King Fahd University of Petroleum & Minerals
College of Computer Science & Engineering

Information & Computer Science Department

Unit 2

Introduction to Complexity Analysis

Reading Assignment

- "Data Structures and Algorithms in Java", 3rd Edition, Adam Drozdek, Cengage Learning, ISBN 978-9814239233
  - Chapter 2 (Sections 1 – 8)
  - Sections 2.9: Amortized Complexity and 2.10: NP-Completeness are not included.

Outline

1. Computational and Asymptotic Complexity
2. Big-O Notation
3. Properties of Big-O Notation
4. Ω and Θ Notations
5. Warnings about Big-O Notation
6. Examples of Complexities
7. Finding Asymptotic Complexity: Examples
8. The Best, Average, and Worst Cases

How do we Measure Efficiency?

- There are often many different *algorithms* which can be used to solve the same problem.
  - For example, assume that we want to search for a key in a sorted array.
- Thus, it makes sense to develop techniques that allow us to:
  - compare different algorithms with respect to their "efficiency"
  - choose the most efficient algorithm for the problem

- The ***efficiency*** of any algorithmic solution to a problem can be measured according to the:
  - **Time efficiency**: the time it takes to execute.
  - **Space efficiency**: the space (primary or secondary memory) it uses.

- We will focus on an algorithm's efficiency with respect to time.

How do we Measure Efficiency?

- Running time in [micro/milli] seconds
  - Advantages
  - Disadvantages

Computational and Asymptotic Complexity

- **Computational complexity** measures the degree of difficulty of an algorithm
- Indicates how much effort is needed to apply an algorithm or how costly it is
- To evaluate an algorithm's efficiency, use logical units that express a relationship such as:
  - The size *n* of a file or an array
  - The amount of time *T* required to process the data
- Hence, it makes sense to specify that in terms of $T(n)$.

Computational and Asymptotic Complexity

- This measure of efficiency is called **asymptotic complexity**
- It is used when disregarding certain terms of a function
  - To express the efficiency of an algorithm
  - When calculating a function is difficult or impossible and only approximations can be found

$$f(n) = n^2 + 100n + \log_{10}n + 1,000$$

Computational and Asymptotic Complexity

n	$f(n)$	n^2		$100n$		$\log_{10}n$		$1{,}000$	
	Value	Value	%	Value	%	Value	%	Value	%
1	1,101	1	0.1	100	9.1	0	0.0	1,000	90.83
10	2,101	100	4.76	1,000	47.6	1	0.05	1,000	47.60
100	21,002	10,000	47.6	10,000	47.6	2	0.001	1,000	4.76
1,000	1,101,003	1,000,000	90.8	100,000	9.1	3	0.0003	1,000	0.09
10,000	101,001,004	100,000,000	99.0	1,000,000	0.99	4	0.0	1,000	0.001
100,000	10,010,001,005	10,000,000,000	99.9	10,000,000	0.099	5	0.0	1,000	0.00

Figure 2-1 The growth rate of all terms of function

$$f(n) = n^2 + 100n + \log_{10}n + 1{,}000$$

Big-O Notation

- Introduced in 1894, the big-O notation specifies asymptotic complexity, which estimates the rate of function growth
- **Definition 1:** $f(n)$ is $O(g(n))$ if there exist positive numbers c and N such that

$$f(n) \le cg(n) \text{ for all } n \ge N$$

c	≥ 6	$\ge 3\frac{3}{4}$	$\ge 3\frac{1}{9}$	$\ge 2\frac{13}{16}$	$\ge 2\frac{16}{25}$	$\dots$	$\rightarrow$	2
N	1	2	3	4	5	$\dots$	$\rightarrow$	∞

Figure 2-2 Different values of c and N for function $f(n) = 2n^2 + 3n + 1 = O(n^2)$ calculated according to the definition of big-O

Big-O Notation (continued)

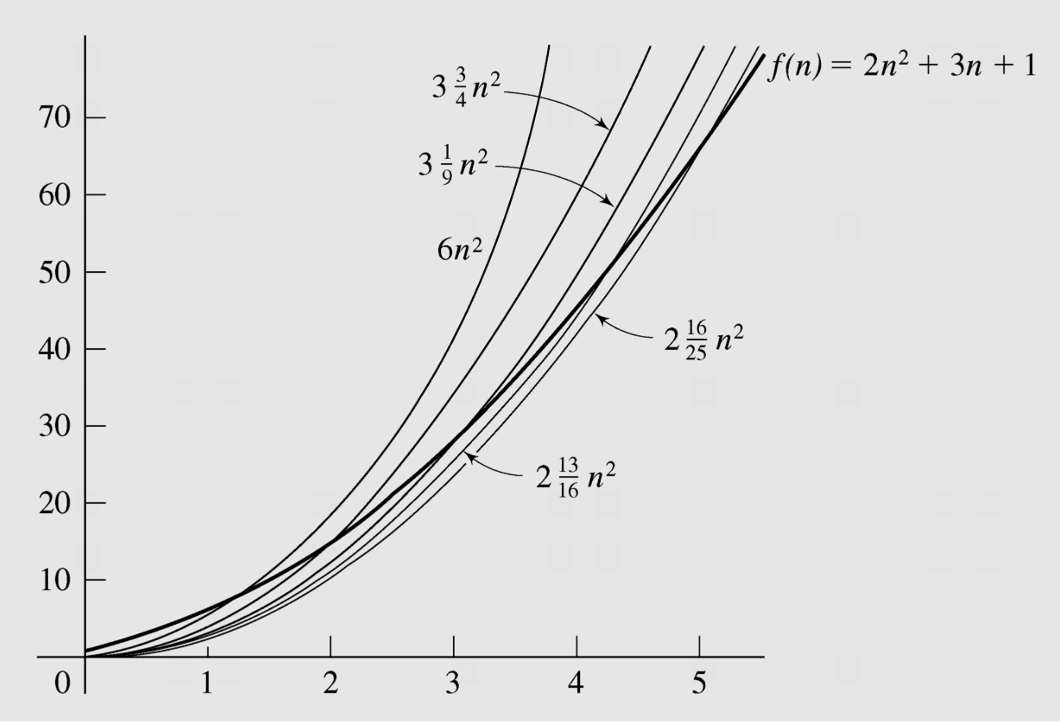

Figure 2-3 Comparison of functions for different values of *c* and *N* from Figure 2-2

Properties of Big-O Notation

- Fact 1 (transitivity)
  If $f(n)$ is $O(g(n))$ and $g(n)$ is $O(h(n))$, then $f(n)$ is $O(h(n))$
- Fact 2
  If $f(n)$ is $O(h(n))$ and $g(n)$ is $O(h(n))$, then $f(n) + g(n)$ is $O(h(n))$
- Fact 3
  The function an^k is $O(n^k)$

Properties of Big-O Notation

- Fact 4
  The function n^k is $O(n^{k+j})$ for any positive j
- Fact 5
  If $f(n) = cg(n)$, then $f(n)$ is $O(g(n))$
- Fact 6
  If $f(n) = g(n) + h(n)$, then $f(n)$ is $O(\max\{g(n), h(n)\})$
- Fact 7
  If $f(n) = g(n)*h(n)$, then $f(n)$ is $O(g(n)*h(n))$
- Fact 8
  The function $\log_a n$ is $O(\log_b n)$ for any positive numbers a and $b \neq 1$
- Fact 9
  $\log_a n$ is $O(\lg n)$ for any positive $a \neq 1$, where
  $\lg n = \log_2 n$

Ω and Θ Notations

- Big-O notation refers to the upper bounds of functions
- There is a symmetrical definition for a lower bound in the definition of big-Ω
- **Definition 2:** The function $f(n)$ is $\Omega(g(n))$ if there exist positive numbers c and N such that

$$f(n) \geq cg(n) \text{ for all } n \geq N$$

Ω and Θ Notations

- The difference between this definition and the definition of big-O notation is the direction of the inequality
- One definition can be turned into the other by replacing "≥" with "≤"
- There is an interconnection between these two notations expressed by the equivalence

$$f(n) \text{ is } \Omega(g(n)) \text{ iff } g(n) \text{ is } O(f(n))$$

Ω and Θ Notations

- **Definition 3:** $f(n)$ is $\Theta(g(n))$ if there exist positive numbers c_1, c_2, and N such that $c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all $n \geq N$
- When applying any of these notations (big-O, Ω, and Θ), remember they are approximations that hide some detail that in many cases may be considered important

Warnings about O-Notation

- Big-O notation cannot compare algorithms in the same complexity class.
- Big-O notation only gives sensible comparisons of algorithms in different complexity classes when **n** is large .
- Consider two algorithms for same task:
  Linear: $\mathbf{f(n) = 1000\ n}$
  Quadratic: $\mathbf{f'(n) = n^2/1000}$
  The quadratic one is faster for $n < 1000000$.

Examples of Complexities

- Algorithms can be classified by their time or space complexities
- An algorithm is called **constant** if its execution time remains the same for any number of elements, and is denoted by $O(1)$
- It is called **quadratic** if its execution time is $O(n^2)$

Examples of Complexities

Class	Complexity	Number of Operations and Execution Time (1 instr/μsec)					
n		10		10^2		10^3	
constant	$O(1)$	1	1 μsec	1	1 μsec	1	1 μsec
logarithimic	$O(\lg n)$	3.32	3 μsec	6.64	7 μsec	9.97	10 μsec
linear	$O(n)$	10	10 μsec	10^2	100 μsec	10^3	1 msec
O(n lg n)	$O(n \lg n)$	33.2	33 μsec	664	664 μsec	9970	10 msec
quadratic	$O(n^2)$	10^2	100 μsec	10^4	10 msec	10^6	1 sec
cubic	$O(n^3)$	10^3	1 msec	10^6	1 sec	10^9	16.7 min
exponential	$O(2^n)$	1024	10 msec	10^{30}	$3.17 * 10^{17}$ yrs	10^{301}	

Figure 2-4 Classes of algorithms and their execution times on a computer executing 1 million operations per second (1 sec = 10^6 μsec = 10^3 msec)

Examples of Complexities

n		10^4		10^5		10^6	
constant	$O(1)$	1	1 μsec	1	1 μsec	1	1 μsec
logarithmic	$O(\lg n)$	13.3	13 μsec	16.6	7 μsec	19.93	20 μsec
linear	$O(n)$	10^4	10 msec	10^5	0.1 sec	10^6	1 sec
$O(n \lg n)$	$O(n \lg n)$	133×10^3	133 msec	166×10^4	1.6 sec	199.3×10^5	20 sec
quadratic	$O(n^2)$	10^8	1.7 min	10^{10}	16.7 min	10^{12}	11.6 days
cubic	$O(n^3)$	10^{12}	11.6 days	10^{15}	31.7 yr	10^{18}	31,709 yr
exponential	$O(2^n)$	10^{3010}		10^{30103}		10^{301030}	

Figure 2-4 Classes of algorithms and their execution times on a computer executing 1 million operations per second (1 sec = 10^6 μsec = 10^3 msec) (continued)

Examples of Complexities

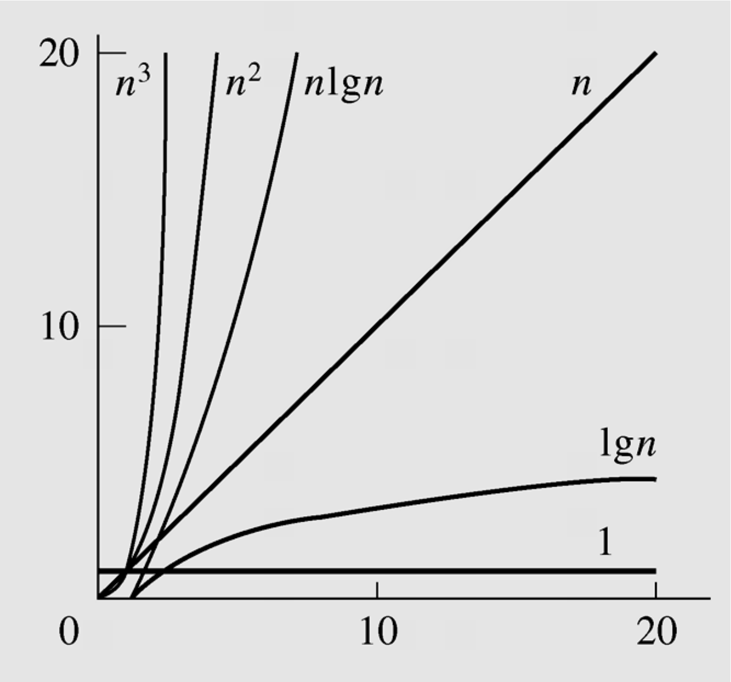

Figure 2-5 Typical functions applied in big-O estimates

Finding Asymptotic Complexity: Examples

- **Asymptotic bounds** are used to estimate the efficiency of algorithms by assessing the amount of time and memory needed to accomplish the task for which the algorithms were designed

```
for (i = sum = 0; i < n; i++)
        sum += a[i];
```

Finding Asymptotic Complexity: Examples

- Represent the cost of the for loop in summation form.
  - The main idea is to make sure that we find an iterator that increases/decreases its value by 1.
  - For example, consider finding the number of times statements 1 and 2 get executed below:

```
for (int i = 1; i <= n; i++)
  for (int j = 1; j <= n; j++)
       statement1;
 }
```

$$\sum_{i=1}^{n}\sum_{j=1}^{n} 1 = \sum_{i=1}^{n} n = n\sum_{i=1}^{n} 1 = n^2$$

```
for (int i = 1; i <= n; i++)
  for (int j = 1; j <= i; j++)
       statement2;
 }
```

$$\sum_{i=1}^{n}\sum_{j=1}^{i} 1 = \sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

Finding Asymptotic Complexity: Examples

- Represent the cost of the for loop in summation form.
  - The problem in the example below is that the value of i does not increase by 1

```
for (int i = k; i <= n; i = i + m)
        statement1;
```

- i: k , k + m , k + 2m , …, k + rm
  - Here, we can assume without loss of generality that k + rm = n, i.e. r = (n – k)/m
  - i.e., an iterator s from 0, 1, …,r can be used

$$\sum_{s=0}^{r} 1 = \sum_{s=0}^{n-k/m} 1 = \frac{n-k}{m} - 0 + 1 = \frac{n-k}{m} + 1$$

Finding Asymptotic Complexity: Examples

```
for (i = 0; i < n; i++) {
      for (j = 1, sum = a[0]; j <= i; j++)
            sum += a[j];
      System.out.println ("sum for subarray 0 through "+i+" is"
            + sum);
}


for (i = 4; i < n; i++) {
      for (j = i-3, sum = a[i-4]; j <= i; j++)
            sum += a[j];
      System.out.println ("sum for subarray "+(i - 4)+" through
   "+i+"        is"+ sum);
}
```

Finding Asymptotic Complexity: Examples

```
for (i = 0, length = 1; i < n-1; i++) {
    for (i1 = i2 = k = i; k < n-1 && a[k] < a[k+1];
         k++, i2++);
    if (length < i2 - i1 + 1)
      length = i2 - i1 + 1;
    System.out.println ("the length of the longest
       ordered subarray is" + length);
}
```

Finding Asymptotic Complexity: Examples

```
int binarySearch(int[] arr, int key) {
    int lo = 0, mid, hi = arr.length-1;
    while (lo <= hi) {
        mid = (lo + hi)/2;
        if (key < arr[mid])
           hi = mid - 1;
        else if (arr[mid] < key)
           lo = mid + 1;
        else return mid; // success
    }
    return -1;        // failure
}
```

Finding Asymptotic Complexity: Examples

- Suppose **n** is a power of 2. Determine the number of times statement 1 is executed:

```
static int myMethod(int n){
   int sum = 0;
   for(int i = 1; i <= n; i = i * 2)
        sum = sum + i + helper(i);
    return sum;
 }
```

```
static int helper(int n){
     int sum = 0;
     for(int i = 1; i <= n; i++)
          sum = sum + i; //statement1
     return sum;
}
```

- Solution:
  - The variables *i* and *n* in myMethod are different from the ones in the helper method.
    - In fact, *n* of "helper" is being called by variable *i* in "myMethod".
    - Hence, we need to change the name of variable *i* in helper because it is independent from *i* in myMethod (let us call it k).
  - We count the number of times statement1 gets executed as follows:
  - (in myMethod) i : $1, 2, 2^2, 2^3, \ldots, 2^r = n$ $(r = \log_2 n)$
    Hence, we can use j where $i = 2^j$ j : $0, 1, 2, 3, \ldots, r = \log_2 n$

$$\sum_{j=0}^{r} \text{cost}(\text{Helper}(i)) = \sum_{j=0}^{r} \sum_{k=1}^{i} 1 = \sum_{j=0}^{r} i = \sum_{j=0}^{r} 2^j = 2^{r+1} - 1 = 2n - 1$$

Some Useful Formulas

$$\sum_{i=m}^{n} c = c\left(\sum_{i=m}^{n} 1\right) = c.(n-m+1)$$

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

$$\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$$

$$\sum_{i=1}^{n} i^3 = \left(\frac{n(n+1)}{2}\right)^2$$

$$\sum_{i=0}^{n} a^i = \frac{a^{n+1}-1}{a-1}, a \neq 1$$

Useful Logarithmic Formulas

$$\log_b a = \frac{\ln a}{\ln b} \quad , \quad \log ab = \log a + \log b$$

$$\log \frac{a}{b} = \log a - \log b \quad , \quad a^{\log_a b} = b$$

$$\left(a^b\right)^c = \left(a^c\right)^b = a^{bc}$$

How to determine complexity of code structures

Sequence of statements: Use Addition rule

O(s1; s2; s3; … sk) = O(s1) + O(s2) + O(s3) + … + O(sk)

= O(max(s1, s2, s3, . . . , sk))

Example:

```
for (int j = 0; j < n * n; j++)
   sum = sum + j;
for (int k = 0; k < n; k++)
   sum = sum - l;
System.out.print("sum is now ” + sum);
```

Complexity is $O(n^2) + O(n) + O(1) = O(n^2)$

How to determine complexity of code structures

Switch: Take the complexity of the most expensive case

```
char key;
int[] X = new int[n];
int[][] Y = new int[n][n];
........
switch(key)  {
    case 'a':
       for(int i = 0; i < X.length; i++)
          sum += X[i];
    break;
    case 'b':
       for(int i = 0; i < Y.length; j++)
          for(int j = 0; j < Y[0].length; j++)
               sum += Y[i][j];
    break;
  }  // End of switch block
```

o(n)

$o(n^2)$

Overall Complexity: $O(n^2)$

How to determine complexity of code structures

If Statement: Take the complexity of the most expensive case **:**

```
char key;
    int[][] A = new int[n][n];
    int[][] B = new int[n][n];
    int[][] C = new int[n][n];
    ........
    if(key == '+')  {
        for(int i = 0; i < n; i++)
          for(int j = 0; j < n; j++)
             C[i][j] = A[i][j] + B[i][j];
    } // End of if block

    else if(key == 'x')
        C = matrixMult(A, B);

    else
        System.out.println("Error! Enter '+' or 'x'!");
```

$O(n^2)$

$O(n^3)$

$O(1)$

Overall complexity **$O(n^3)$**

How to determine complexity of code structures

- Sometimes if-else statements must carefully be checked:

  O(if-else) = O(Condition)+ Max[O(if), O(else)]

```
int[] integers = new int[n];
........
if(hasPrimes(integers) == true)
      integers[0] = 20;
else
      integers[0] = -20;


public boolean hasPrimes(int[] arr) {
      for(int i = 0; i < arr.length; i++)
                ..........
                ..........
} // End of hasPrimes()
```

O(1)

O(1)

O(n)

O(if-else) = O(Condition) = ***O(n)***

Best, Average, and Worst case complexities

- What is the best case complexity analysis?
  - The smallest number of operations carried out by the algorithm for a given input.
- What is the worst case complexity analysis?
  - The largest number of operations carried out by the algorithm for a given input.
- What is the average case complexity analysis?
  - The number of operations carried out by the algorithm on average <u>for all inputs</u>.

$$\sum_{\text{for each input } i} (\text{Probability of input } i * \text{Cost of input } i)$$

- We are usually interested in the **worst case** complexity
  - Easier to compute
  - Represents an upper bound on the actual running time for all inputs
  - Crucial to real-time systems (e.g. air-traffic control)

Best, Average, and Worst case complexities: Example

- For linear search algorithm, searching for a key in an array of n elements, determine the situation and the number of comparisons in each of the following cases
  - Best Case
  - Worst Case
  - Average Case