

Asymptotic Analysis of Algorithms



EECS2011 X:
Fundamentals of Data Structures
Winter 2023

CHEN-WEI WANG

What You're Assumed to Know

- You will be required to **implement** Java classes and methods, and to **test** their correctness using JUnit.

Review them if necessary:

https://www.eecs.yorku.ca/~jackie/teaching/lectures/index.html#EECS2030_F21

- Implementing classes and methods in Java [Weeks 1 – 2]
- Testing methods in Java [Week 4]
- Also, make sure you know how to trace programs using a **debugger**:

https://www.eecs.yorku.ca/~jackie/teaching/tutorials/index.html#java_from_scratch_w21

- Debugging actions (Step Over/Into/Return) [Parts C – E, Week 2]

Learning Outcomes

This module is designed to help you learn about:

- Notions of *Algorithms* and *Data Structures*
- Measurement of the “goodness” of an algorithm
- Measurement of the *efficiency* of an algorithm
- Experimental measurement vs. *Theoretical* measurement
- Understand the purpose of *asymptotic* analysis.
- Understand what it means to say two algorithms are:
 - equally efficient, **asymptotically**
 - one is more efficient than the other, **asymptotically**
- Given an algorithm, determine its *asymptotic upper bound* .

Algorithm and Data Structure

- A **data structure** is:
 - A systematic way to store and organize data in order to facilitate *access* and *modifications*
 - Never suitable for all purposes: it is important to know its *strengths* and *limitations*
- A **well-specified computational problem** precisely describes the desired *input/output relationship*.
 - **Input:** A sequence of n numbers $\langle a_1, a_2, \dots, a_n \rangle$
 - **Output:** A permutation (reordering) $\langle a'_1, a'_2, \dots, a'_n \rangle$ of the input sequence such that $a'_1 \leq a'_2 \leq \dots \leq a'_n$
 - An *instance* of the problem: $\langle 3, 1, 2, 5, 4 \rangle$
- An **algorithm** is:
 - A solution to a well-specified *computational problem*
 - A *sequence of computational steps* that takes value(s) as *input* and produces value(s) as *output*
- Steps in an *algorithm* manipulate well-chosen *data structure(s)*.

Measuring “Goodness” of an Algorithm

1. *Correctness* :

- Does the algorithm produce the expected output?
- Use JUnit to ensure this.

2. Efficiency:

- *Time Complexity*: processor time required to complete
- *Space Complexity*: memory space required to store data

Correctness is always the priority.

How about efficiency? Is time or space more of a concern?

Measuring Efficiency of an Algorithm

- **Time** is more of a concern than is **storage**.
- Solutions that are meant to be run on a computer should run **as fast as possible**.
- Particularly, we are interested in how **running time** depends on two **input factors**:
 1. **size**
e.g., sorting an array of 10 elements vs. 1m elements
 2. **structure**
e.g., sorting an already-sorted array vs. a hardly-sorted array
- **How do you determine the running time of an algorithm?**
 1. Measure time via **experiments**
 2. Characterize time as a **mathematical function** of the input size

Measure Running Time via Experiments

- Once the algorithm is implemented (e.g., in Java):
 - Execute program on **test inputs** of various **sizes** & **structures**.
 - For each test, record the **elapsed time** of the execution.

```
long startTime = System.currentTimeMillis();  
/* run the algorithm */  
long endTime = System.currenctTimeMillis();  
long elapsed = endTime - startTime;
```

- **Visualize** the result of each test.
- To make **sound statistical claims** about the algorithm's **running time**, the set of input tests must be “reasonably” **complete**.

Example Experiment

- **Computational Problem:**
 - **Input:** A character c and an integer n
 - **Output:** A string consisting of n repetitions of character c
e.g., Given input '*' and 15, output *****.
- **Algorithm 1** using String Concatenations:

```
public static String repeat1(char c, int n) {  
    String answer = "";  
    for (int i = 0; i < n; i++) { answer += c; }  
    return answer; }  
}
```

- **Algorithm 2** using append from StringBuilder:

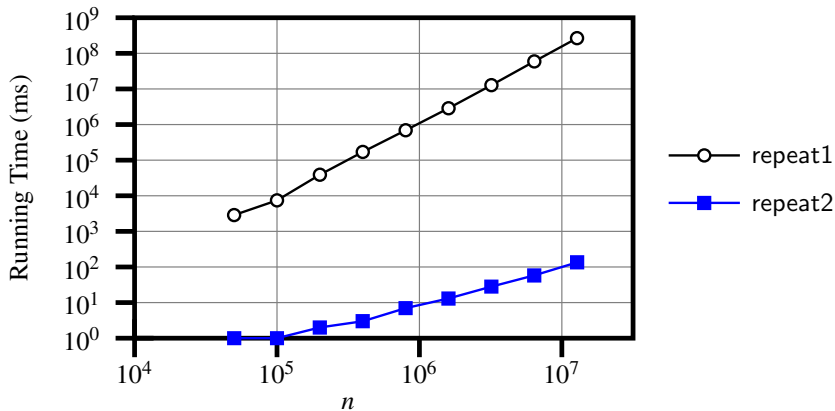
```
public static String repeat2(char c, int n) {  
    StringBuilder sb = new StringBuilder();  
    for (int i = 0; i < n; i++) { sb.append(c); }  
    return sb.toString(); }  
}
```


Example Experiment: Detailed Statistics

n	repeat1 (in ms)	repeat2 (in ms)
50,000	2,884	1
100,000	7,437	1
200,000	39,158	2
400,000	170,173	3
800,000	690,836	7
1,600,000	2,847,968	13
3,200,000	12,809,631	28
6,400,000	59,594,275	58
12,800,000	265,696,421 (\approx 3 days)	135

- As **input size** is doubled, **rates of increase** for both algorithms are **linear**:
 - Running time** of repeat1 increases by \approx 5 times.
 - Running time** of repeat2 increases by \approx 2 times.

Example Experiment: Visualization



Experimental Analysis: Challenges

1. An algorithm must be **fully implemented** (e.g., in Java) in order to study its runtime behaviour **experimentally**.
 - What if our purpose is to **choose among alternative** data structures or algorithms to implement?
 - Can there be a **higher-level analysis** to determine that one algorithm or data structure is more “**superior**” than others?
2. Comparison of multiple algorithms is only **meaningful** when experiments are conducted under the **same** working environment of:
 - **Hardware**: CPU, running processes
 - **Software**: OS, JVM version
3. Experiments can be done only on **a limited set of test inputs**.
 - What if **worst-case** inputs were not included in the experiments?
 - What if “**important**” inputs were not included in the experiments?

Moving Beyond Experimental Analysis

- A better approach to analyzing the **efficiency** (e.g., **running time**) of algorithms should be one that:
 - Allows us to calculate the **relative efficiency** (rather than absolute elapsed time) of algorithms in a way that is **independent of** the hardware and software environment.
 - Can be applied using a **high-level description** of the algorithm (without fully implementing it).
[e.g., Pseudo Code, Java Code (with “tolerances”)]
 - Considers **all** possible inputs (esp. the **worst-case scenario**).
- We will learn a better approach that contains 3 ingredients:
 1. Counting **primitive operations**
 2. Approximating running time as **a function of input size**
 3. Focusing on the **worst-case** input (requiring most running time)

Counting Primitive Operations

A **primitive operation** corresponds to a low-level instruction with a **constant execution time**.

- (Variable) Assignment [e.g., `x = 5;`]
- Indexing into an array [e.g., `a[i]`]
- Arithmetic, relational, logical op. [e.g., `a + b, z > w, b1 && b2`]
- Accessing an attribute of an object [e.g., `acc.balance`]
- Returning from a method [e.g., `return result;`]

Q: Is a **method call** a primitive operation?

A: **Not** in general. It may be a call to:

- a “**cheap**” method (e.g., printing `Hello World`), or
- an “**expensive**” method (e.g., sorting an array of integers)

Example: Counting Primitive Operations (1)

```

1  int findMax (int[] a, int n) {
2      currentMax = a[0];
3      for (int i = 1; i < n; ) {
4          if (a[i] > currentMax) {
5              currentMax = a[i]; }
6          i ++ }
7      return currentMax; }

```

of times $i < n$ in **Line 3** is executed? [n]

of times the loop body (**Line 4** to **Line 6**) is executed? [$n - 1$]

- **Line 2:** 2 [1 indexing + 1 assignment]
- **Line 3:** $n + 1$ [1 assignment + n comparisons]
- **Line 4:** $(n - 1) \cdot 2$ [1 indexing + 1 comparison]
- **Line 5:** $(n - 1) \cdot 2$ [1 indexing + 1 assignment]
- **Line 6:** $(n - 1) \cdot 2$ [1 addition + 1 assignment]
- **Line 7:** 1 [1 return]
- **Total # of Primitive Operations:** $7n - 2$

Example: Counting Primitive Operations (2)

Count the number of primitive operations for

```
1  boolean foundEmptyString = false;
2  int i = 0;
3  while (!foundEmptyString && i < names.length) {
4      if (names[i].length() == 0) {
5          /* set flag for early exit */
6          foundEmptyString = true;
7      }
8      i = i + 1;
9  }
```

- # times the stay condition of the `while` loop is checked?
[between 1 and `names.length + 1`]
[**worst case**: `names.length + 1` times]
- # times the body code of `while` loop is executed?
[between 0 and `names.length`]
[**worst case**: `names.length` times]

From Absolute RT to Relative RT

- Each **primitive operation (PO)** takes approximately the same, constant amount of time to execute. [say t]

The absolute value of t depends on the **execution environment**.

- The **number of primitive operations** required by an algorithm should be **proportional** to its **actual running time** on a specific working environment.

e.g., `findMax (int[] a, int n)` has $7n - 2$ POs

$$RT = (7n - 2) \cdot t$$

Say two algorithms with RT $(7n - 2) \cdot t$ and RT $(10n + 3) \cdot t$.

⇒ It suffices to compare their **relative** running time:

$$7n - 2 \text{ vs. } 10n + 3.$$

- To determine the **time efficiency** of an algorithm, we only focus on their **number of POs**.

Example: Approx. # of Primitive Operations

- Given # of primitive operations counted precisely as $7n - 2$, we view it as

$$7 \cdot n^1 - 2 \cdot n^0$$

- We say
 - n is the **highest power**
 - 7 and 2 are the **multiplicative constants**
 - 2 is the **lower term**
- When approximating a function (considering that input size may be very large):
 - Only** the **highest power** matters.
 - multiplicative constants** and **lower terms** can be dropped.

$\Rightarrow 7n - 2$ is approximately n

Exercise: Consider $7n + 2n \cdot \log n + 3n^2$:

- highest power?**
- multiplicative constants?**
- lower terms?**

$$[n^2]$$

$$[7, 2, 3]$$

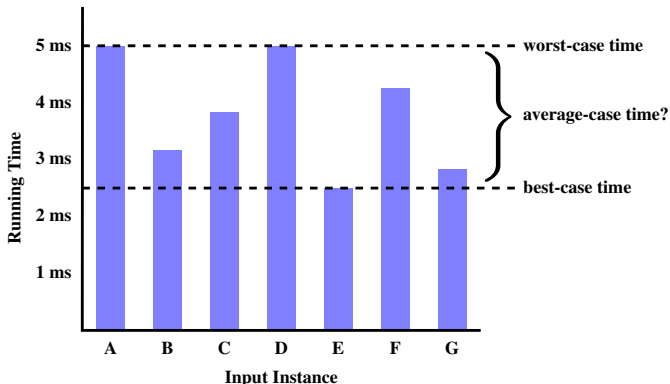
$$[7n + 2n \cdot \log n]$$

Approximating Running Time as a Function of Input Size

Given the **high-level description** of an algorithm, we associate it with a function f , such that $f(n)$ returns the **number of primitive operations** that are performed on an **input of size n** .

- $f(n) = 5$ [constant]
- $f(n) = \log_2 n$ [logarithmic]
- $f(n) = 4 \cdot n$ [linear]
- $f(n) = n^2$ [quadratic]
- $f(n) = n^3$ [cubic]
- $f(n) = 2^n$ [exponential]

Focusing on the Worst-Case Input



- **Average-case** analysis calculates the *expected running time* based on the probability distribution of input values.
- **worst-case** analysis or **best-case** analysis?

What is Asymptotic Analysis?

Asymptotic analysis

- Is a method of describing **behaviour in the limit**:
 - How the **running time** of the algorithm under analysis changes as the **input size** changes without bound
 - e.g., Contrast: $RT_1(n) = n$ vs. $RT_2(n) = n^2$
- Allows us to compare the **relative performance** of alternative algorithms:
 - For large enough inputs, the multiplicative constants and lower-order terms of an exact running time can be disregarded.
 - e.g., $RT_1(n) = 3n^2 + 7n + 18$ and $RT_2(n) = 100n^2 + 3n - 100$ are considered **equally efficient, asymptotically**.
 - e.g., $RT_1(n) = n^3 + 7n + 18$ is considered **less efficient** than $RT_2(n) = 100n^2 + 100n + 2000$, **asymptotically**.

Three Notions of Asymptotic Bounds

We may consider three kinds of *asymptotic bounds* for the *running time* of an algorithm:

- Asymptotic *upper* bound $[O]$
- Asymptotic lower bound $[\Omega]$
- Asymptotic tight bound $[\Theta]$

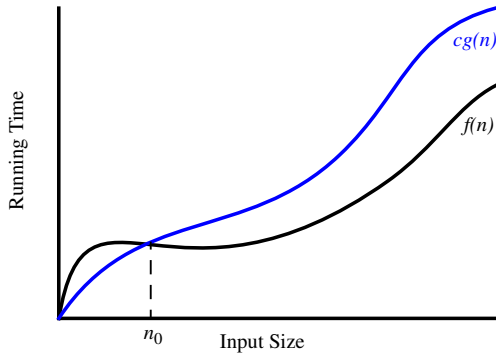
Asymptotic Upper Bound: Definition

- Let $f(n)$ and $g(n)$ be functions mapping positive integers (input size) to positive real numbers (running time).
 - $f(n)$ characterizes the running time of some algorithm.
 - $O(g(n))$:
 - denotes a collection of functions
 - consists of all functions that can be **upper bounded by $g(n)$** , starting at some point, using some constant factor
- $f(n) \in O(g(n))$ if there are:
 - A real **constant** $c > 0$
 - An integer **constant** $n_0 \geq 1$
 such that:

$$f(n) \leq c \cdot g(n) \quad \text{for } n \geq n_0$$

- For each member function $f(n)$ in $O(g(n))$, we say that:
 - $f(n) \in O(g(n))$ [f(n) is a member of "big-O of g(n)"]
 - $f(n)$ **is** $O(g(n))$ [f(n) is "big-O of g(n)"]
 - $f(n)$ **is order of** $g(n)$

Asymptotic Upper Bound: Visualization



From n_0 , $f(n)$ is *upper bounded by* $c \cdot g(n)$, so $f(n)$ is $O(g(n))$.

Asymptotic Upper Bound: Example (1)

Prove: The function $8n + 5$ is $O(n)$.

Strategy: Choose a real constant $c > 0$ and an integer constant $n_0 \geq 1$, such that for every integer $n \geq n_0$:

$$8n + 5 \leq c \cdot n$$

Can we choose $c = 9$? What should the corresponding n_0 be?

n	$8n + 5$	$9n$
1	13	9
2	21	18
3	29	27
4	37	36
5	45	45
6	53	54

...

Therefore, we prove it by choosing $c = 9$ and $n_0 = 5$.

We may also prove it by choosing $c = 13$ and $n_0 = 1$. Why?

Asymptotic Upper Bound: Example (2)

Prove: The function $f(n) = 5n^4 + 3n^3 + 2n^2 + 4n + 1$ is $O(n^4)$.

Strategy: Choose a real constant $c > 0$ and an integer constant $n_0 \geq 1$, such that for every integer $n \geq n_0$:

$$5n^4 + 3n^3 + 2n^2 + 4n + 1 \leq c \cdot n^4$$

$$f(1) = 5 + 3 + 2 + 4 + 1 = 15$$

Choose $c = 15$ and $n_0 = 1$!

Asymptotic Upper Bound: Proposition (1)

If $f(n)$ is a polynomial of degree d , i.e.,

$$f(n) = a_0 \cdot n^0 + a_1 \cdot n^1 + \dots + a_d \cdot n^d$$

and a_0, a_1, \dots, a_d are integers, then $f(n)$ is $O(n^d)$.

- We prove by choosing

$$\begin{aligned} c &= |a_0| + |a_1| + \dots + |a_d| \\ n_0 &= 1 \end{aligned}$$

- We know that for $n \geq 1$: $n^0 \leq n^1 \leq n^2 \leq \dots \leq n^d$
- Upper-bound effect: $n_0 = 1$? $[f(1) \leq (|a_0| + |a_1| + \dots + |a_d|) \cdot 1^d]$

$$a_0 \cdot 1^0 + a_1 \cdot 1^1 + \dots + a_d \cdot 1^d \leq |a_0| \cdot 1^d + |a_1| \cdot 1^d + \dots + |a_d| \cdot 1^d$$

- Upper-bound effect holds? $[f(n) \leq (|a_0| + |a_1| + \dots + |a_d|) \cdot n^d]$

$$a_0 \cdot n^0 + a_1 \cdot n^1 + \dots + a_d \cdot n^d \leq |a_0| \cdot n^d + |a_1| \cdot n^d + \dots + |a_d| \cdot n^d$$

Asymptotic Upper Bound: Proposition (2)

$$O(n^0) \subset O(n^1) \subset O(n^2) \subset \dots$$

If a function $f(n)$ is **upper bounded by** another function $g(n)$ of degree d , $d \geq 0$, then $f(n)$ is also **upper bounded by** all other functions of a **strictly higher degree** (i.e., $d + 1$, $d + 2$, etc.).

e.g., Family of $O(n)$ contains all $f(n)$ that can be **upper bounded by** $g(n) = n$:

$$n^0, 2n^0, 3n^0, \dots$$

[functions with degree 0]

$$n, 2n, 3n, \dots$$

[functions with degree 1]

e.g., Family of $O(n^2)$ contains all $f(n)$ that can be **upper bounded by** $g(n) = n^2$:

$$n^0, 2n^0, 3n^0, \dots$$

[functions with degree 0]

$$n, 2n, 3n, \dots$$

[functions with degree 1]

$$n^2, 2n^2, 3n^2, \dots$$

[functions with degree 2]

Asymptotic Upper Bound: More Examples

- $5n^2 + 3n \cdot \log n + 2n + 5$ is $O(n^2)$ [$c = 15, n_0 = 1$]
- $20n^3 + 10n \cdot \log n + 5$ is $O(n^3)$ [$c = 35, n_0 = 1$]
- $3 \cdot \log n + 2$ is $O(\log n)$ [$c = 5, n_0 = 2$]
 - Why can't n_0 be 1?
 - Choosing $n_0 = 1$ means $\Rightarrow f(1)$ **is** upper-bounded by $c \cdot \log 1$:
 - We have $f(1) = 3 \cdot \log 1 + 2$, which is 2.
 - We have $c \cdot \log 1$, which is 0.
 - $\Rightarrow f(1)$ **is not** upper-bounded by $c \cdot \log 1$ [Contradiction!]
- 2^{n+2} is $O(2^n)$ [$c = 4, n_0 = 1$]
- $2n + 100 \cdot \log n$ is $O(n)$ [$c = 102, n_0 = 1$]

Using Asymptotic Upper Bound Accurately

- Use the big-O notation to characterize a function (of an algorithm's running time) **as closely as possible**.

For example, say $f(n) = 4n^3 + 3n^2 + 5$:

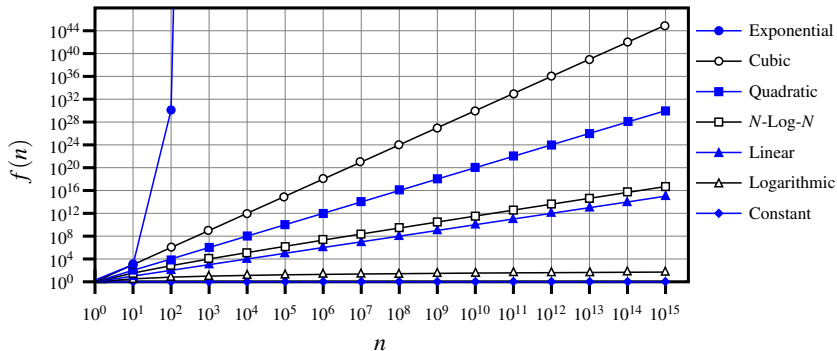
- Recall: $O(n^3) \subset O(n^4) \subset O(n^5) \subset \dots$
 - It is the **most accurate** to say that $f(n)$ is $O(n^3)$.
 - It is **true**, but not very useful, to say that $f(n)$ is $O(n^4)$ and that $f(n)$ is $O(n^5)$.
 - It is **false** to say that $f(n)$ is $O(n^2)$, $O(n)$, or $O(1)$.
- Do **not** include **constant factors** and **lower-order terms** in the big-O notation.

For example, say $f(n) = 2n^2$ is $O(n^2)$, do not say $f(n)$ is $O(4n^2 + 6n + 9)$.

Classes of Functions

upper bound	class	cost
$O(1)$	constant	<i>cheapest</i>
$O(\log(n))$	logarithmic	
$O(n)$	linear	
$O(n \cdot \log(n))$	"n-log-n"	
$O(n^2)$	quadratic	
$O(n^3)$	cubic	
$O(n^k), k \geq 1$	polynomial	
$O(a^n), a > 1$	exponential	<i>most expensive</i>

Rates of Growth: Comparison



Upper Bound of Algorithm: Example (1)

```
1 int maxOf (int x, int y) {  
2     int max = x;  
3     if (y > x) {  
4         max = y;  
5     }  
6     return max;  
7 }
```

- # of primitive operations: 4
 2 assignments + 1 comparison + 1 return = 4
- Therefore, the running time is $O(1)$.
- That is, this is a *constant-time* algorithm.

Upper Bound of Algorithm: Example (2)

```
1  int findMax (int[] a, int n) {  
2      currentMax = a[0];  
3      for (int i = 1; i < n; ) {  
4          if (a[i] > currentMax) {  
5              currentMax = a[i]; }  
6          i ++ }  
7      return currentMax; }
```

- From last lecture, we calculated that the # of primitive operations is $7n - 2$.
- Therefore, the running time is $O(n)$.
- That is, this is a *linear-time* algorithm.

Upper Bound of Algorithm: Example (3)

```
1  boolean containsDuplicate (int[] a, int n) {  
2    for (int i = 0; i < n; ) {  
3      for (int j = 0; j < n; ) {  
4        if (i != j && a[i] == a[j]) {  
5          return true; }  
6        j ++; }  
7      i ++; }  
8    return false; }
```

- Worst case is when we reach Line 8.
- # of primitive operations $\approx c_1 + n \cdot n \cdot c_2$, where c_1 and c_2 are some constants.
- Therefore, the running time is $O(n^2)$.
- That is, this is a *quadratic* algorithm.

Upper Bound of Algorithm: Example (4)

```
1  int sumMaxAndCrossProducts (int[] a, int n) {  
2    int max = a[0];  
3    for(int i = 1; i < n; i++) {  
4      if (a[i] > max) { max = a[i]; }  
5    }  
6    int sum = max;  
7    for (int j = 0; j < n; j++) {  
8      for (int k = 0; k < n; k++) {  
9        sum += a[j] * a[k]; } }  
10   return sum; }
```

- # of primitive operations $\approx (c_1 \cdot n + c_2) + (c_3 \cdot n \cdot n + c_4)$, where c_1 , c_2 , c_3 , and c_4 are some constants.
- Therefore, the running time is $O(n + n^2) = O(n^2)$.
- That is, this is a *quadratic* algorithm.

Upper Bound of Algorithm: Example (5)

```
1  int triangularSum (int[] a, int n) {  
2    int sum = 0;  
3    for (int i = 0; i < n; i++) {  
4      for (int j = i; j < n; j++) {  
5        sum += a[j]; } }  
6    return sum; }
```

- # of primitive operations $\approx n + (n - 1) + \dots + 2 + 1 = \frac{n \cdot (n + 1)}{2}$
- Therefore, the running time is $O\left(\frac{n^2 + n}{2}\right) = O(n^2)$.
- That is, this is a *quadratic* algorithm.

Beyond this lecture ...

- You will be required to **implement** Java classes and methods, and to **test** their correctness using JUnit.

Review them if necessary:

https://www.eecs.yorku.ca/~jackie/teaching/lectures/index.html#EECS2030_F21

- Implementing classes and methods in Java [Weeks 1 – 2]
- Testing methods in Java [Week 4]
- Also, make sure you know how to trace programs using a **debugger**:

https://www.eecs.yorku.ca/~jackie/teaching/tutorials/index.html#java_from_scratch_w21

- Debugging actions (Step Over/Into/Return) [Parts C – E, Week 2]

Index (1)

What You're Assumed to Know

Learning Outcomes

Algorithm and Data Structure

Measuring "Goodness" of an Algorithm

Measuring Efficiency of an Algorithm

Measure Running Time via Experiments

Example Experiment

Example Experiment: Detailed Statistics

Example Experiment: Visualization

Experimental Analysis: Challenges

Moving Beyond Experimental Analysis

Index (2)

Counting Primitive Operations

Example: Counting Primitive Operations (1)

Example: Counting Primitive Operations (2)

From Absolute RT to Relative RT

Example: Approx. # of Primitive Operations

**Approximating Running Time
as a Function of Input Size**

Focusing on the Worst-Case Input

What is Asymptotic Analysis?

Three Notions of Asymptotic Bounds

Asymptotic Upper Bound: Definition

Index (3)

Asymptotic Upper Bound: Visualization

Asymptotic Upper Bound: Example (1)

Asymptotic Upper Bound: Example (2)

Asymptotic Upper Bound: Proposition (1)

Asymptotic Upper Bound: Proposition (2)

Asymptotic Upper Bound: More Examples

Using Asymptotic Upper Bound Accurately

Classes of Functions

Rates of Growth: Comparison

Upper Bound of Algorithm: Example (1)

Upper Bound of Algorithm: Example (2)

Index (4)

Upper Bound of Algorithm: Example (3)

Upper Bound of Algorithm: Example (4)

Upper Bound of Algorithm: Example (5)

Beyond this lecture ...