Asymptotic Analysis of Algorithms

EECS3101 E:
Design and Analysis of Algorithms
Fall 2025

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 $\langle a_1, a_2, \ldots, a_n \rangle$
  - **Output:** A permutation (reordering) $\langle a'_1, a'_2, \ldots, a'_n \rangle$ of the input sequence such that $a'_1 \leq a'_2 \leq \ldots \leq a'_n$
  - 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***
- An ***algorithm*** manipulates some chosen ***data structure(s)***.

Measuring “Goodness” of an Algorithm

1. ***Correctness*** :
   - Does the ***algorithm*** produce the **expected** output?
   - Use ***unit & regression testing*** (e.g., 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 (run on computers) should be ***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

Q. How does one 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 ***test inputs*** should be "***complete***".

  e.g., To experiment with the ***RT*** of a sorting algorithm:
  - **Unreasonable**: **only** consider small-sized and/or almost-sorted arrays
  - **Reasonable**: **also** consider large-sized, randomly-organized arrays

Experimental Analysis: Challenges

1. An algorithm must be ***fully implemented*** (e.g., in Java) in order 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, Version of Compiler
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:
  - Can be applied using a *high-level description* of the algorithm (**without** fully implementing it).
    [e.g., Pseudo Code, Java Code (with "tolerances")]
  - 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.
  - 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*** (***POs***) 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)
- ***RT*** of an ***algorithm*** is approximated as the number of ***POs*** involved (**despite** the execution environment).

From Absolute RT to Relative RT

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

  The <u>absolute</u> value of **t** depends on the ***execution environment***.

<u>Q.</u> How do you relate the ***number of POs*** required by an algorithm and its ***actual RT*** on a specific working environment?

<u>A.</u> ***Number of POs*** should be *<u>proportional</u>* to the actual ***RT***.

$$RT = \mathbf{t} \cdot \textit{number of POs}$$

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

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

- e.g., Say two algorithms with ***RT*** $(7n - 2) \cdot \mathbf{t}$ and ***RT*** $(10n + 3) \cdot \mathbf{t}$:
  It suffices to compare their *<u>relative</u>* running time:

  7n - 2 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*** [e.g., RT $\approx$ f($\boldsymbol{n}$)] (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

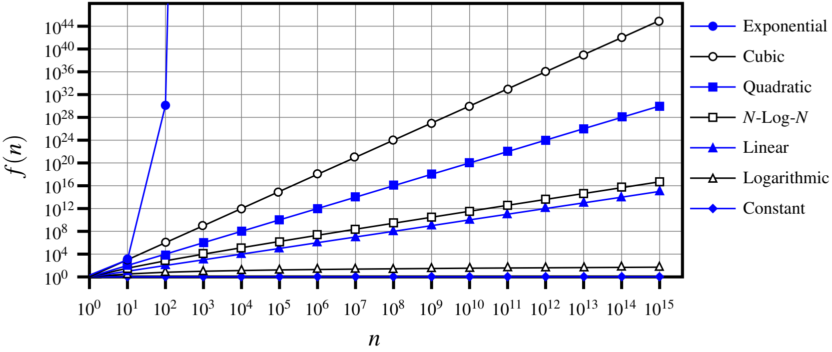
  Exercise: Consider $7n + 2n \cdot log\ n + 3n^2$:
  - ***highest power***? [n^2]
  - ***multiplicative constants***? [7, 2, 3]
  - ***lower terms***? [$7n$, $2n \cdot log\ n$]

Approximating Running Time as a Function of Input Size

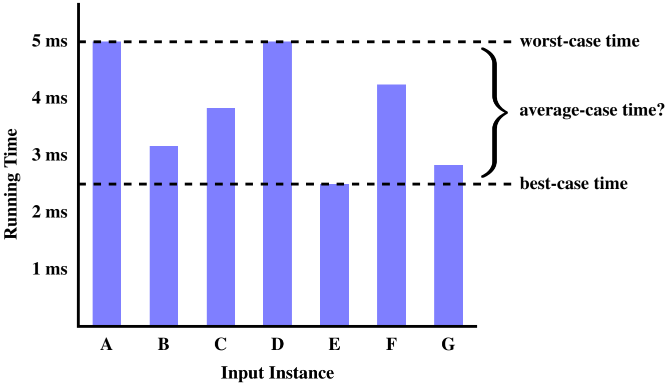
Given the ***high-level description*** of an algorithm, we associate it with a function f, such that $f(n)$ returns the ***number of primitive operations*** that are performed on an ***input of size*** n.

- $f(n) = 5$ [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]

Rates of Growth: Comparison



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 towards 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 $RT_1(n) = 100n^2 + 3n - 100$ are considered **equally efficient**, ***asymptotically***.
  - e.g., $RT_1(n) = n^3 + 7n + 18$ is considered **less efficient** than $RT_1(n) = 100n^2 + 100n + 2000$, ***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 [Ω]
- Asymptotic tight bound [Θ]

Asymptotic Upper Bound: Definition

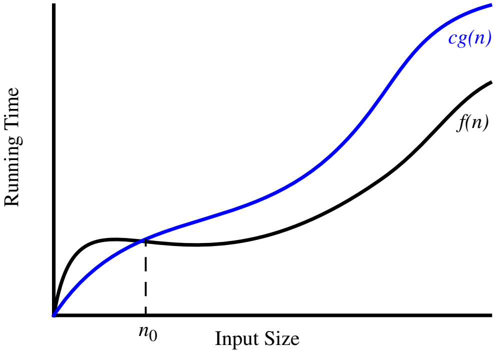
- Let ***f(n)*** and ***g(n)*** be functions mapping pos. integers (input size) to pos. real numbers (running time).
  - ***f(n)*** characterizes the running time of some algorithm.
  - ***O(g(n))***:
    - denotes a collection of functions
    - consists of all functions that can be ***upper bounded by g(n)***, starting at some point, using some constant factor
- ***f(n) ∈ 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: Proposition

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

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

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

- We prove by choosing

$$\begin{array}{rcl} c & = & |a_0| + |a_1| + \cdots + |a_d| \\ n_0 & = & 1 \end{array}$$

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

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

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

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

Asymptotic Upper Bound: Example

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 $n \geq n_0$:

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

Using the proven **proposition**, choose:

- $c = |5| + |-3| + |2| + |-4| + |1| = 15$
- $n_0 = 1$

Asymptotic Upper Bound: Families

- 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*** $g(n) = n^1$:

    $n, 2n, 3n, \ldots$ [functions with degree 1]
    $n^0, 2n^0, 3n^0, \ldots$ [functions with degree 0]
  - e.g., Family of $O(n^2)$ contains all $f(n)$ that can be ***upper bounded by*** $g(n) = n^2$:

    $n^2, 2n^2, 3n^2, \ldots$ [functions with degree 2]
    $n, 2n, 3n, \ldots$ [functions with degree 1]
    $n^0, 2n^0, 3n^0, \ldots$ [functions with degree 0]
- Consequently:

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

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: $O(n^3) \subset O(n^4) \subset O(n^5) \subset \ldots$
  - 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 $O(4n^2 + 6n + 9)$.

Asymptotic Upper Bound: More Examples

LASSONDE SCHOOL OF ENGINEERING

- $5n^2 + 3n \cdot logn + 2n + 5$ is $O(n^2)$ [$c = 15$, $n_0 = 1$]
- $20n^3 + 10n \cdot logn + 5$ is $O(n^3)$ [$c = 35$, $n_0 = 1$]
- $3 \cdot logn + 2$ is $O(logn)$ [$c = 5$, $n_0 =$ 2]
  - Why can't n_0 be 1?
  - Choosing $n_0 = 1$ means $\Rightarrow f(\boxed{1})$ ***is*** upper-bounded by $c \cdot log\boxed{1}$:
    - We have $f(\boxed{1}) = 3 \cdot log1 + 2$, which is 2.
    - We have $c \cdot log\boxed{1}$, which is 0.

  $\Rightarrow f(\boxed{1})$ ***is not*** upper-bounded by $c \cdot log\boxed{1}$ [Contradiction!]
- 2^{n+2} is $O(2^n)$ [$c = 4$, $n_0 = 1$]
- $2n + 100 \cdot logn$ is $O(n)$ [$c = 102$, $n_0 = 1$]

Classes of Functions

upper bound	class	cost
$O(1)$	constant	*cheapest*
$O(log(n))$	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	*most expensive*

Upper Bound of Algorithm: Example (1)

```
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 (2)

```
int sumMaxAndCrossProducts (int[] a, int n) {
  int max = a[0];
  for(int i = 1; i < n; i ++) {
    if (a[i] > max) { max = a[i]; }
  }
  int sum = max;
  for (int j = 0; j < n; j ++) {
    for (int k = 0; k < n; k ++) {
      sum += a[j] * a[k]; } }
  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 (3)

```
int triangularSum (int[] a, int n) {
  int sum = 0;
  for (int i = 0; i < n; i ++) {
    for (int j = i; j < n; j ++) {
      sum += a[j]; } }
  return sum; }
```

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

Array Implementations: Stack and Queue

- When implementing ***stack*** and ***queue*** via ***arrays***, we imposed a maximum capacity:

```
public class ArrayStack<E> implements Stack<E> {
  private final int MAX_CAPACITY = 1000;
  private E[] data;
  ...
  public void push(E e) {
    if (size() == MAX_CAPACITY) { /* Precondition Violated */ }
    else { ... }
  }
  ...
}
```

```
public class ArrayQueue<E> implements Queue<E> {
  private final int MAX_CAPACITY = 1000;
  private E[] data;
  ...
  public void enqueue(E e) {
    if (size() == MAX_CAPACITY) { /* Precondition Violated */ }
    else { ...
  }
  ...
}
```

- This made the ***push*** and ***enqueue*** operations both cost ***O(1)***.

Dynamic Array: Constant Increments

Implement ***stack*** using a ***dynamic array*** resizing itself by a constant increment:

```
1  public class ArrayStack<E> implements Stack<E> {
2    private int I;
3    private int C;
4    private int capacity;
5    private E[] data;
6    public ArrayStack() {
7      I = 1000; /* arbitrary initial size */
8      C = 500; /* arbitrary fixed increment */
9      capacity = I;
10     data = (E[]) new Object[capacity];
11     t = -1;
12   }
13   public void push(E e) {
14     if (size() == capacity) {
15       /* resizing by a fixed constant */
16       E[] temp = (E[]) new Object[capacity + C];
17       for(int i = 0; i < capacity; i ++) {
18         temp[i] = data[i];
19       }
20       data = temp;
21       capacity = capacity + C
22     }
23     t++;
24     data[t] = e;
25   }
26 }
```

- This alternative strategy ***resizes*** the array, whenever needed, by a ***constant*** amount.
- **L17 – L19** make ***push*** cost ***O(n)***, in the ***worst case***.
- However, given that ***resizing*** only happens rarely, how about the *average* running time?
- We will refer **L14 – L22** as the ***resizing*** part and **L23 – L24** as the ***update*** part.

Dynamic Array: Doubling

Implement ***stack*** using a ***dynamic array*** resizing itself by doubling:

```
1  public class ArrayStack<E> implements Stack<E> {
2    private int I;
3    private int capacity;
4    private E[] data;
5    public ArrayStack() {
6      I = 1000; /* arbitrary initial size */
7      capacity = I;
8      data = (E[]) new Object[capacity];
9      t = -1;
10   }
11   public void push(E e) {
12     if (size() == capacity) {
13       /* resizing by doubling */
14       E[] temp = (E[]) new Object[capacity * 2];
15       for(int i = 0; i < capacity; i ++) {
16         temp[i] = data[i];
17       }
18       data = temp;
19       capacity = capacity * 2
20     }
21     t++;
22     data[t] = e;
23   }
24 }
```

- This alternative strategy ***resizes*** the array, whenever needed, by ***doubling*** its current size.
- **L15 – L17** make ***push*** cost ***O(n)***, in the ***worst case***.
- However, given that ***resizing*** only happens rarely, how about the *average* running time?
- We will refer **L12 – L20** as the resizing part and **L21 – L22** as the update part.

Avg. RT: Const. Increment vs. Doubling

- Without loss of generality, assume: There are n ***push*** operations, and the **last push** triggers the **last** ***resizing*** routine.

	Constant Increments	Doubling
RT of exec. update part for n pushes	$O(n)$	
RT of executing 1st resizing	I	
RT of executing 2nd resizing	$I + C$	$2 \cdot I$
RT of executing 3rd resizing	$I + 2 \cdot C$	$4 \cdot I$
RT of executing 4th resizing	$I + 3 \cdot C$	$8 \cdot I$
RT of executing k^{th} resizing	$I + (k - 1) \cdot C$	$2^{k-1} \cdot I$
RT of executing last resizing	n	
# of resizing needed (solve k for $RT = n$)	$O(n)$	$O(log_2 n)$
Total RT for n pushes	$O(n^2)$	$O(n)$
Amortized/Average RT over n pushes	**O(n)**	**O(1)**

- Over n push operations, the *amortized* / *average* running time of the ***doubling*** strategy is more efficient.

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

Experimental Analysis: Challenges

Moving Beyond Experimental Analysis

Counting Primitive Operations

From Absolute RT to Relative RT

Example: Approx. # of Primitive Operations

Index (2)

Approximating Running Time
as a Function of Input Size

Rates of Growth: Comparison

Focusing on the Worst-Case Input

What is Asymptotic Analysis?

Three Notions of Asymptotic Bounds

Asymptotic Upper Bound: Definition

Asymptotic Upper Bound: Visualization

Asymptotic Upper Bound: Proposition

Asymptotic Upper Bound: Example

Asymptotic Upper Bound: Families

Index (3)

Using Asymptotic Upper Bound Accurately

Asymptotic Upper Bound: More Examples

Classes of Functions

Upper Bound of Algorithm: Example (1)

Upper Bound of Algorithm: Example (2)

Upper Bound of Algorithm: Example (3)

Array Implementations: Stack and Queue

Dynamic Array: Constant Increments

Dynamic Array: Doubling

Avg. RT: Const. Increment vs. Doubling