SD 421

Algorithms:
The Principle of Least Action

Lecture 1: Introduction to Analysis of Algorithms

Wholeness Statement

The study of algorithms is a core part of computer science and brings the scientific method to the discipline; it has its theoretical aspects (a systematic expression in mathematics), can be verified experimentally, has a wide range of applications, and has a record of achievements.

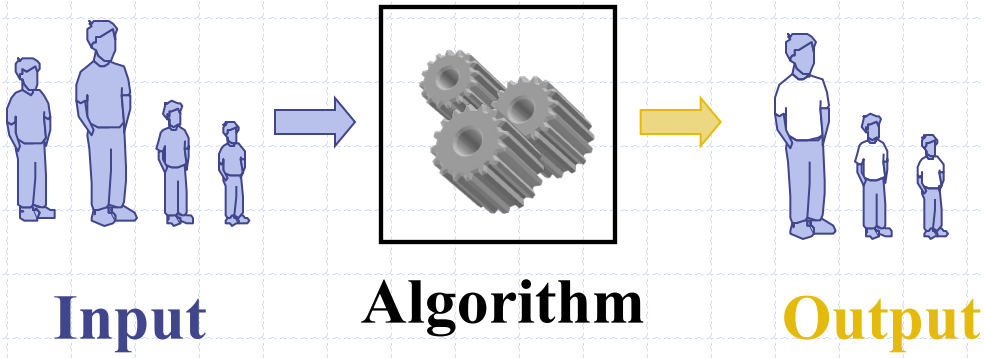
The Science of Consciousness also has theoretical and experimental aspects, and can be applied and verified universally by anyone.

Overview

- About Algorithms & Why to study
- Analysis of Algorithms
  - Computational Complexity
  - Pseudocode
  - Growth Rate of Running Time
  - Asymptotic analysis

What is an algorithm?

- An **algorithm** is simply a step-by-step procedure for solving a problem in a finite amount of time.
  - Has a unique first step
  - Each step has a unique successor step
  - Sequence of steps terminate with desired output.



Why study algorithms?

- You might ask, “why would I want to know any of the details of the sorting algorithms?”
  - If I want to sort, I’ll just call a library sort routine.
- This is fine if you’re going to remain a programmer in a simplistic since,
  - i.e., just get the job done without regard to efficiency, best practices, or high-level knowledge of how things are done by API’s

Computer Science Majors

- CS majors should at least be exposed to and have a basic understanding of what the different sorting algorithms do because
  - there are tradeoffs between the different sorting algorithms
  - they illustrate important algorithm design strategies
    - We need people who can develop and implement new algorithms based on exposure to well designed algorithms
    - Also, we don't have to be great at math, but we need some math to push computer science forward (particularly in algorithms)

Natural Things to Ask

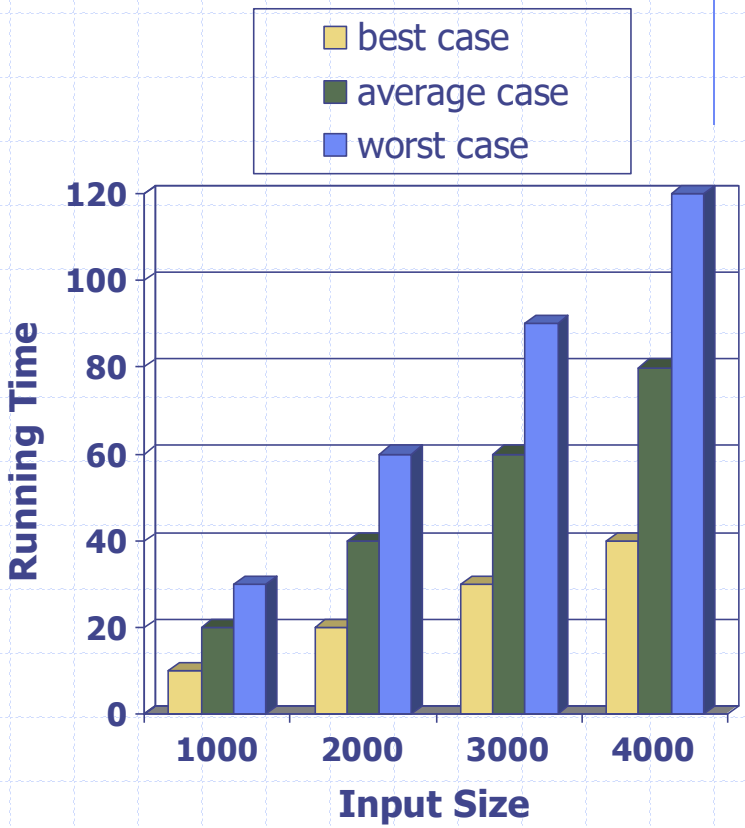
- How can we determine whether an algorithm is *efficient* ?
- Given two algorithms that achieve the same goal, how can we decide which one is better? (E.g. sorting)
  - Can our analysis be independent of a particular operating system or implementation in a language?
- How can we express the steps of an algorithm without depending on a particular implementation or programming language?

Computational Complexity

- The theoretical study of time and space requirements of algorithms
- Time complexity is the amount of work done by an algorithm
  - Roughly proportional to the critical (inherently important) operations

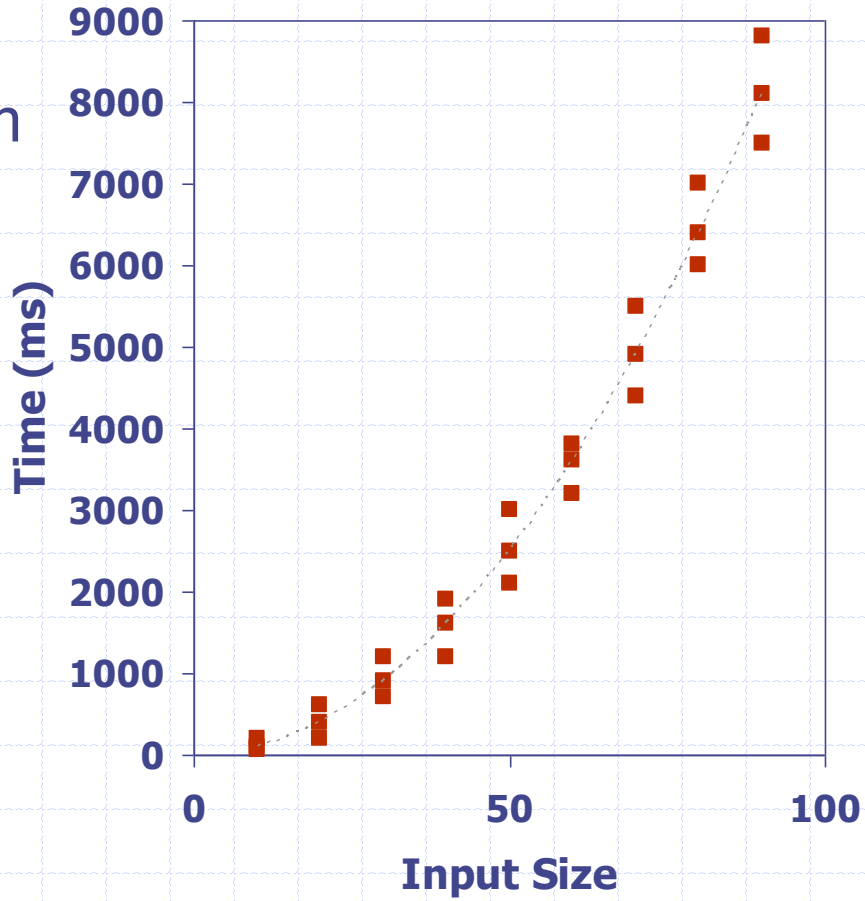
Running Time

- Most algorithms transform input objects into output objects.
- The running time of an algorithm typically grows with the input size.
- Average case time is often difficult to determine.
- So, we usually focus on 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 varying size and composition
- Use a method like System.currentTimeMillis() to get an accurate measure of the actual running time
- Plot the results



Limitations of Experiments

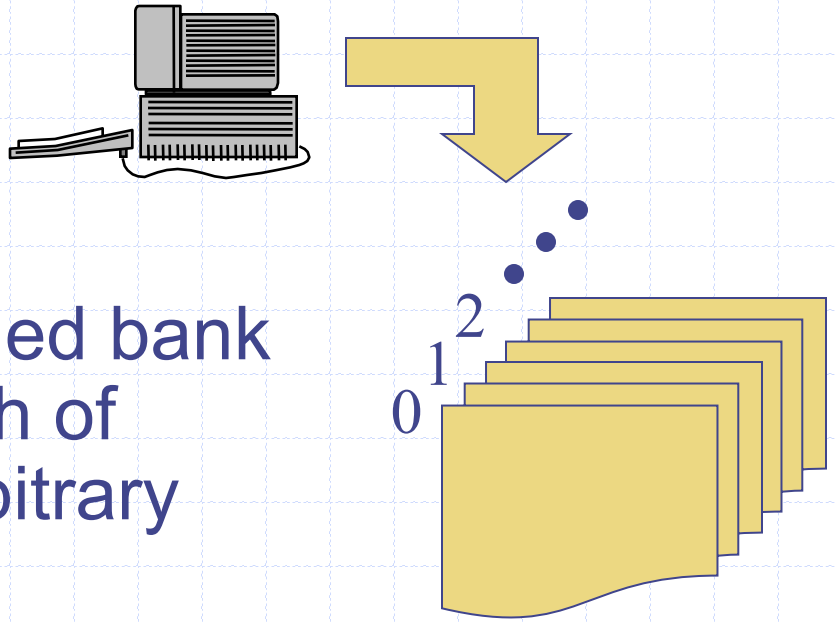
- Requires implementation of the algorithm,
  - which may be difficult and/or time consuming
- Results may not be indicative of the running time on other inputs not included in the experiment.
- To compare two algorithms,
  - the same hardware and software environments must be used
  - better to compare before actually implementing them (to save time)

Theoretical Analysis

- A high-level description of the algorithm is used instead of an implementation
- Running time is characterized as a function of the input size, *n*
- Takes into account all possible inputs
- Allows evaluation of the speed of an algorithm independent of the hardware/software environment

The Random Access Machine (RAM) Model

- 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.

Algorithm 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)
- Assumed to take a constant amount of time in the RAM model

Primitive Operations in this Course

- Performing an arithmetic operation (+, *, etc., a+b)
- Comparing two numbers (cnt != prvCnt)
- Assigning a value to a variable (cnt ← cnt+1)
- Indexing into an array (A[5])
- Calling a method (mySort(A,n))
- Returning from a method (return(cnt))
- Following an object reference(ob.x)

Pseudocode Details

- Control flow
  - **if** … **then** … [**else …**]
  - **while** … **do** …
  - **do** … **while** …
  - **for** … **do** …
  - Indentation replaces braces
- Method declaration
  - **Algorithm** ***method*** (***arg*** [, ***arg***…])
  - **Input** …
  - **Output** …
- Method call
  - ***var.method*** (***arg*** [, ***arg***…])
- Return value
  - **return** ***expression***
- Expressions
  - Assignment (like = in Java)
  - Equality testing (like == in JavaScript)
  - n^2 Superscripts and other mathematical formatting allowed

Pseudocode Example

- 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

Example: find max element of an array

Algorithm ***arrayMax***(A, n)
  Input array A of $n > 0$ 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 $currentMax$

Counting Primitive Operations

- Inspect the pseudocode to determine the maximum number of primitive operations executed by an algorithm as a function of the input size

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

                                                  Total
```

Counting Primitive Operations

- Inspect the pseudocode to determine the maximum number of primitive operations executed by an algorithm as a function of the input size

Algorithm ***arrayMax***(A, n)

	# operations
$currentMax \leftarrow A[0]$	2
for $i \leftarrow 1$ **to** $n - 1$ **do**	$1 + n$
if $A[i] > currentMax$ **then**	$2(n - 1)$
$currentMax \leftarrow A[i]$	$2(n - 1)$
{ increment counter i (add & assign) }	$2(n - 1)$
return $currentMax$	1
Total	$7n - 2$

Estimating Running Time

- Algorithm *arrayMax* executes $7n - 2$ primitive operations in the worst 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

$$a(7n - 2) \leq T(n) \leq b(7n - 2)$$

- Hence, the running time $T(n)$ is bounded by two linear functions

Basic Rules For Computing Asymptotic Running Times

- **Rule-1: For Loops**
  The running time of a for loop is at most the running time of the statements inside the loop (including tests) times the number of iterations (see ***arrayMax***)

- **Rule-2: Nested Loops**
- Analyze from inside out. The total running time of a statement inside a group of nested loops is the running time of the statement times the sizes of all the loops

```
for i ← 0 to n-1 do  ( n times)
    for j ← 0 to n-1 do (n times)
        k ← i + j
```

(Runs in $O(n^2)$)

(continued)

- **Rule-3: Consecutive Statements**

  Running times of consecutive statements should be added in order to compute running time of the whole

```
for i ← 0 to n-1 do ( n times)
    a[i] ←  0
for i ← 0 to n-1 do ( n times)
        for j ← 0 to i  do (n^2 times)
            a[i] ← a[i] + i + j
```

j value depends on i.
i=0, then j executes 1 time
i=1, then j executes 2 time
i=2, then j executes 3 time
i=n-1, then j executes n time
It's 1 + 2 + 3 +..+n, which is sum of n natural numbers = $n(n+1)/2 \rightarrow (n^2 + n)/2$. Take the highest polynomial is n^2

(Running time is $O(n) + O(n^2)$. This is $O(n^2)$)

(continued)

- **Rule-4: If/Else**

  For the fragment

  if *condition* **then**
  S1
  else
  S2

  the running time is never more than the running time of the *condition* plus the larger of the running times of S1 and S2.

Examples: Analyze running times

```
int[] arrays(int n) {
   int[] arr = new int[n];
   for(int i = 0; i < n; ++i){
         arr[i] = 1;
   }
   for(int i = 0; i < n; ++i) {
         for(int j = i; j < n; ++j){
               arr[i] += arr[j] + i + j;
         }
   }
   return arr;
}
```

The first for loop executes n times

The outer loop executes n times
The inner loop can be executed approximated by the sum of integers from 0 to n-1 = n(n+1)/2 → $(n^2 + n)/2$.
Take the highest polynomial is n^2

$n + n^2 = O(n^2)$

Main Point

1. Complexity analysis determines the resources (time and space) needed by an algorithm so we can compare the relative efficiency of various algorithmic solutions. To design an efficient algorithm, we need to be able to determine its complexity so we can compare any refinements of that algorithm so we know when we have created a better, more efficient solution.

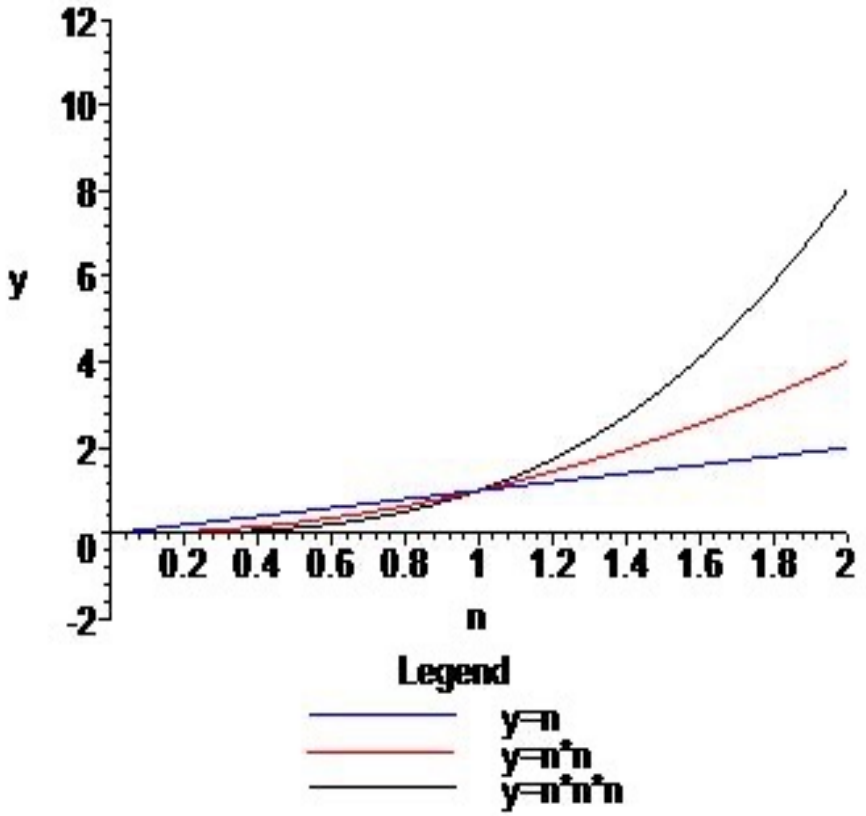
Science of Consciousness: Through regular deep rest (transcending) and dynamic activity we refine our mind and body until our thoughts and actions become most efficient; in the state of enlightenment, the conscious mind operates at the level of pure consciousness, which always operates with maximum efficiency, according to the natural law of least action, so we can spontaneously fulfill our desires and solve even non-computable problems.

Asymptotic Analysis

- **Asymptote**:
  - A line whose distance from a given curve approaches zero as they tend to infinity
    - A term derived from analytic geometry
  - Originates from the Greek word *asymptotes* which means not intersecting
  - Thus an asymptote is a limiting line
- **Asymptotic**:
  - Relating to or having the nature of an asymptote
- **Asymptotic analysis in complexity theory**:
  - Describes the upper (or lower) bounds of an algorithm's behavior in terms of its usage of time and space
  - Used to classify computational problems and algorithms according to their inherent difficulty
- We are going to classify algorithms in terms of functions of their input size
  - Therefore, how can we draw graphs of quadratic or cubic functions so the graphs look and behave like asymptotes (a limiting line)?

Growth Rates

- Growth rates of functions:
  - Linear $\approx n$
  - Quadratic $\approx n^2$
  - Cubic $\approx n^3$
- The graph of the cubic begins as the slowest but eventually overtakes the quadratic and linear graphs
- Main factor for growth rates is the behavior as n gets large

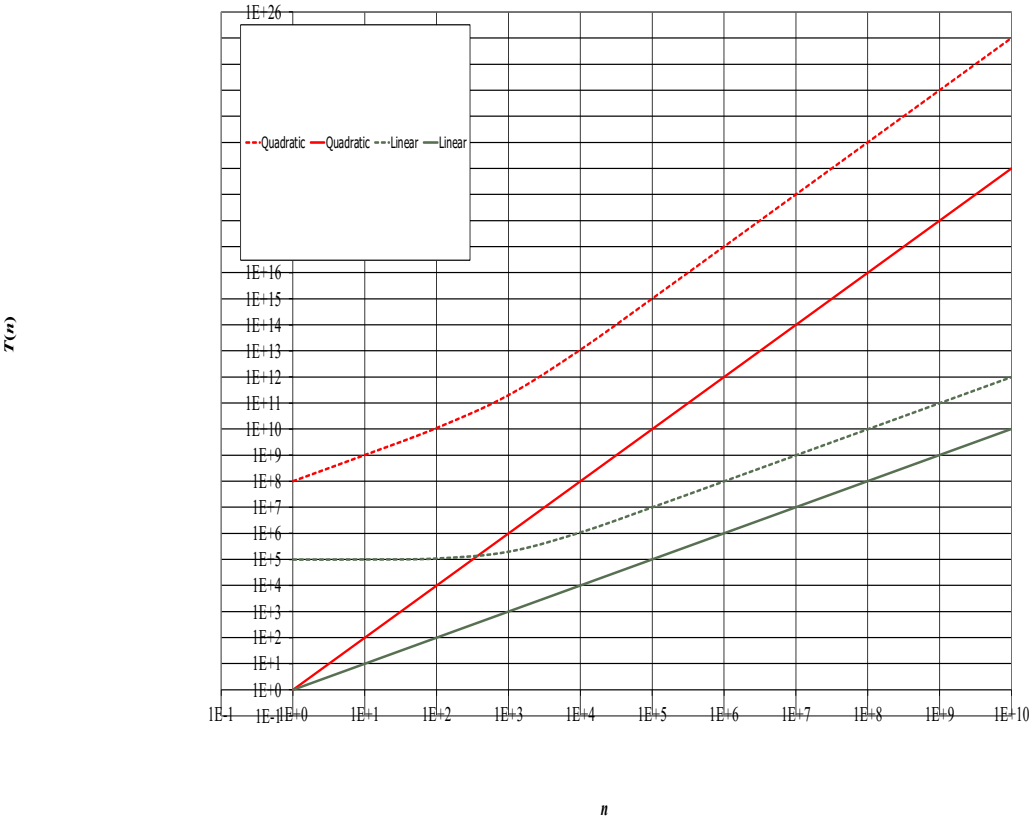


Growth Rate of Running Time

- The hardware/software environment
  - Affects $T(n)$ by a constant factor,
  - But does not alter the asymptotic growth rate of $T(n)$
- For example: The linear growth rate of the running time $T(n)$ is an intrinsic property of algorithm ***arrayMax***

Constant Factors

- The growth rate is not affected by
  - constant factors or
  - lower-order terms
- Examples
  - $10^2n + 10^5$ is a linear function
  - $10^5n^2 + 10^8n$ is a quadratic function



Big-Oh Notation

- Definition:
  - 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
  - $f(n) \leq cg(n)$ for $n \geq n_0$
- Example:
  - prove that $2n + 10$ is $O(n)$

Big-oh

- If f(n) *grows no faster* than g(n), we say f(n) is O(g(n)) ("big-oh")
- We also say that g(n) is an ***asymptotic upper bound*** on f(n)
- Limit criterion(for comparing growth):

  f(n) is O(g(n)) if

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} \text{ is finite}$$

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 $g(n)$. $f(n) \leq cg(n)$
- 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.,
  1. Drop lower-order terms
  2. 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 $O(3n)$"

In-Class Exercise- Big-Oh

- $7n-2$ is _____
- $3n^3 + 20n^2 + 5$ is _____
- $3 \log n + \log \log n$ is _____

In-Class Exercise- Big-Oh

- 7n-2 is **O(n)**
- $3n^3 + 20n^2 + 5$ is **$O(n^3)$**
- 3 log n + log log n is **O(log n)**

Standard Complexity Classes

- The most common complexity classes used in analysis of algorithms are, in increasing order of growth rate:

$$\Theta(1), \Theta(\log n), \Theta(n^{1/k}), \Theta(n), \Theta(n\log n), \Theta(n^k)\ (k > 1),$$
$$\Theta(2^n), \Theta(n!), \Theta(n^n)$$

- Functions that belong to classes in the first row are known as *polynomial time bounded* because they are $O(n^k)$ for some $k \geq 0$.

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 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)

little-oh

- f(n) is o(g(n)) if f(n) is **asymptotically strictly less** than g(n)

little-omega

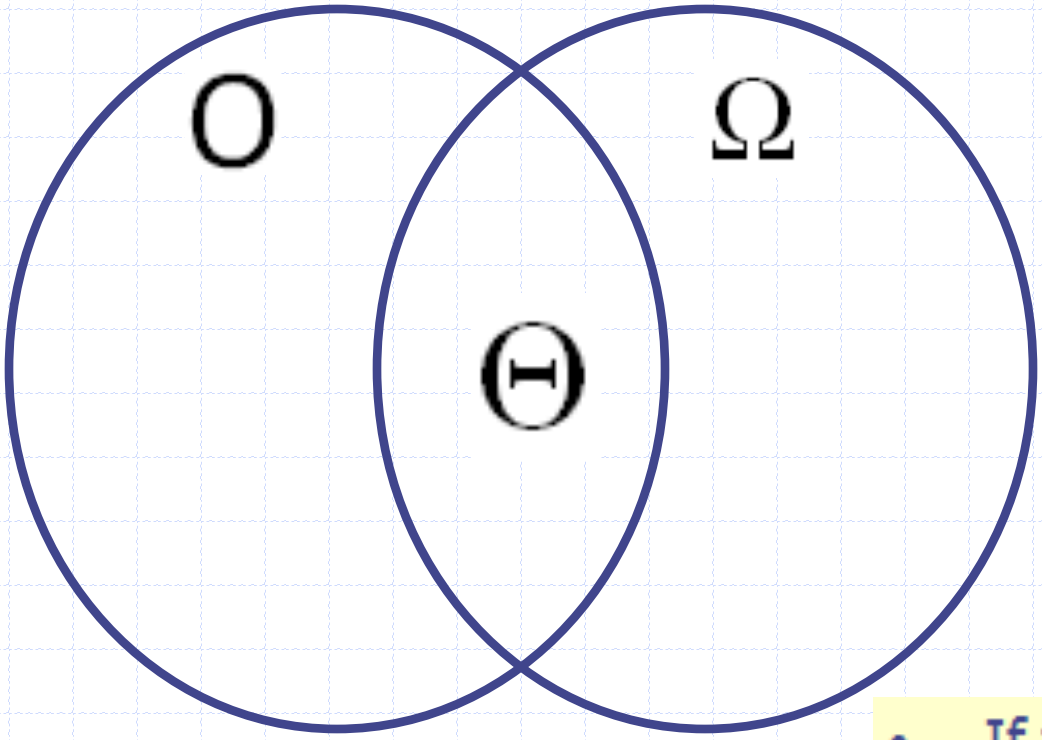
- f(n) is μ(g(n)) if f(n) is **asymptotically strictly greater** than g(n)

In this course we focus of big-Oh(upper bound) and big-Omega(lower bound).

Examples

- Both $2n + 1$ and $3n^2$ are $O(n^2)$ (Upper bound)
  - f(n) is **asymptotically less than or equal** to g(n)
- Both $2n^2 + 1$ and $3n^2$ are $\Theta(n^2)$
  - f(n) is **asymptotically equal** to g(n)
- Both $6n^3 + 2$ and $4n^2$ are $\Omega(n^2)$ (Lower bound)
  - f(n) is **asymptotically greater than or equal** to g(n)

Relationships Between the Complexity Classes



- If f(n) is in both O(g(n)) and Ω(g(n)), it is in Θ(g(n)).

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 determine that algorithm *arrayMax* executes at most $7n-2$ primitive operations
  - We say that algorithm *arrayMax* "runs in $O(n)$ time"
- Since constant factors and lower-order terms are eventually dropped, we can disregard them when counting primitive operations

Counting Primitive Operations using Big-oh Notation

- Why don't we need to precisely count every primitive operation like we did previously?

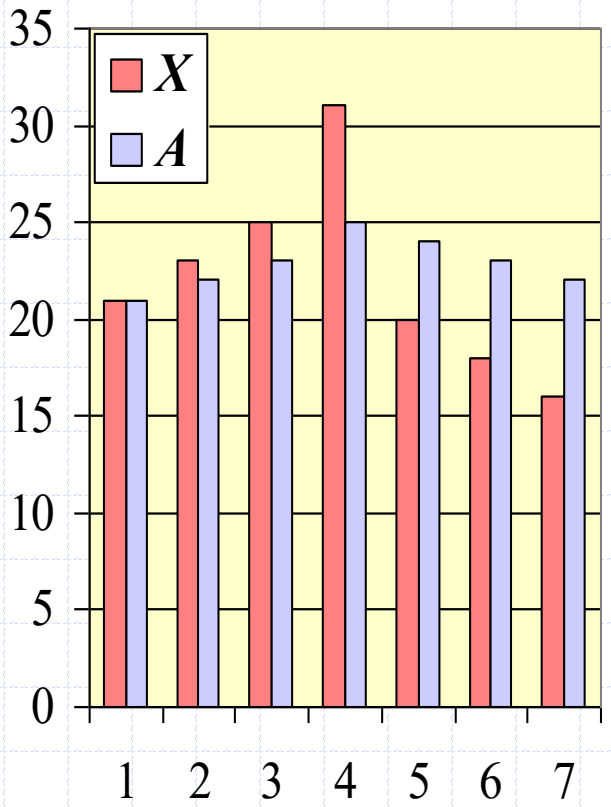
Algorithm *arrayMax*(*A*, *n*)	# operations
currentMax ← *A*[0]	O(1)
for *i* ← 1 **to** *n* – 1 **do**	O(*n*)
if *A*[*i*] > *currentMax* **then**	O(*n*)
currentMax ← *A*[*i*]	O(*n*)
{ increment counter *i* (add & assign) }	O(*n*)
return *currentMax*	O(1)
Total	O(*n*)

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:

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

- Computing the array A of prefix averages of another array X has applications to financial analysis



Prefix Averages (Quadratic)

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

```
Algorithm prefixAverages1(X, n)
    Input array X of n integers
    Output array A of prefix averages of X          #operations
    A ← new array of n integers                          n
    for i ← 0 to n − 1 do                                n
        s ← X[0]                                         n
        for j ← 1 to i do                    1 + 2 + …+ (n − 1)
            s ← s + X[j]                     1 + 2 + …+ (n − 1)
        A[i] ← s / (i + 1)                               n
    return A                                             1
```

Arithmetic Progression

- The running time of ***prefixAverages1*** is $O(1 + 2 + \ldots + n)$
- The sum of the first n integers is $n(n + 1) / 2$
- Thus, algorithm ***prefixAverages1*** runs in $O(n^2)$ time

Prefix Averages (Linear)

- The following algorithm computes prefix averages in linear time by keeping a running sum

```
Algorithm prefixAverages2(X, n)
    Input array X of n integers
    Output array A of prefix averages of X          #operations
    A ← new array of n integers                     n
    s ← 0                                           1
    for i ← 0 to n − 1 do                           n
        s ← s + X[i]                                n
        A[i] ← s / (i + 1)                          n
    return A                                        1
```

- Algorithm *prefixAverages2* runs in $O(n)$ time

Main Point

2. An algorithm is “optimal” if its computational complexity is equal to the “maximal lower bound” of all algorithmic solutions to that problem; that is, an algorithm is optimal if it can be proven that no algorithmic solution can do asymptotically better.

Science of Consciousness: An individual’s actions are optimal if they are the most effective and life-supporting. Development of higher states of consciousness results in optimal action because thoughts are performed while established in the silent state of pure consciousness, the source of creativity and intelligence in nature.

Connecting the Parts of Knowledge with the Wholeness of Knowledge

1. An algorithm is like a recipe to solve a computable problem starting with an initial state and terminating in a definite end state.
2. To help develop the most efficient algorithms possible, mathematical techniques have been developed for formally expressing algorithms (pseudocode) so their complexity can be measured through mathematical reasoning and analysis; these results can be further tested empirically.

3. **Transcendental Consciousness** is the home of all knowledge, the source of thought. The TM technique is like a recipe we can follow to experience the home of all knowledge in our own awareness.
4. **Impulses within Transcendental Consciousness**: Within this field, the laws of nature continuously calculate and determine all activities and processes in creation.
5. **Wholeness moving within itself** : In unity consciousness, all expressions are seen to arise from pure simplicity--diversity arises from the unified field of one's own Self.