f(n)$ is $\Omega(g(n))$ if there is a constant $c > 0$ and an integer constant $n_0 \geq 1$ such that
$$f(n) \geq cg(n) \quad \forall n \geq n_0$$

big-Theta

- ▶ $f(n)$ is $\Theta(g(n))$ if there are constants $c' > 0$ and $c'' > 0$ and an integer constant $n_0 \geq 1$ such that
$$c'g(n) \leq f(n) \leq c''g(n) \quad \forall n \geq n_0$$

Intuition for Asymptotic Notation

big-Oh

- ▶ $f(n)$ is $O(g(n))$ if $f(n)$ is asymptotically **less than or equal to** $g(n)$

big-Omega

- ▶ $f(n)$ is $\Omega(g(n))$ if $f(n)$ is asymptotically **greater than or equal to** $g(n)$

big-Theta

- ▶ $f(n)$ is $\Theta(g(n))$ if $f(n)$ is asymptotically **equal to** $g(n)$

Intuition for Efficiency

When a problem asks you to find an “efficient” solution, as a rule of thumb

- ▶ find the straightforward solution
- ▶ determine its complexity as a function of n
- ▶ think of a way of reducing the complexity to the function directly “preceding” it in the growth order

Examples

- ▶ If the straightforward complexity is $O(n^2)$, then the efficient solution probably runs in

Intuition for Efficiency

When a problem asks you to find an “efficient” solution, as a rule of thumb

- ▶ find the straightforward solution
- ▶ determine its complexity as a function of n
- ▶ think of a way of reducing the complexity to the function directly “preceding” it in the growth order

Examples

- ▶ If the straightforward complexity is $O(n^2)$, then the efficient solution probably runs in $n \log n$.
Is it possible to reduce it ever further?
- ▶ If the straightforward complexity is $O(n)$, then the efficient solution probably runs in

Intuition for Efficiency

When a problem asks you to find an “efficient” solution, as a rule of thumb

- ▶ find the straightforward solution
- ▶ determine its complexity as a function of n
- ▶ think of a way of reducing the complexity to the function directly “preceding” it in the growth order

Examples

- ▶ If the straightforward complexity is $O(n^2)$, then the efficient solution probably runs in $n \log n$.
Is it possible to reduce it ever further?
- ▶ If the straightforward complexity is $O(n)$, then the efficient solution probably runs in $\log n$

Math you need to Review

- ▶ Summations
- ▶ Powers
- ▶ Logarithms
- ▶ Proof techniques

Properties of powers

- ▶ $a^{b+c} = a^b a^c$
- ▶ $a^{bc} = (a^b)^c$
- ▶ $\frac{a^b}{a^c} = a^{b-c}$
- ▶ $b = a^{\log_a b}$
- ▶ $b^c = a^{c \cdot \log_a b}$

Properties of logarithms

- ▶ $\log_b(xy) = \log_b x + \log_b y$
- ▶ $\log_b(x/y) = \log_b x - \log_b y$
- ▶ $\log_b x^a = a \log_b x$
- ▶ $\log_b a = \log_x a / \log_x b$

Summary

Reading

Chapter 4 Algorithm Analysis

Questions?

Acknowledgement: Some contents presented today was part of the Presentation for use with the textbook Data Structures and Algorithms in Java, 6th edition, by M. T. Goodrich, R. Tamassia, and M. H. Goldwasser, Wiley, 2014