School of Computer Science

COMP 8547 “Advanced Computing Concepts”

Winter 2026

Chapter 1: Algorithm Analysis

DR. OLENA SYROTKINA

University of Windsor

Agenda

- Get to know course team
- Course outline
- Introduction to computing
- Algorithms
- Experimental analysis
- Pseudocode
- Functions in algorithm analysis
- Asymptotic analysis
- Asymptotic notation
- Big-Oh, Big-Omega, Big-Theta, Little-Oh, Little-Omega
- Examples
- Search
- Prefix averages
- Maximum contiguous subsequence sum
- Java and Eclipse

Course Outline

This course covers advanced topics in principles and applications of algorithm design and analysis, programming techniques, advanced data structures, languages, compilers and translators, regular expressions, grammars, computing and intractability. Cases studies and applications in current programming languages are explored in class and labs. This course is restricted to students in the Master of Applied Computing program.

Assessments

- ✓ *Labs & Assignments*
- ✓ *Participation in seminars/workshops*
- ✓ *Quizzes*
- ✓ *Mid-Term Exam*
- ✓ *Final project*

COURSE EVALUATION

Labs	4%	You need to score 4% out of 5% possible (each lab = 0.5%, 10 labs in total). Labs should be completed during the class. Students are required to show a GA their partially completed lab during/after the class and the GA marks it. Grace period: 5 hours. Submitting the lab within this time frame is considered a regular submission. No submissions will be allowed after the deadline.
Assignments	21%	You will have 4 assignments in total. Please always read the submission requirements before turning in your assignment. Any submission after the deadline will receive a penalty of 10% for the first 24 hrs, and so on, for up to three days (10% per day = 30% total). After three days, the mark will be zero. Grace period: 5 hours. Submitting the assignment within this time frame is considered a regular submission with no penalties applied.
Participation in seminars/workshops	5%	To receive your participation marks you are required to attend 10 Seminars/Workshops (CS Workshops, Colloquiums, and Thesis defence or proposal) during the Winter 2026 term. You will be required to register for the event and sign in after the event. The attendance will be tracked by the graduate secretary and will be provided to the instructors at the end of the term to calculate your participation marks.
Mid-Term Exam I	20%	TBA, Ch. 1-4.
Mid-Term Exam II	20%	TBA, Ch. 5-8.
Final project	25%	Group work, the details will be announced on Brightspace
Quizzes	5%	These are either announced or unannounced quizzes regarding lectures to test the topics discussed in the lectures. Those quizzes may be conducted randomly at any time (before or after the lecture/lab) and will contain multiple-choice questions or short answer questions. **There are no make-ups for missed quizzes.**

COURSE POLICY

- **Academic Honesty**:
  - ✓ Refer to UWindsor Policy Page.
  - ✓ If you're not sure, ask the professor.
  - ✓ Plagiarism will not be tolerated.

All announcements will be available on Brightspace.

- **Missed Assessment Make-up:**

No make-ups will be considered for

- ✓ missing a mid-exam;
- ✓ missing labs;
- ✓ missing a random lecture quiz;
- ✓ missing a final project presentation;

- **Late Assignment:**
  - ✓ Any submission after the deadline will receive a penalty of 10% for the first 24 hrs, and so on, for up to three days. After three days, the mark will be zero.

ASSESSMENTS

➢ **Labs:** *to understand the topics discussed in the lectures and to learn more about advanced algorithms and data structures.*

1. Labs should be completed during the class. Students are required to show a GA/TA their completed lab after the class, and the GA/TA will mark it as 'done,' 'not done,' or 'partially done.'

2. If a student misses the class, they will lose points for the lab, **even if they submit it on Brightspace after the lab has concluded.**

3. Each lab must be submitted through Brightspace in **both *.java and *.txt formats (+ screenshots)** and is subject to a plagiarism check.

4. 0.5 points will be awarded for each lab if two conditions are met:
   A) The student attended the lab and showed the result to the GA;
   B) The plagiarism check originality score does not **exceed 50%**.

5. **No points will be awarded for labs submitted via email, Teams, or other platforms, for sending zip archives, or for failing to submit your code in *.txt files or screenshots.**

Student Responsibilities

- Be polite in all dealings with the professor, the GA/TAs, and the other students.
- Come to the class **on time** and ready to participate in the learning process.
- If you miss any announcement, it is your responsibility to catch up on instruction you have missed.
- Make sure that you do not plagiarize in any assessments.
- Make sure to submit all assessments on time.

ASSESSMENTS

➢ Participation in seminars/workshops:

To receive your participation marks you are required to attend a total of 10 Seminars/Workshops (CS Workshops, Colloquiums, and Thesis defence or proposal) during the Winter 2026 term. You will be required to register for the event, sign in and complete the QR code after the event. The attendance will be tracked by the Admin staff and will be provided to the instructors at the end of the term to calculate your participation marks.

These events are regularly announced, and you can find the announcements at the following link, as well:
https://www.uwindsor.ca/science/computerscience/event-calendar/month
The participation is saved by registering to the attendance list (or scanning a QR code) available at each event.

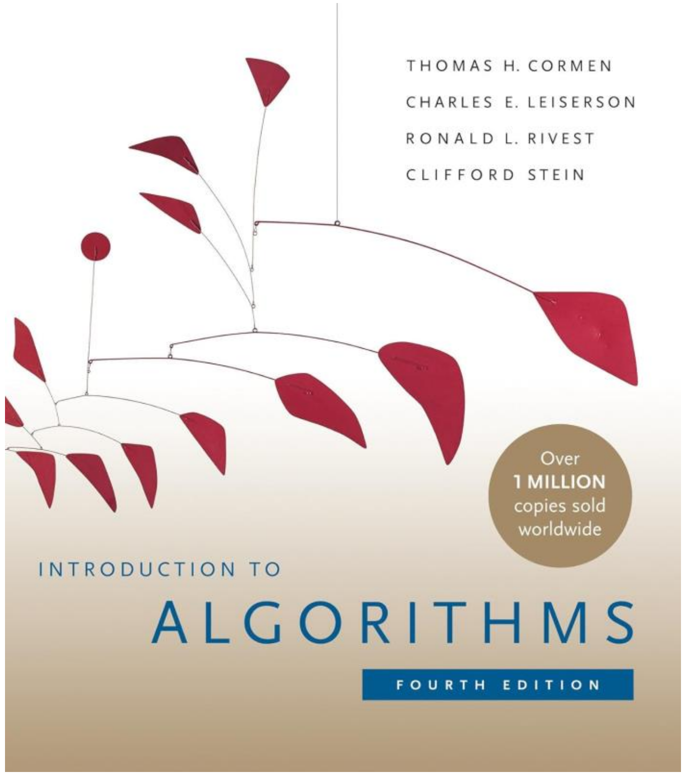
ASSESSMENTS

- **Quizzes:**

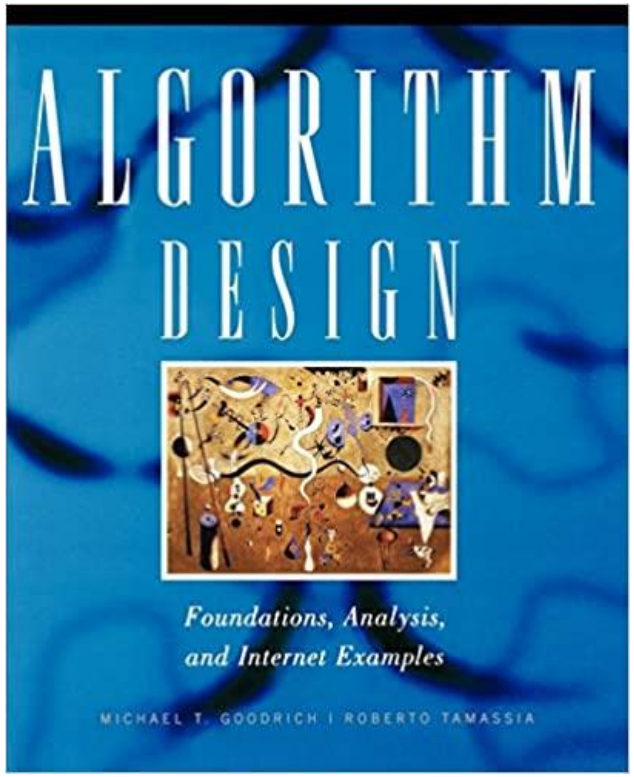
These are quizzes regarding lectures to test the topics discussed in the lectures. Such quizzes may be conducted randomly at any time (before or after the class) and may contain multiple-choice and/or short-answer questions.

There are no make-ups for a missed quiz.

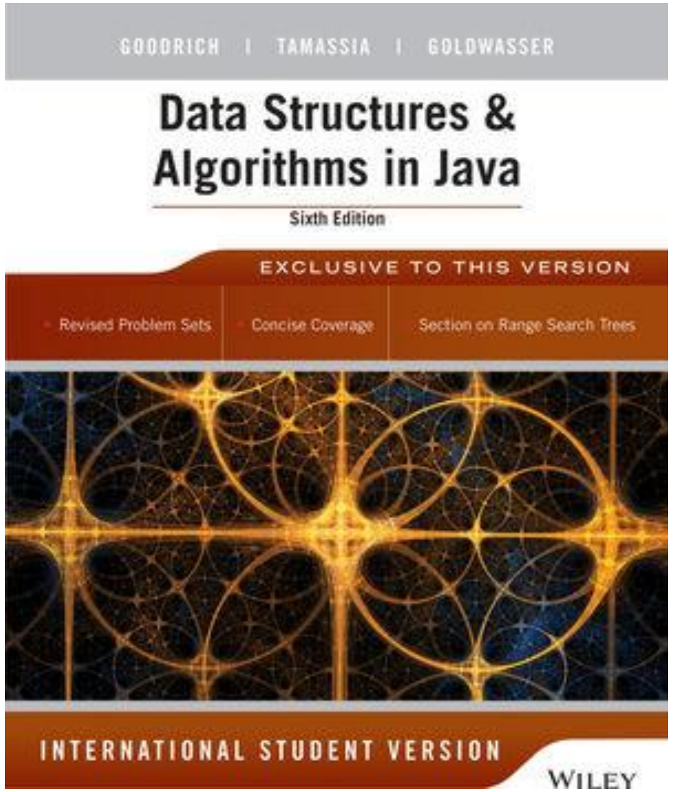
TEXTBOOKS



Introduction to Algorithms, fourth edition by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein, The MIT Press, 2022.

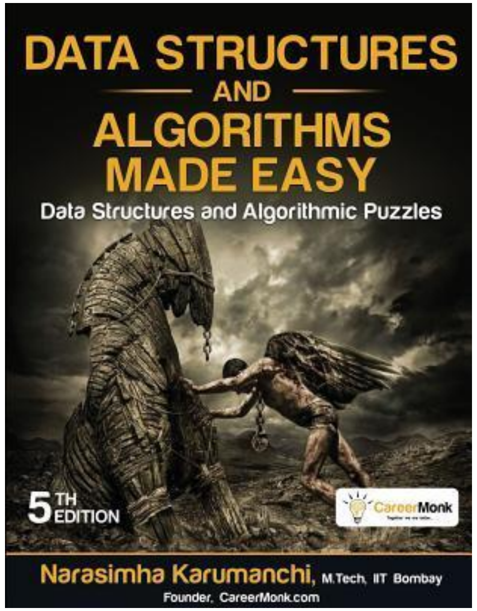

ALGORITHM
DESIGN
Foundations, Analysis, and Internet Examples
MICHAEL T. GOODRICH | ROBERTO TAMASSIA

Algorithm Design and Applications by M. Goodrich and R. Tamassia, Wiley, 2015.

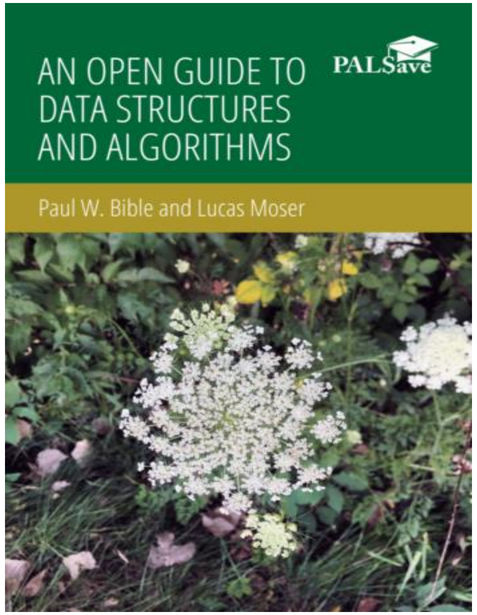

GOODRICH | TAMASSIA | GOLDWASSER
Data Structures & Algorithms in Java
Sixth Edition
EXCLUSIVE TO THIS VERSION
Revised Problem Sets
Concise Coverage
Section on Range Search Trees
INTERNATIONAL STUDENT VERSION
WILEY

Data Structures and Algorithms in Java, 6th Edition, by M. Goodrich and R. Tamassia, Wiley, 2014

TEXTBOOKS (OPTIONAL)



Data Structures and Algorithms Made Easy: Data Structures and Algorithmic Puzzles, 5th Edition, by Narasimha Karumanchi, CareerMonk Publications, 2017.

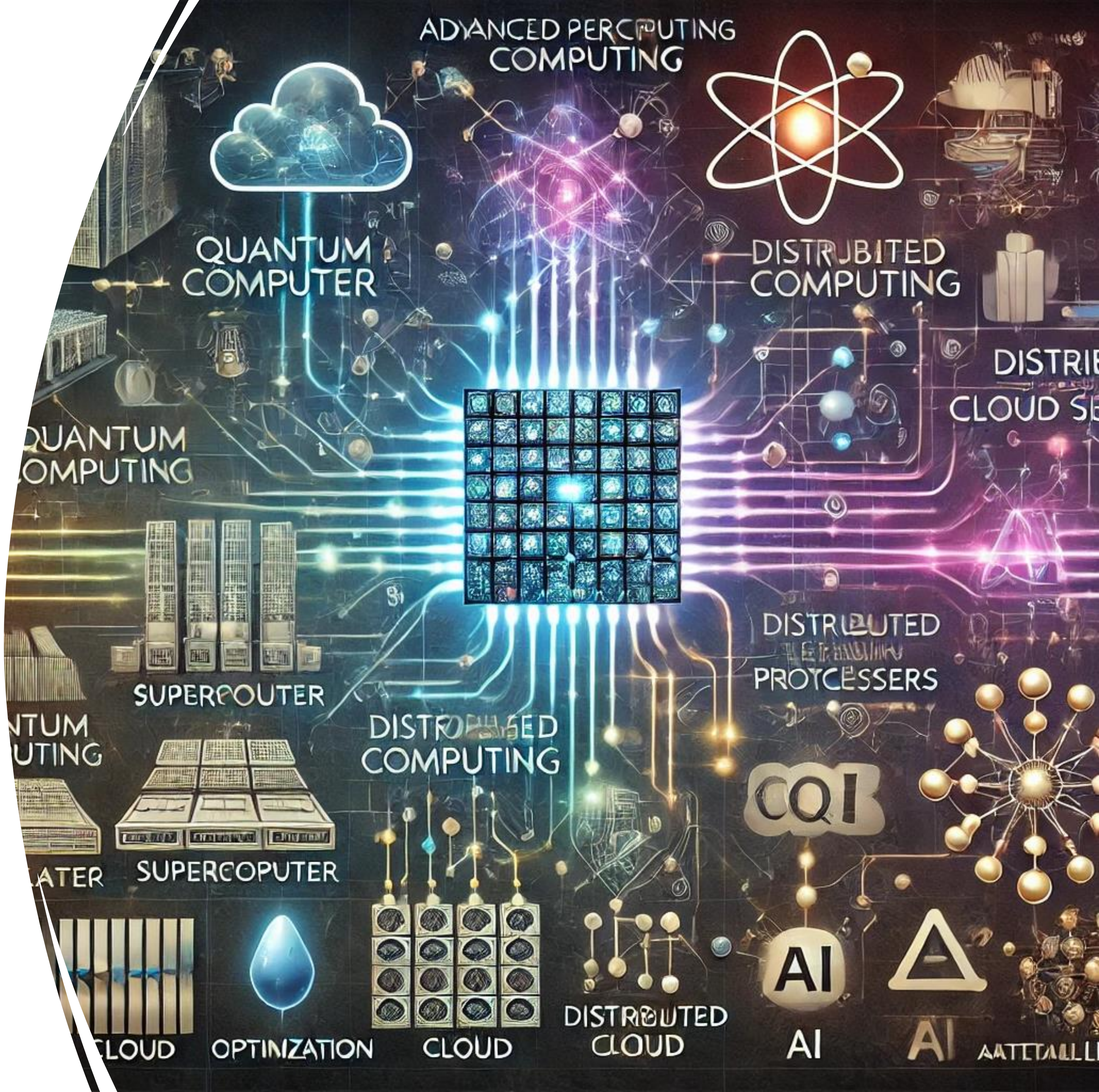


An Open Guide to Data Structures and Algorithms by Paul W. Bible and Lucas Moser, PALNI Press, 2023.

Introduction to Computing

- **Computing** refers to the use of computers and other electronic systems to process, store, retrieve, and manage data. It encompasses a wide range of activities and technologies related to calculations, problem-solving, and information processing. At its core, computing involves the execution of instructions (usually in the form of algorithms) by a computer to perform specific tasks, such as calculations, data analysis, communication, and automation.

- Algorithms are the foundation of any computational process, and advanced computing concepts rely heavily on the efficiency and effectiveness of these algorithms to perform at scale or in novel ways.



What is an Algorithm?

Let us consider the problem of preparing an omelette. To prepare an omelette, we follow the steps given below:

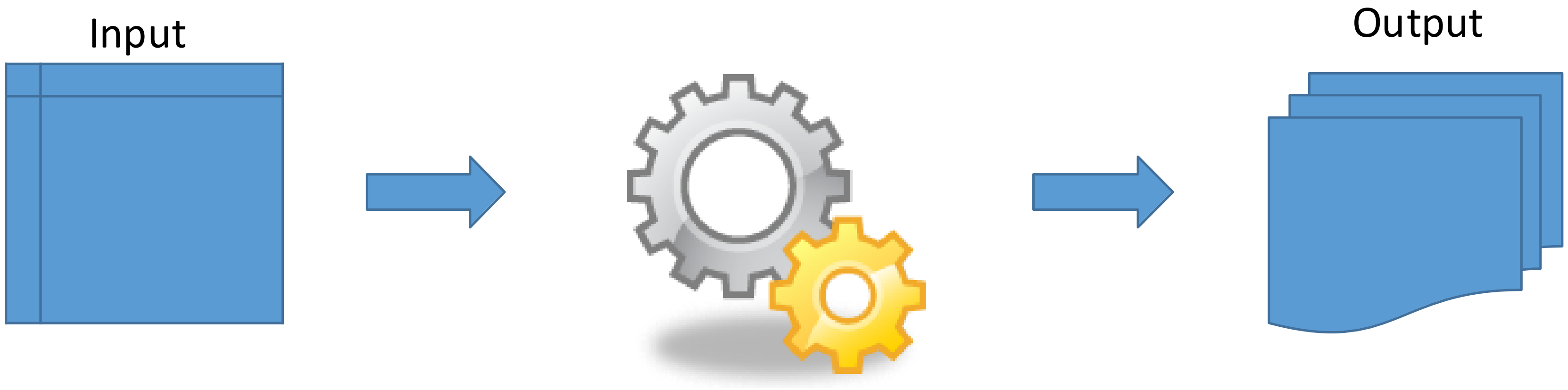
```
1) Get the frying pan.
2) Get the oil.
        a. Do we have oil?
                i.  If yes, put it in the pan.
                ii. If no, do we want to buy oil?
                        1. If yes, then go out and buy.
                        2. If no, we can terminate.

3) Turn on the stove, etc...
```

What we are doing is, for a given problem (preparing an omelette), we are providing a step-by-step procedure for solving it. The formal definition of an algorithm can be stated as:

An algorithm is the step-by-step unambiguous instructions to solve a given problem.

Algorithm

- Definition: An algorithm is a sequence of steps used to solve a problem in a finite amount of time



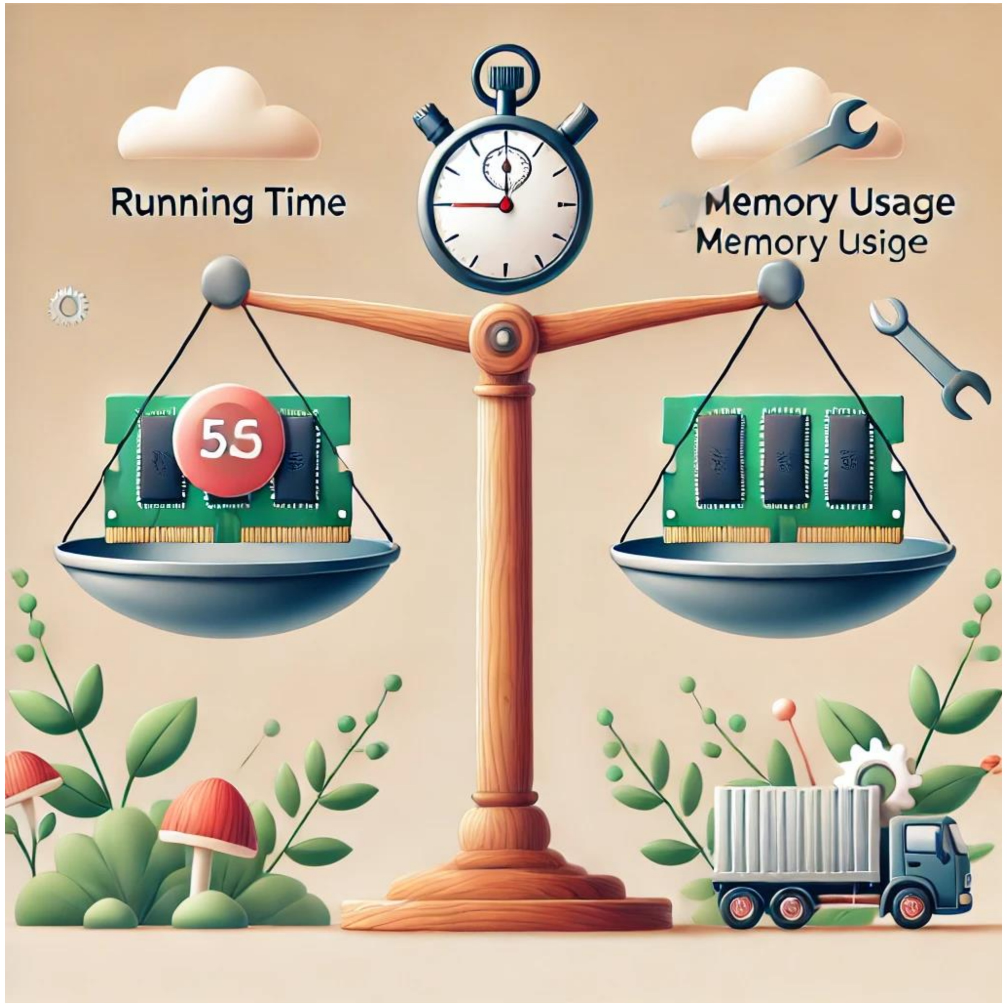
- Properties
  - Correctness: must provide the correct output for *every* input
  - Performance: Measured in terms of the resources used (time and space)
  - End: must finish in a *finite* amount of time

Why the Analysis of Algorithms?

To go from city “A” to city “B”, there can be many ways of accomplishing this: by flight, by bus, by train and also by bicycle. Depending on the availability and convenience, we choose the one that suits us. Similarly, in computer science, multiple algorithms are available for solving the same problem (for example, a sorting problem has many algorithms, like insertion sort, selection sort, quick sort and many more). Algorithm analysis helps us to determine which algorithm is most efficient in terms of time and space consumed.

Goal of the Analysis of Algorithms

The goal of the analysis of algorithms is to compare algorithms (or solutions) mainly in terms of running time but also in terms of other factors (e.g., memory, developer effort, etc.)



What is Running Time Analysis?

It is the process of determining how processing time increases as the size of the problem (input size) increases. Input size is the number of elements in the input, and depending on the problem type, the input may be of different types. The following are the common types of inputs.

- Size of an array
- Polynomial degree
- Number of elements in a matrix
- Number of bits in the binary representation of the input
- Vertices and edges in a graph.

A	**21**	**22**	**23**	**25**	**24**	**23**	**22**
	0	1	2	3	4	5	6

How to Compare Algorithms?

To compare algorithms, let us define *a few objective measures:*

Execution times? *Not a good measure* as execution times are specific to a particular computer.

Number of statements executed? *Not a good measure*, since the number of statements varies with the programming language as well as the style of the individual programmer.

Ideal solution? Let us assume that we express the running time of a given algorithm as a function of the input size *n* (i.e., *f(n)*) and compare these different functions corresponding to running times. This kind of comparison is independent of machine time, programming style, etc.

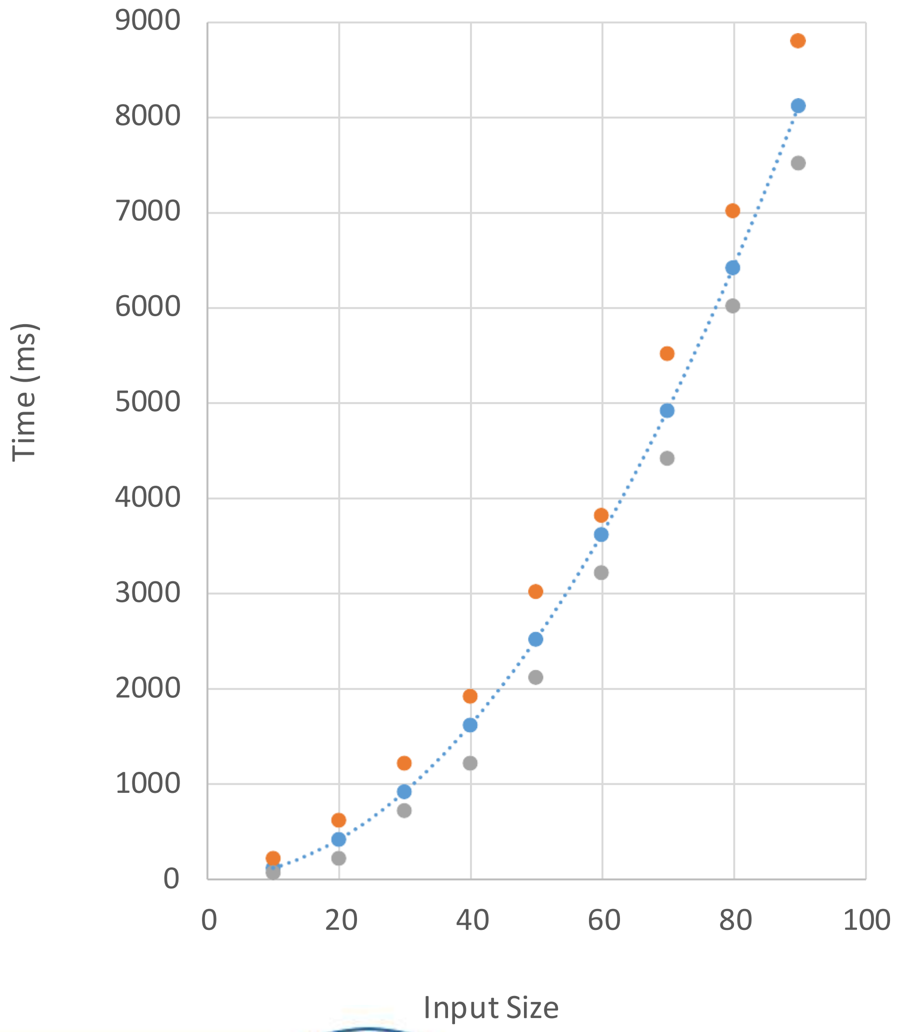
Experimental vs Theoretical analysis

Experimental analysis:

- Write a program that implements the algorithm
- Run the program with inputs of varying size and composition
- Keep track of the CPU time used by the program on each input size
- Plot the results on a two-dimensional plot

Limitations:

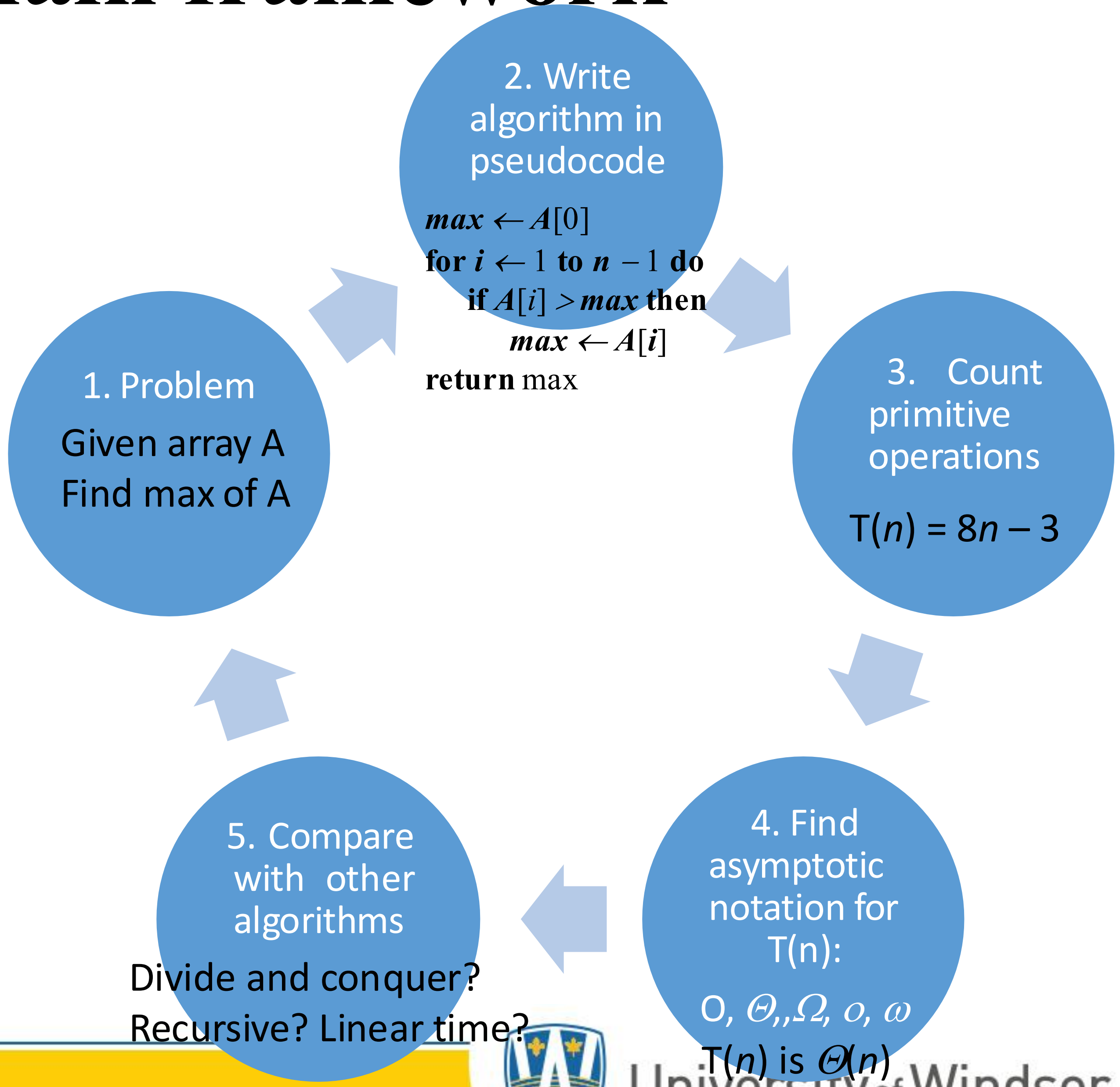
- Depends on hardware and programming language
- Need to implement the algorithm and debug the programs



Theoretical analysis – main framework

Advantages:

- 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 independently of the hardware/software environment



1. Problem

Given array A

Find max of A

2. Write algorithm in pseudocode

```
$max \leftarrow A[0]$
for $i \leftarrow 1$ to $n - 1$ do
    if $A[i] > max$ then
        $max \leftarrow A[i]$
return max
```

3. Count primitive operations

$T(n) = 8n - 3$

4. Find asymptotic notation for T(n):

O, Θ,,Ω, o, ω

T(n) is Θ(n)

5. Compare with other algorithms

Divide and conquer?

Recursive? Linear time?

What is Rate of Growth?

The rate at which the running time increases as a function of input is called *rate of growth*. Let us assume that you go to a shop to buy a car and a bicycle. If your friend sees you there and asks what you are buying, then in general you say *buying a car*. This is because the cost of the car is high compared to the cost of the bicycle (approximating the cost of the bicycle to the cost of the car).

```
Total Cost = cost_of_car + cost_of_bicycle
Total Cost ≈ cost of car (approximation)
```

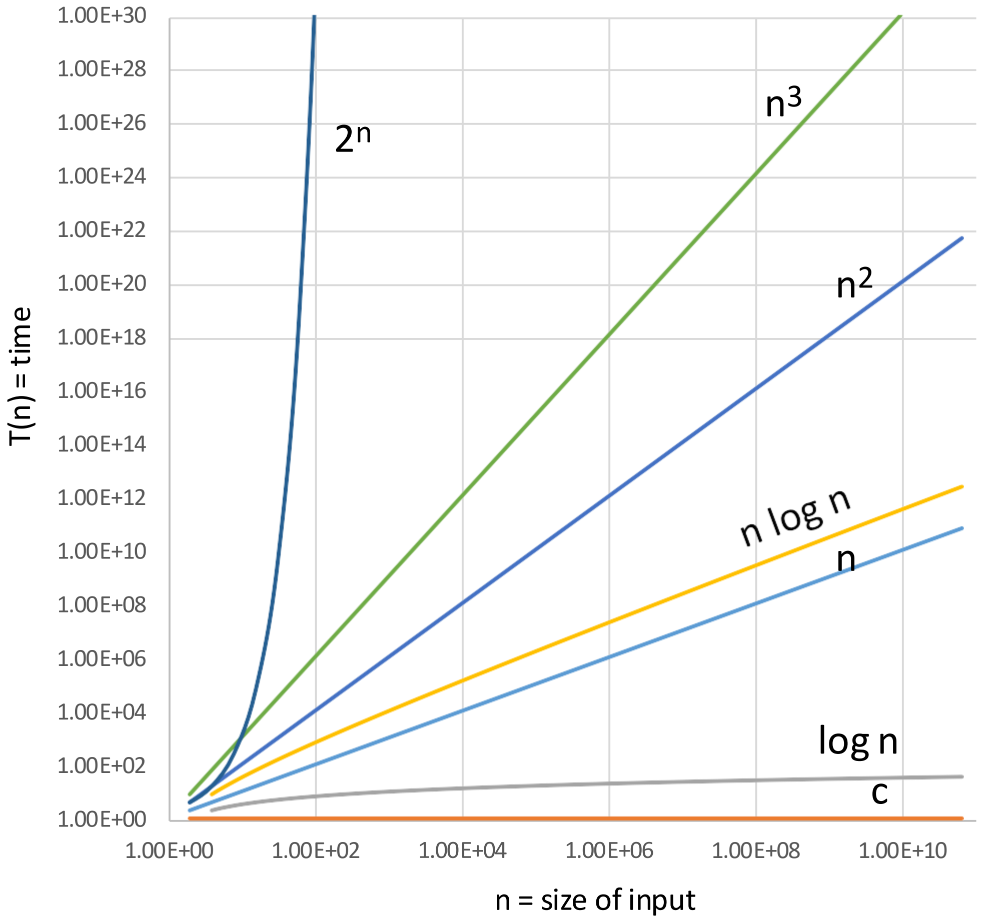
For the above-mentioned example, we can represent the cost of the car and the cost of the bicycle in terms of function, and for a given function ignore the low order terms that are relatively insignificant (for large value of input size, n). As an example, in the case below, n^4, $2n^2$, $100n$ and 500 are the individual costs of some function and approximate to n^4 since n^4 is the highest rate of growth.

$$n^4 + 2n^2 + 100n + 500 \approx n^4$$

Commonly Used Rates of Growth

- The following functions often appear in algorithm analysis:
  - Constant $\approx \mathit{1}$ (or c)
  - 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 chart, the slope of the line corresponds to the growth rate of the function (except exponential)



University of Windsor

Pseudocode

- Control flow
  - **if** … **then** … [**else** …]
  - **while** … **do** …
  - **repeat** … **until** …
  - **for** … **do** …
  - Indentation replaces braces
- Method declaration

```
Algorithm $method$ ($arg$ [, $arg$…])
    Input …
    Output …
    return …
```

- Method call

  var.method (*arg* [, *arg*…])

- Return value

  return *expression*

- Expressions

  $\leftarrow$Assignment
  (like $=$in Java)

  $=$Equality testing
  (like $==$in Java)

  n^2 Superscripts and other mathematical formatting allowed

Example: find max element of an array

```
Algorithm $arrayMax(A, n)$
    Input array $A$ of $n$ integers
    Output maximum element of $A$

    $currentMax \leftarrow A[0]$
    for $i \leftarrow 1$ to $n - 1$ do
        if $A[i] > currentMax$ then
            $currentMax \leftarrow A[i]$
    return $currentMax$
```

Pseudocode provides a high-level description of an algorithm and avoids to show details that are unnecessary for the analysis.

Primitive operations

- Basic computations performed by an algorithm
- Identifiable in pseudocode
- Largely independent of the programming language
- Exact definition not important (we will see why later)
- Take a constant amount of time in the RAM model (one unit of time or constant time)

Examples:

- Evaluating an expression

e.g. $a - 5 + c\sqrt{b}$

- Assigning a value to a variable

e.g. $a \leftarrow 23$

- Indexing into an array

e.g. $A[i]$

- Calling a method

e.g. *v.method()*

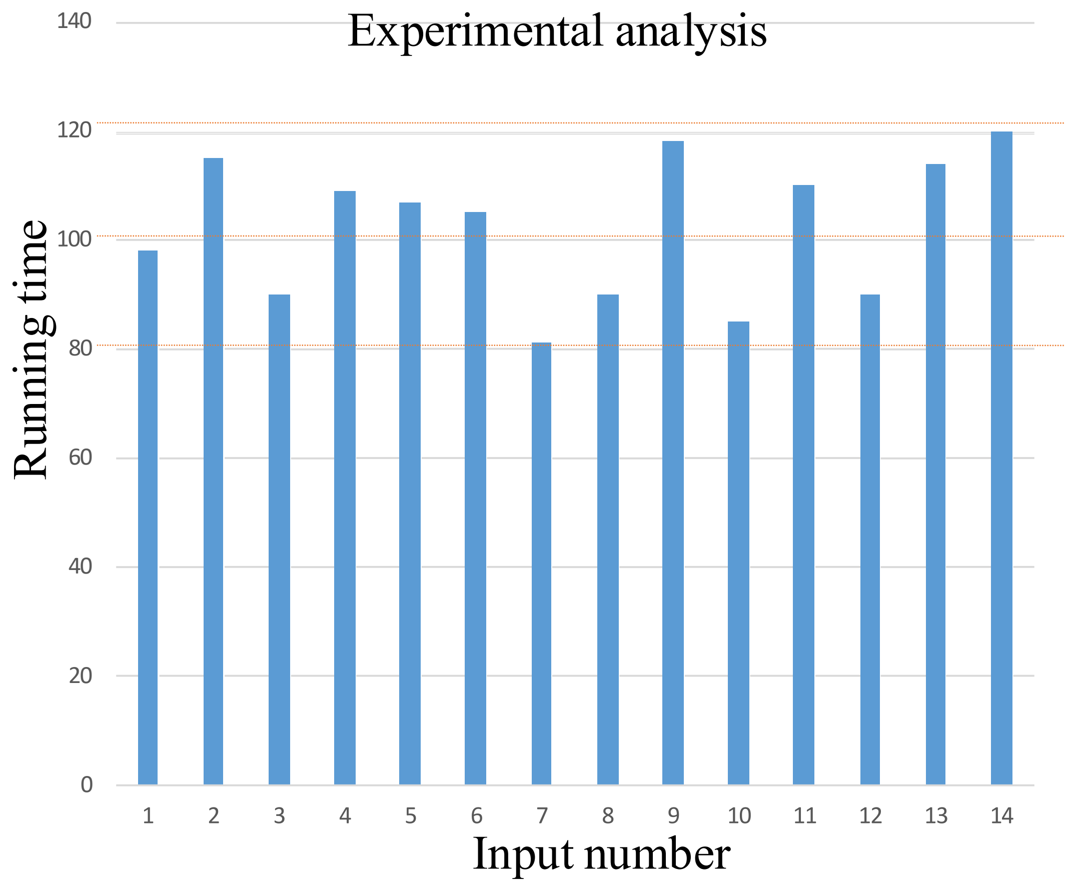
- Returning from a method

e.g. *return a*

Counting primitive operations:

```
Algorithm arrayMax(A, n)                        # operations
    currentMax ← A[0]                                2
    for i ← 1 to n − 1 do                            2n
        if A[i] > currentMax then                 2(n − 1)
            currentMax ← A[i]                     2(n − 1)
        { increment counter i }                   2(n − 1)
    return currentMax                                1
                                               __________

                                    Total: T(n) = 8n − 3
```

Case analysis



Three cases

- Worst case:
  - among all possible inputs, the one which takes the largest amount of time.
- Best case:
  - The input for which the algorithm runs the fastest
- Average case:
  - The average is over all possible inputs
  - Can be considered as the expected value of T(n), which is a random variable

Example: Best vs Worst Case Scenario

Asymptotic notation

Name	Notation /use	Informal name	Bound	Notes
Big-Oh	$O(n)$	order of	Upper bound – tight	The most commonly used notation for assessing the complexity of an algorithm
Big-Theta	$\Theta(n)$		Upper and lower bound – tight	The most accurate asymptotic notation
Big-Omega	$\Omega(n)$		Lower bound – tight	Mostly used for determining lower bounds on problems rather than algorithms (e.g., sorting)
Little-Oh	$o(n)$		Upper bound – loose	Used when it is difficult to obtain a tight upper bound
Little-Omega	$\omega(n)$		Lower bound – loose	Used when it is difficult to obtain a tight lower bound

Asymptotic notation

Big - Oh Notation (O)

- Big - Oh notation is used to define the upper boundary of an algorithm in terms of Time Complexity.
- Worst Case.

Big - Omega Notation (Ω)

- Big - Omega notation is used to define the lower boundary of an algorithm in terms of Time Complexity.
- Best Case

Big - Theta Notation (Θ)

- Big - Theta notation is used to define the average boundary of an algorithm in terms of Time Complexity.
- Average Case

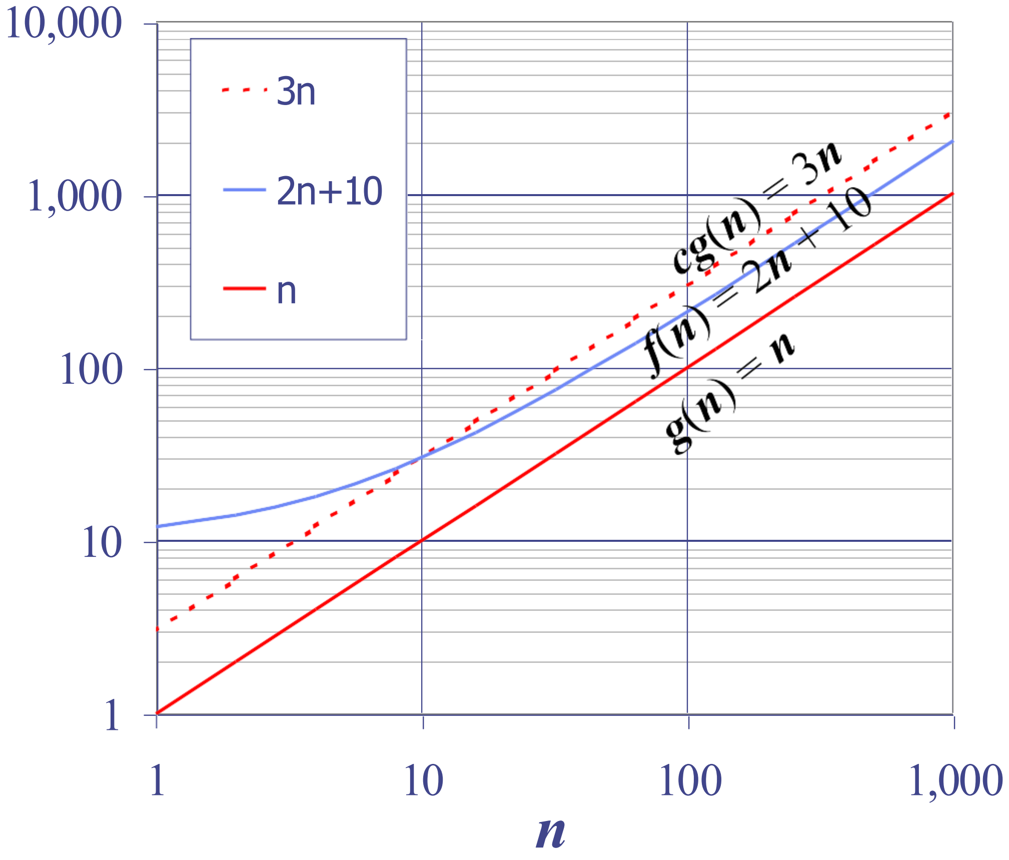
Big-Oh notation

- 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

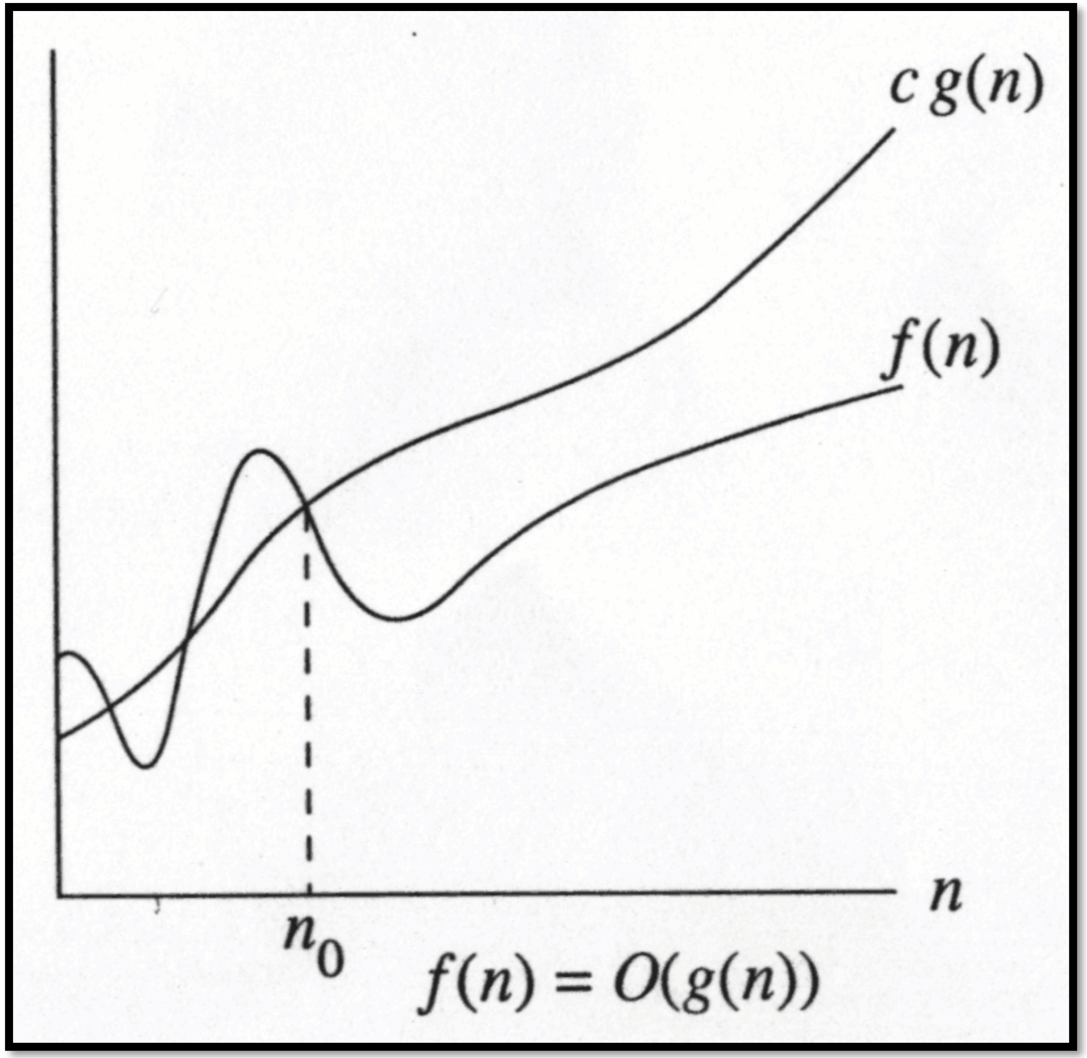
$$f(n) \leq cg(n) \text{ for } n \geq n_0$$

- Example: $2n + 10$ is $O(n)$
  - $2n + 10 \leq cn$
  - $(c - 2)\, n \geq 10$
  - $n \geq 10/(c - 2)$
  - Pick $c = 3$ and $n_0 = 10$
  - It follows that

$$2n + 10 \leq 3n \text{ for } n \geq 10$$



Asymptotic notation: Big-Oh



The idea is to establish a relative order among functions for large

n ∃ c , $n_0 > 0$ such that

$$f(n) \leq c\ g(n)$$

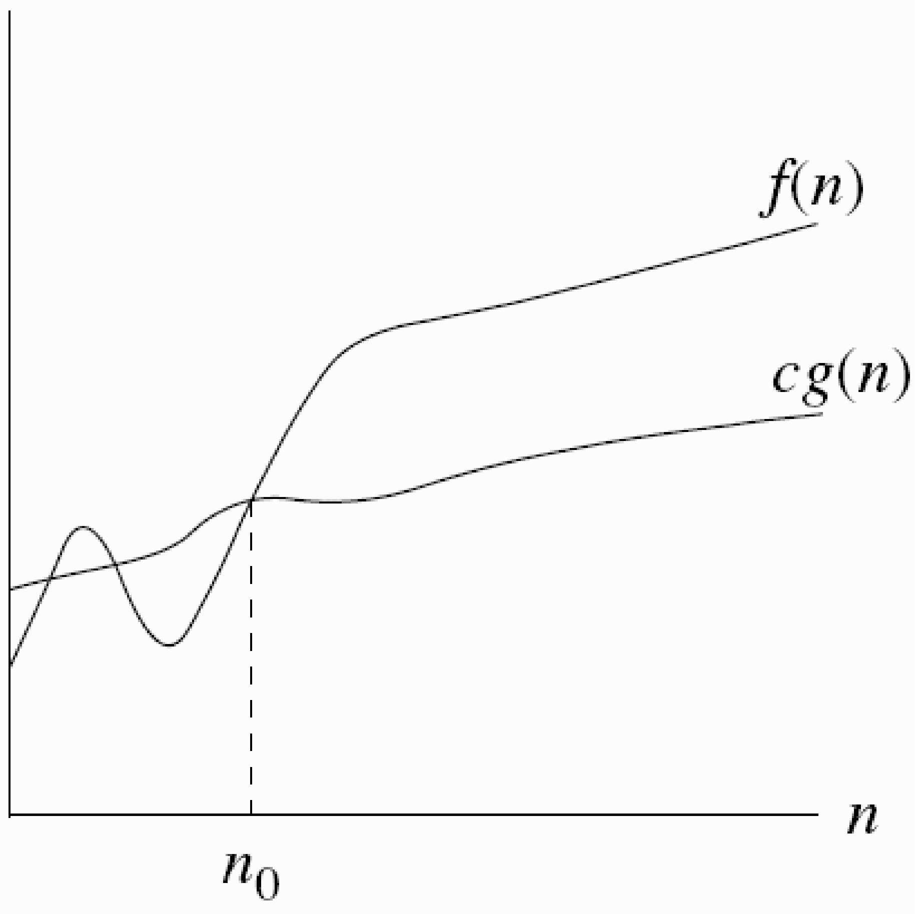
when $n \geq n_0$

f(n) grows no faster than g(n) for "large" n

Big-Oh rules - properties

- The statement "$f(n)$ is $O(g(n))$" means that the growth rate of $f(n)$ is no more than the growth rate of $g(n)$
- $O(g(n))$ is a *set* or *class* of functions: it contains all the functions that have the *same growth rate*
- If $f(n)$ is a polynomial of degree d, then $f(n)$ is $O(n^d)$
  - If $d = 0$, then $f(n)$ is $O(1)$
  - Example: $n^2 + 3n - 1$ is $O(n^2)$
- We always use the *simplest* expression of the class/set
  - E.g., we state 2n + 3 is O(n) instead of O(4n) or O(3n+1)
- We always use the *smallest* possible class/set of functions
  - E.g., we state 2n is O(n) instead of $O(n^2)$ or $O(n^3)$
- Linearity of asymptotic notation
  - O(f(n)) + O(g(n)) = O(f(n) + g(n)) = O(max{f(n),g(n)}), where "max" is wrt "growth rate"
  - Example: $O(n) + O(n^2) = O(n + n^2) = O(n^2)$

Asymptotic notation: Big-Ω (Big-Omega)



f(n) = O(g(n))

If there are positive constants c and n_0

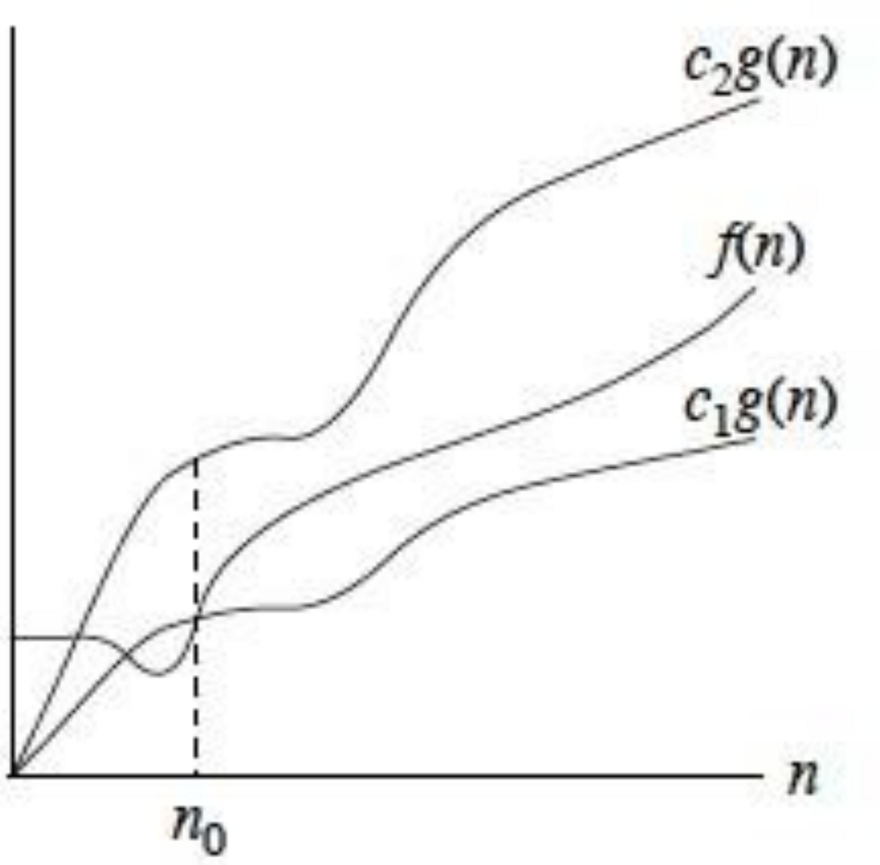
Such that

f(n) > c x g(n)

for all n >= n_0

g(n) is an asymptotic Lower bound for f(n)

Asymptotic notation: Big-θ (Big-Theta)



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

If there are positive constants c1,c2 and n_0

Such that

c1g(n) < f(n) < c2 x g(n)

for all n >= n_0

g(n) is an asymptotic Tight boundary for f(n)

Big-Omega and Big-Theta notations

- **big-Omega**
  - f(n) is Ω(g(n)) if there is a constant c > 0 and an integer constant $n_0 \geq 1$ such that f(n) $\geq$ c g(n) for n $\geq n_0$

  Example: $3n^3 - 2n + 1$ is $\Omega(n^3)$

- **big-Theta**
  - f(n) is Θ(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) for n $\geq n_0$
  - Example: 5n log n – 2n is Θ(n log n)

- Important axiom:
  - f(n) is O(g(n)) and Ω(g(n)) $\Leftrightarrow$ f(n) is Θ(g(n)
  - Example: $5n^2$ is $O(n^2)$ and $\Omega(n^2)$ $\Leftrightarrow$ $5n^2$ is $\Theta(n^2)$

Asymptotic notation – graphical comparison

Big-Oh

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

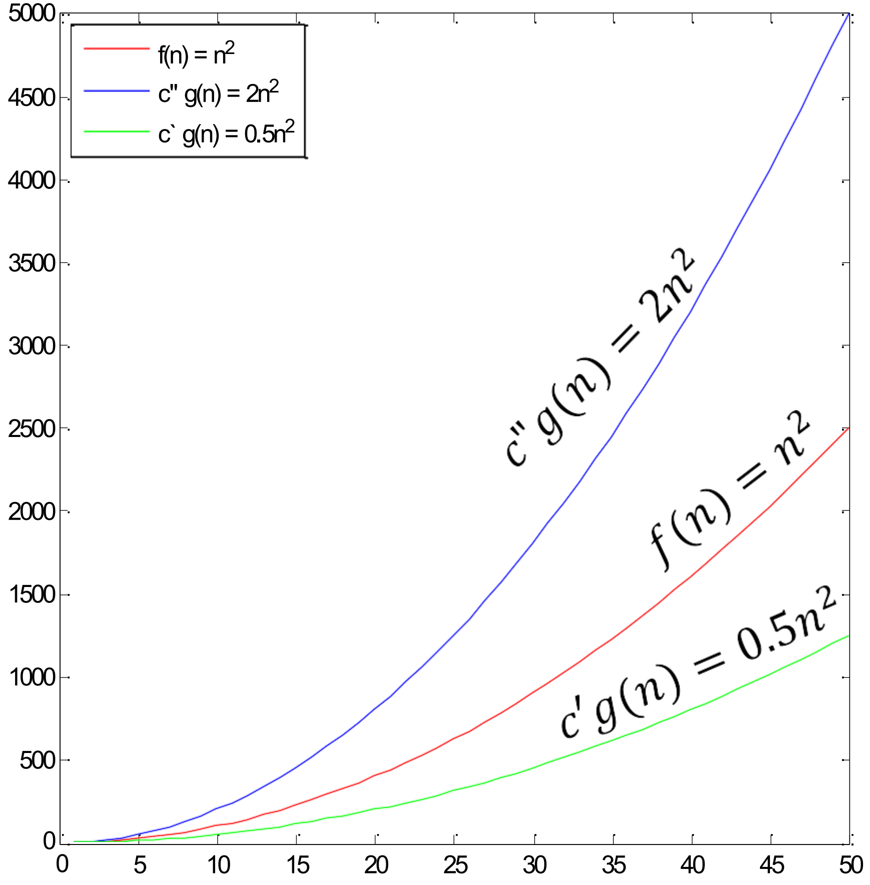
Big-Omega

- f(n) is Ω(g(n)) if f(n) is asymptotically **greater than or equal** to g(n)

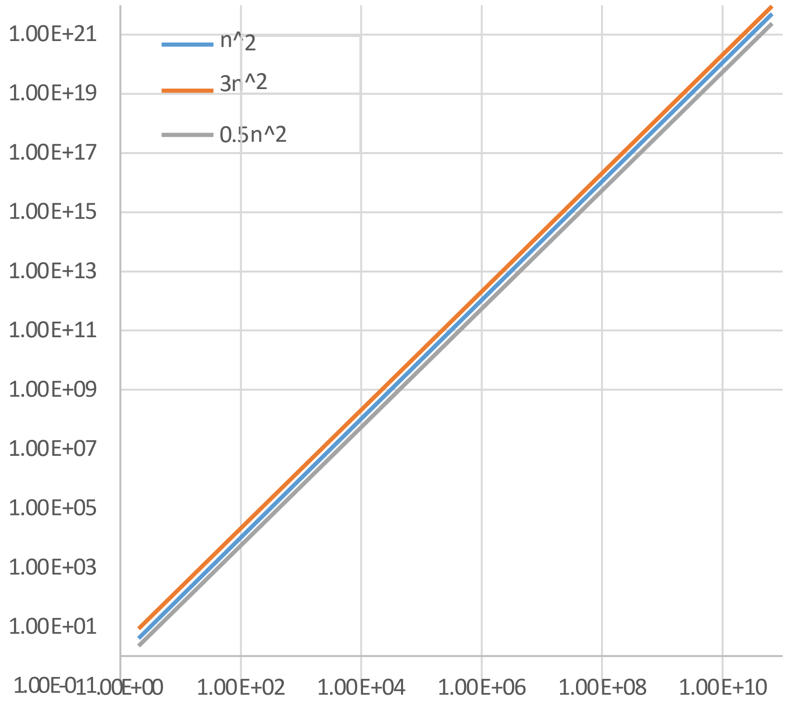
Big-Theta

- f(n) is Θ(g(n)) if f(n) is asymptotically **equal** to g(n)

Normal scale:



Log-log scale:



Little-Oh and Little-Omega notations

- **Little-Oh**
  - f(n) is o(g(n)) if *for any* constant $c > 0$, there is a constant $n_0 > 0$ such that $f(n) < c\, g(n)$ for $n \geq n_0$

  Example: $3n^2 - 2n + 1$ is $o(n^3)$, while $3n^2 - 2n + 1$ **is not** $o(n^2)$

- **Little-Omega**
  - f(n) is ω(g(n)) if for any constant $c > 0$, there is a constant $n_0 > 0$ such that $f(n) > c\, g(n)$ for $n \geq n_0$
  - Example: $3n^2 - 2n + 1$ is $\omega(n)$, while $3n^2 - 2n + 1$ **is not** $\omega(n^2)$

Important axiom:

$$f(n) \text{ is } o(g(n)) \Leftrightarrow g(n) \text{ is } \omega(f(n))$$

- Comparison with O and Ω
  - For O and Ω, the inequality holds if **there exists** a constant $c > 0$
  - For o and ω , the inequality holds **for all** constants $c > 0$

Guidelines for Asymptotic Analysis

There are some general rules to help us determine the running time of an algorithm.

1) **Loops:** The running time of a loop is, at most, the running time of the statements inside the loop (including tests) multiplied by the number of iterations.

```
// executes n times
for (i=1; i<=n; i++)
     m = m + 2; // constant time, c
```

Total time = a constant c **×** ***n = c n*** **= O(*n*).**

Guidelines for Asymptotic Analysis

2) **Nested loops:** Analyze from the inside out. Total running time is the product of the sizes of all the loops.

```
//outer loop executed n times
for (i=1; i<=n; i++) {
    // inner loop executes n times
    for (j=1; j<=n; j++)
        k = k+1; //constant time
}
```

Total time $= c \times n \times n = cn^2 = \mathrm{O}(n^2)$.

Guidelines for Asymptotic Analysis

3) **Consecutive statements:** Add the time complexities of each statement.

```
x = x + 1; //constant time
// executes n times
for (i=1; i<=n; i++)
  m = m + 2; //constant time
// outer loop executes n times
for (i=1; i<=n; i++) {
  // inner loop executed n times
  for (j=1; j<=n; j++)
    k = k+1; //constant time
}
```

Total time $= c_0 + c_1 n + c_2 n^2 = \mathrm{O}(n^2)$.

Guidelines for Asymptotic Analysis

4) **If-then-else statements:** Worst-case running time: the test, plus either the then part or the else part (whichever is the larger).

```
//test: constant
if (length() == 0) {
    return false; // then part: constant
}
else { // else part: (constant + constant) * n
    for (int n = 0; n < length(); n++) {
        // another if: constant + constant (no else part)
        if (!list[n].equals(otherList.list[n]))
            // constant
            return false;
    }
}
```

Total time $= c_0 + c_1 + (c_2 + c_3) * n = O(n)$.

Guidelines for Asymptotic Analysis

5) **Logarithmic complexity:** An algorithm is O(*logn*) if it takes a constant time to cut the problem size by a fraction (usually by ½). As an example, let us consider the following program:

```
for (i=1; i<=n;)
    i = i*2;
```

If we observe carefully, the value of i is doubling every time. Initially $i = 1$, in next step $i = 2$, and in subsequent steps $i = 4,8$ and so on. Let us assume that the loop is executing some k times. At k^{th} step $2^k = n$, and at $(k + 1)^{th}$ step we come out of the *loop*. Taking logarithm on both sides, gives

```
log(2^k) = logn
klog2 = logn
k = logn // if we assume base-2
```

Total time = O(logn).

Guidelines for Asymptotic Analysis

5) **Logarithmic complexity**

Similarly, for the case below, the worst case rate of growth is O(*logn*). The same discussion holds good for the decreasing sequence as well.

```
for (i=n; i>=1;)
    i = i/2;
```

Another example: binary search (finding a word in a dictionary of *n* pages)

- Look at the center point in the dictionary
- Is the word towards the left or right of center?
- Repeat the process with the left or right part of the dictionary until the word is found.

Example 1

What is the complexity of the program given below:

```
void function(int n) {
    int i, j, k , count =0;
    for(i=n/2; i<=n; i++)
        for(j=1; j + n/2<=n; j= j+1)
            for(k=1; k<=n; k= k * 2)
                count++;
}
```

Example 1 – Solution

Consider the comments in the following function.

```
void function(int n) {
    int i, j, k, count = 0;
    // outer loop execute n/2 times
    for(i=n/2; i<=n; i++) {
        // middle loop executes n/2 times
        for(j=1; j + n/2<=n; j= j+1) {
            // inner loop execute logn times
            for(k=1; k<=n; k= k * 2)
                count++;
        }
    }
}
```

The complexity of the above function is O(n^2logn).

Example 2

What is the complexity of the program given below:

```
void function(int n) {
    int i, j, k, count = 0;
    for(i=n/2; i<=n; i++)
        for(j=1; j<=n; j= 2 * j)
            for(k=1; k<=n; k= k * 2)
                count++;
}
```

Example 2 – Solution

Consider the comments in the following function.

```
void function(int n) {
    int i, j, k, count = 0;
    // outer loop execute n/2 times
    for(i=n/2; i<=n; i++) {
        // middle loop executes logn times
        for(j=1; j<=n; j= 2 * j) {
            // inner loop execute logn times
            for(k=1; k<=n; k= k*2)
                count++;
        }
    }
}
```

***The complexity of the above function is** $\mathbf{O(\mathit{nlog}^2\mathit{n})}$.*

Example 3

What is the complexity of the program given below:

```
function( int n ) {
    if(n == 1) return;
    for(int i = 1; i <= n; i++) {
        for(int j = 1; j <= n; j++) {
            printf("*");
            break;
        }
    }
}
```

Example 3 – Solution

Consider the comments in the following function.

```
function( int n ) {
    // constant time
    if( n == 1 ) return;
    // outer loop execute n times
    for(int i = 1; i <= n; i++) {
        // inner loop executes only one time due to break statement.
        for(int j = 1; j <= n; j++) {
            printf("*");
            break;
        }
    }
}
```

The complexity of the above function is **O(*n*)**. *Even though the inner loop is bounded by n, due to the break statement it is executing only once.*

Case study 1: Search in a Map (sorted list)

- Problem: Given a sorted array S of integers (a map), find a key k in that map.
- One of the most important problems in computer science
- Solution 1: Linear search
  - Scan the elements in the list one by one
  - Until the key k is found
- Example:

S[*i*]	8	12	19	22	23	34	41	48
i	0	1	2	3	4	5	6	7

- Linear search runs in *linear* time.

```
Algorithm linearSearch(S, $k$, $n$):
Input: Sorted array S of size $n$, and key $k$
Output:  Null or the element found
$i \leftarrow 0$
while $i < n$ and $S[i] != k$
    $i \leftarrow i + 1$
if $i = n$ then
    return null
else
    $e \leftarrow S[i]$
    return $e$
```

Worst-case running time: $T(n) = 3n + 4$ ➔ $T(n)$ is $O(n)$

Case study 1: Search in a Map (sorted list)

- Problem: Given a sorted array of integers (a map), find a key k in that map.
- Solution 2: Binary search
- Binary search runs in *logarithmic* time
- Same problem:
  - Two algorithms run in different times

```
Algorithm binarySearch(S, k, low, high):
Input: A key k
Output: Null or the element found
if low > high then
        return null
else
  mid ← ⌊(low + high) / 2⌋
  e ← S[mid]
  if k = e.getKey() then
        return e
  else if k < e.getKey() then
           return binarySearch(S, k, low, mid-1)
     else
           return binarySearch(S, k, mid+1, high)
```

S[*i*]	8	12	19	22	23	34	41	48
i	0	1	2	3	4	5	6	7

Worst-case running time: $T(n) = T(n/2) + 1$ ➔ $T(n)$ is $O(\log n)$

Case study 2: Prefix averages

- The ***i***-th prefix average of an array ***S*** is the average of the first $(\boldsymbol{i}+1)$ elements of ***S***:

$$A[\boldsymbol{i}] = (\boldsymbol{S}[0] + \boldsymbol{S}[1] + \ldots + \boldsymbol{S}[\boldsymbol{i}])/(\boldsymbol{i}+1)$$

- Problem: Compute the array ***A*** of prefix averages of another array ***S***
- Has applications in financial analysis
- Solution 1: A quadratic-time algorithm: quadPrefixAve
- Example:

S	**21**	**23**	**25**	**31**	**20**	**18**	**16**
	0	1	2	3	4	5	6
A	**21**	**22**	**23**	**25**	**24**	**23**	**22**
	0	1	2	3	4	5	6

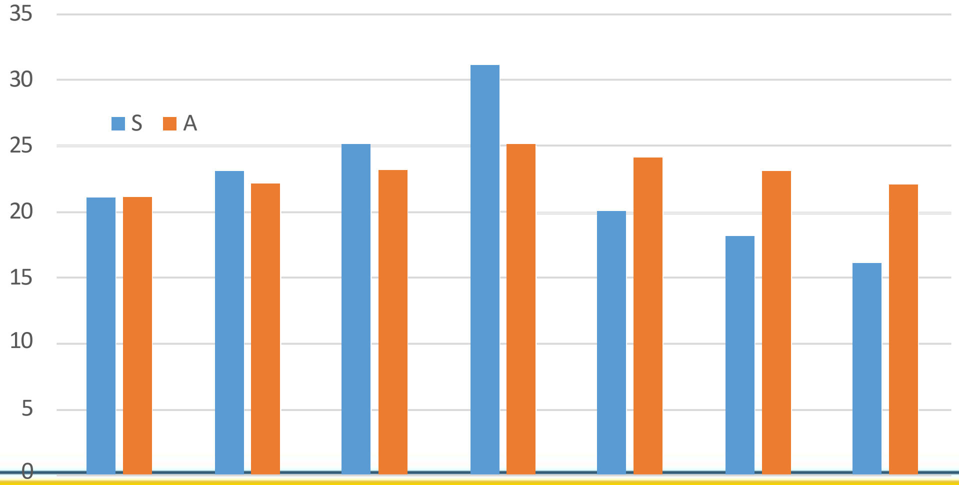
```
Algorithm quadPrefixAve(S, n)
    Input: array S of n integers
    Output: array A of prefix averages of S
                                                        #operations
    A ← new array of n integers                              n
    for i ← 0 to n - 1 do                                    n
        s ← S[0]                                             n – 1
        for j ← 1 to i do                          1 + 2 + …+ (n - 1) ⎫
            s ← s + S[j]                           1 + 2 + …+ (n - 1) ⎭  $\frac{n(n-1)}{2}$
        A[i] ← s / (i + 1)                                   n - 1
    return A                                                 1
```

$T_2(n) = 2n + 2(n\text{-}1) + 2n(n\text{-}1)/2 + 1$ is $O(n^2)$

Case study 2: Prefix averages

- Solution 2: A linear-time algorithm: linearPrefixAve
- For each element being scanned, keep the running sum

S	**21**	**23**	**25**	**31**	**20**	**18**	**16**
	0	1	2	3	4	5	6
A	**21**	**22**	**23**	**25**	**24**	**23**	**22**
	0	1	2	3	4	5	6



```
Algorithm linearPrefixAve($S$, $n$)
    Input: array $S$ of $n$ integers
    Output: array $A$ of prefix averages of $S$
                                                 #operations
    $A \leftarrow$ new array of $n$ integers                 n
    $s \leftarrow 0$                                          1
    for $i \leftarrow 0$ to $n - 1$ do                            n
        $s \leftarrow s + S[i]$                                n – 1
        $A[i] \leftarrow s / (i + 1)$                            n – 1
    return $A$                                         1
```

$T_2(n) = 4n$ is $O(n)$

Case study 3: Maximum contiguous subsequence sum (MCSS)

- Problem:
  - Given: a sequence of integers (possibly negative) $A = A_1, A_2, \ldots, A_n$
  - Find: the maximum value of $\sigma_{k=}^{j} \; A_k$
  - If all integers are negative the MCSS is 0
- Example:
  - For A = -3, 10, -2, 11, -5, -2, 3 the MCSS is 19
  - For A = -7, -10, -1, -3 the MCSS is 0
  - For A = 12, -5, -6, -4, 3 the MCSS is 12
- Various algorithms solve the *same* problem
  - Cubic time
  - Quadratic time
  - Divide and conquer
  - Linear time

MCSS: Cubic vs quadratic time algorithms

Algorithm cubicMCSS(*A*,*n*)

Input: A sequence of integers *A* of length *n*

Output: The value of the MCSS

```
$maxS \leftarrow 0$
for $i \leftarrow 0$ to $n - 1$ do
    for $j \leftarrow i$ to $n - 1$ do
        $curS \leftarrow 0$
        for $k \leftarrow i$ to $j$ do
            $curS \leftarrow curS + A[k]$
        if $curS > maxS$
            $maxS \leftarrow curS$
return $maxS$
```

$$\sum_{i=0}^{n-1} \sum_{j=i}^{n-1} \sum_{k=i}^{j} 1 = \frac{n^3 + 3n^2 + 2n}{6}$$

$T(n) = \frac{n^3+3n^2+2n}{6} + c$ is $O(n^3)$

Algorithm quadraticMCSS(*A*,*n*)

Input: A sequence of integers *A* of length *n*

Output: The value of the MCSS

```
$maxS \leftarrow 0$
for $i \leftarrow 0$ to $n - 1$ do
    for $j \leftarrow i$ to $n - 1$ do
        $curS \leftarrow curS + A[j]$
        if $curS > maxS$
            $maxS \leftarrow curS$
return $maxS$
```

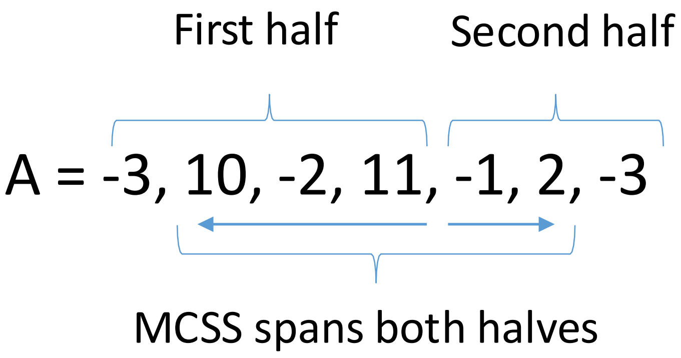
$$\propto \frac{n(n-1)}{2}$$

The double sum will give $O(n^2)$

Example: for A = -3, 10, -2, 11, -5, -2, 3 the MCSS is 19

MCSS: Divide and conquer

- Main features:
  - Rather lengthy
  - Split the sequence into two



First half

Second half

A = -3, 10, -2, 11, -1, 2, -3

MCSS spans both halves

- Algorithm:
  - Divide subsequence into two halves
  - Find max left border sum (left arrow)
  - Find max right border sum (right arrow)
  - Return the sum of both maximums as the max sum
  - Do this recursively for each half
  - Complexity given by T(n) = 2T(n/2) + n, where T(1) = 1
  - Runs in O(n log n)

Linear time algorithm

- Tricky parts of this algorithm are:
  - No MCSS will **start** or **end** with a negative number
  - We only find the value of the MCSS
  - But if we need the actual subsequence, we'll need to resort on at least divide and conquer

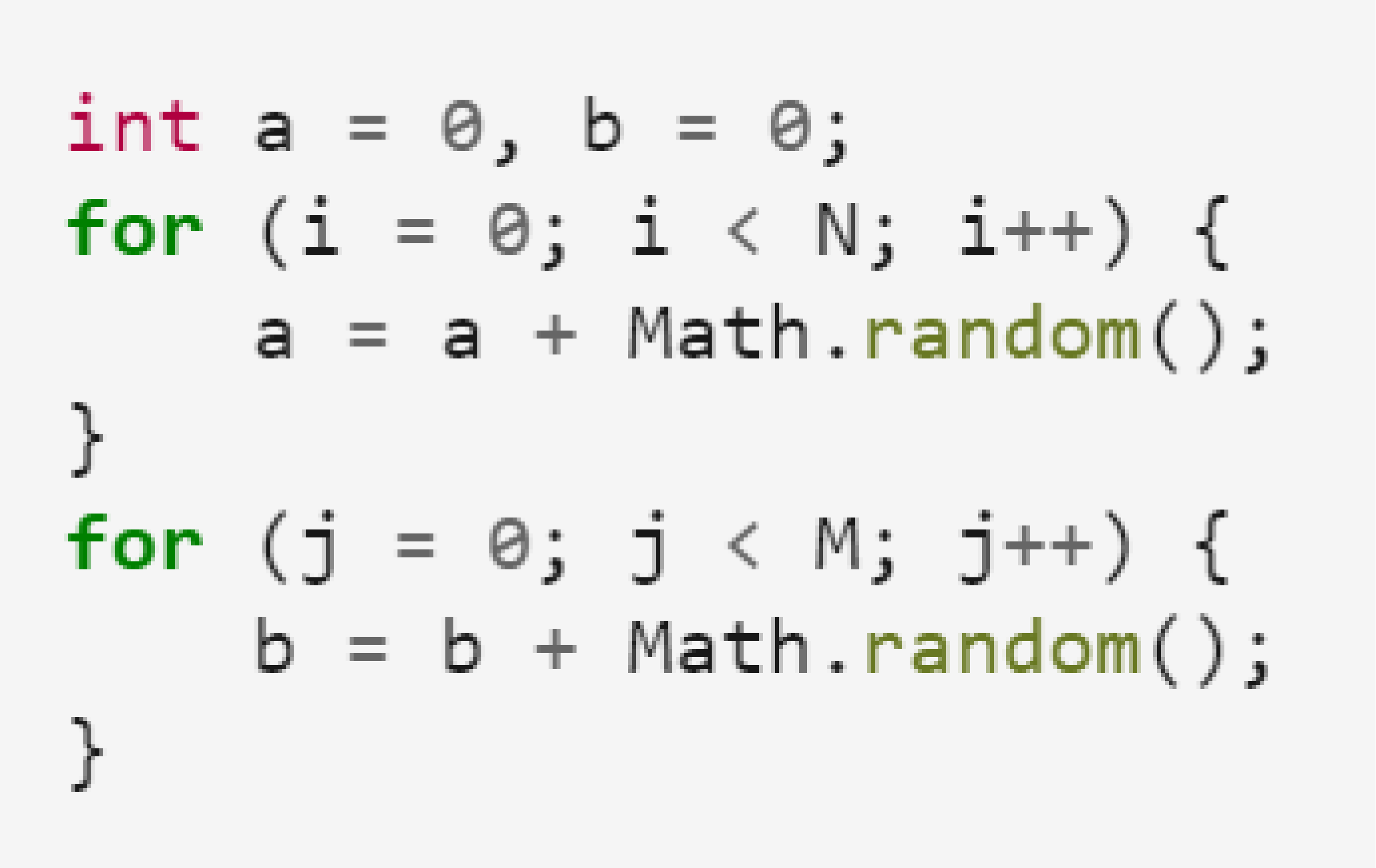
```
Algorithm linearMCSS(A,n)
Input: A sequence of integers A of length n
Output: The value of the MCSS
$maxS \leftarrow 0$; $curS \leftarrow 0$
for $j \leftarrow 0$ to $n - 1$ do
    $curS \leftarrow curS + A[j]$
    if $curS > maxS$
        $maxS \leftarrow curS$
    else
        if $curS < 0$
            $curS \leftarrow 0$
return $maxS$
```

Example: for A = -3, 10, -2, 11, -5, -2, 3 the MCSS is 19

The single for loop gives $O(n)$

Sample Question

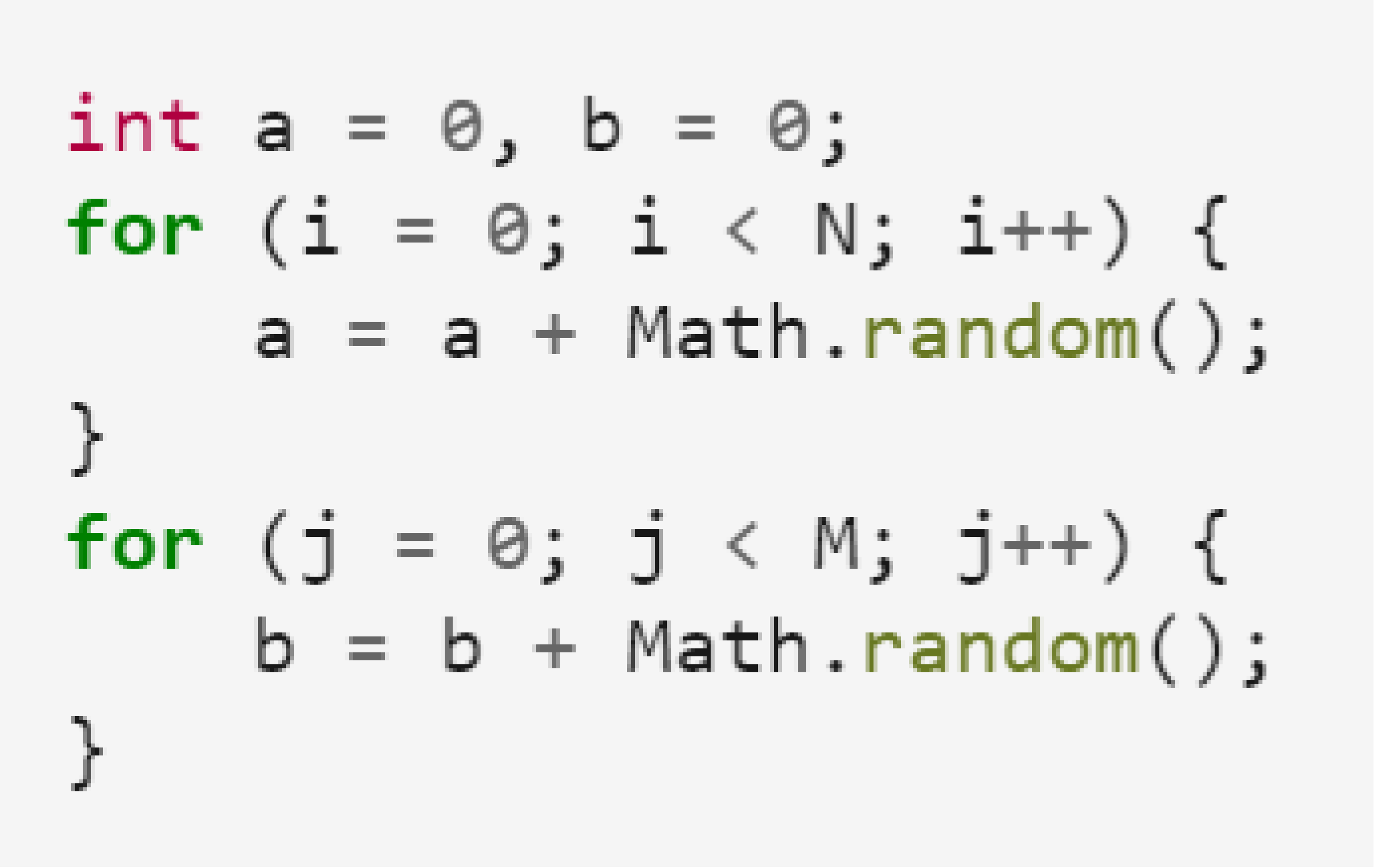
- What is the time complexity of the following code:



```
int a = 0, b = 0;
for (i = 0; i < N; i++) {
    a = a + Math.random();
}
for (j = 0; j < M; j++) {
    b = b + Math.random();
}
```

Sample Question

- What is the time complexity of the following code?



```
int a = 0, b = 0;
for (i = 0; i < N; i++) {
    a = a + Math.random();
}
for (j = 0; j < M; j++) {
    b = b + Math.random();
}
```

The first loop is O(N) and the second loop is O(M).
Since N and M are independent variables, so we can't say which one is the leading term.
Therefore,
 Time complexity of the given problem will be O(N+M).

Example: Best vs worst case

Loops:

Worst Case: take maximum

Best Case: take minimum

	worst	best
$i \leftarrow 0$	1	1
while $i < n$ **and** $A[i]$!= 7	n	1
$\quad i \leftarrow i + 1$	n	0
	O(n)	**O(1)**

Worst-case input:

3	**1**	**4**	**2**	**3**	**2**	**1**	**8**	
0	1	2	3	4	5	6	7	**n**

Best-case input:

7	**1**	**5**	**4**	**8**	**2**	**1**	**9**	
0	1	2	3	4	5	6	7	**n**

Our programming language: Java

According to Sun’s Java developers:

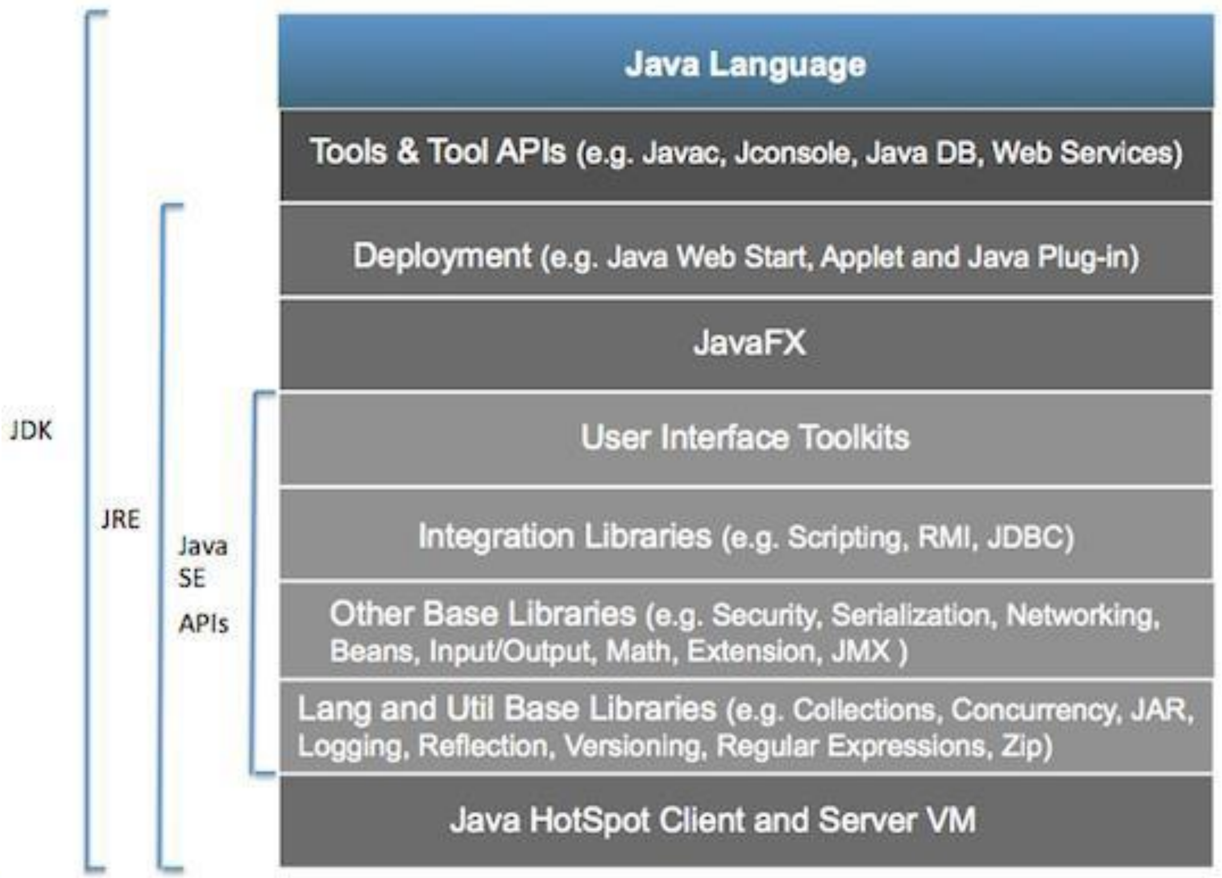
“Java is a simple, object-oriented, distributed, secure, architecture, robust, multi threaded and dynamic language. The program can be written once, and run anywhere”

•It runs on the Java Virtual Machine (JVM)

Main features of Java:

- Object is the main data type, and can be as general as we wanted:
  - public Object read()
  - Public void write(Object x)
- Classes and Interfaces can be generic too:
  - public class MyClass<AnyType> {
    public AnyType read() {…}
    ….
- Static methods can be generic too
  - Public static <AnyType> boolean find(AnyType [] a, AnyType x) { … }
- Generic classes/objects have some restrictions (primitive types, and others)

Java SE Conceptual Diagram



JDK

JRE

Java SE APIs

Java Language

Tools & Tool APIs (e.g. Javac, Jconsole, Java DB, Web Services)

Deployment (e.g. Java Web Start, Applet and Java Plug-in)

JavaFX

User Interface Toolkits

Integration Libraries (e.g. Scripting, RMI, JDBC)

Other Base Libraries (e.g. Security, Serialization, Networking, Beans, Input/Output, Math, Extension, JMX)

Lang and Util Base Libraries (e.g. Collections, Concurrency, JAR, Logging, Reflection, Versioning, Regular Expressions, Zip)

Java HotSpot Client and Server VM

Why Java?

1. Widely Used: Over 9 million developers.
2. Platform Independence: Write Once, Run Anywhere.
3. Object-Oriented: Modular and scalable.
4. Robust and Secure: Strong memory management.
5. Huge Community and Ecosystem: Extensive libraries and frameworks.
6. Used in Enterprise Applications: Financial services, Android apps, etc.

Java SE and EE

- Our implementations will use Java Standard Edition (Java SE)
  - http://www.oracle.com/technetwork/java/javase/overview/index.html
- A more advanced edition is the Java Enterprise Edition (Java EE)
  - http://www.oracle.com/technetwork/java/javaee/overview/index.html
- Java EE is developed using the Java Community Process
- Includes contributions from experts (industry, commercial, organizations, etc.)
- Releases and new features are aligned with contributors
- Main features:
  - a rich platform, widely used, scalable, low risk, etc.
  - enhances HTML5 support and increase developer productivity
  - Java EE developers have support for latest Web applications and frameworks

*Image from www.oracle.com

Installing Java and Eclipse

Installing Java:

Step 1: Visit the official website of Oracle: https://www.oracle.com/java/technologies/downloads/

Step 2: Click on "JDK Download" tab.

Step 3: Select the appropriate operating system for your computer and click on "Download" button for the latest version of Java.

Step 4: Once downloaded, run the installer and follow the instructions to install Java on your computer.

Installing Eclipse:

Step 1: Visit the official website of Eclipse (https://www.eclipse.org/downloads/packages/).

Step 2: Choose the latest version of Eclipse IDE for Java Developers.

Step 3: Select the appropriate operating system for your computer and click on "Download" button for the latest version of Eclipse.

Step 4: Once downloaded, extract the zip file to a location of your choice.

Step 5: Open the extracted folder and double-click on the Eclipse icon to start Eclipse.

You now have Java and Eclipse installed on your computer and are ready to start coding in Java. If you face any issues during the installation process, refer to the documentation provided on the official websites or seek help from your instructor or TA/GA.

References

1. Data Structures and Algorithms Made Easy: Data Structures and Algorithmic Puzzles, 5th Edition, by Narasimha Karumanchi, CareerMonk Publications, 2017.
2. Algorithm Design and Applications by M. Goodrich and R. Tamassia, Wiley, 2015.
3. Data Structures and Algorithms in Java, 6th Edition, by M. Goodrich and R. Tamassia, Wiley, 2014. *(On reserve in the Leddy Library)*
3. Data Structures and Algorithm Analysis in Java, 3rd Edition, by M. Weiss, Addison-Wesley, 2012.
4. Algorithm Design by J. Kleinberg and E. Tardos, Addison-Wesley, 2006.
5. Introduction to Algorithms, 2nd Edition, by T. Cormen et al., McGraw-Hill, 2001.
6. The Eclipse Foundation, http://www.eclipse.org/
7. Java by Oracle: http://www.oracle.com/technetwork/java/index.html
8. Java Standard Edition (Java SE 8) http://www.oracle.com/technetwork/java/javaee/overview/index.html
9. Java Enterprise Edition (Java EE 8) http://www.oracle.com/technetwork/java/javaee/overview/index.html

Exercises

1. Sort the functions $12n^2$, $3n$, $0.5 \log n$, $n \log n$, $2n^3$ in increasing order of growth rate.
2. Algorithm A uses $20 n \log n$ operations, while algorithm B uses $2n^2$ operations. Assume all operations take the same time. What is the value of n0 for which A will run faster? Which algorithm will you use if your inputs are of size 10,000?
3. *Prove or disprove that (a) $f(n) = 2n^2 + 3$ is $O(n)$, (b) $f(n)$ is $O(n^{10})$, (c) $f(n)$ is $\Omega(1)$, (d) $f(n)$ is $\omega(1)$, (e) $f(n)$ is $\omega(n^{0.5})$, (f) $f(n)$ $\Theta(n^2)$, (e) $f(n)$ is $o(n^{9/4})$.
4. Consider A = 3, 4, -7, 3, 6, -3, 2, 8, -1. Find the MCCS using the four algorithms discussed in class. *How many operations will the linear-time algorithm use?
5. Find the worst-case running time in O-notation for algorithms linearMCCS and quadraticMCCS. Show all steps used in the calculations. What about Ω and Θ? Also, ω and o?
6. *Implement linearMCCS and divide-and-conquer-MCCS. Run them on several random sequences of size 1,000. Which one is faster? What are the inputs for which the algorithms run faster/slower? Take the average CPU times and compare the with the running times of the algorithms.
7. Are these statements true? (a) If $f(n)$ is $O(g(n))$, then $f(n)$ is $\Omega(g(n))$; (b) if $f(n)$ is $O(g(n))$, then $g(n)$ is $\Omega(f(n))$; (c) if $f(n)$ is $O(g(n))$ and $f(n)$ is $\Omega(g(n))$, then $f(n)$ is $\Theta(g(n))$; (d) if $f(n)$ is $O(g(n))$ and $g(n)$ is $\Omega(f(n))$, then $f(n)$ is $\Theta(g(n))$; (e) if $f(n)$ is $o(g(n))$, then $f(n)$ is $O(g(n))$; (f) if $f(n)$ is $\Omega(g(n))$, then $f(n)$ is $\omega(g(n))$.
8. Give five different reasons for why we use Java in this course.
9. Describe the main features of the Eclipse IDE platform.
10. What is the advantage of using Java Enterprise Edition?
11. Consider A = 3, 4, -7, 3, 6, -3, 2, 8, -1. Can the algorithm linear MCCS be modified to run a little bit faster on this example? Yes/no? why not?
12. Give an example of an algorithm whose best and worst case running times differ in more than a constant (i.e., asymptotically different).
13. *Implement linear search and binary search on an array A. Run a variety experiments on different lists of different sizes. Which algorithm will you use and for which sizes?
14. List the main properties of the rules for O-notation. Do these apply to Ω and Θ? How?
15. What is the difference between problem and algorithm? Explain how a problem can be solved using different algorithms and how they may have different complexities. Give an example.
16. Give examples of functions for the different types of asymptotic notation.