BIG O-NOTATION

CS A250 – C++ Programming Language 2

INTRODUCTION

- Key factors in designing software:

  - **Reliability:** Your program should anticipate and handle all types of exceptional circumstances.

  - **Flexibility:** Your program should be easy to modify to handle circumstances that may change in the future.

  - **Reusability and expandability:** If your program is successful, it will frequently spawn new computing needs; you should be able to incorporate solutions to these new needs into the original system with relative ease.

INTRODUCTION (CONT.)

- Key factors in designing software:

  - **User-friendliness**: Your program should be clearly documented so that it is easy to use—***both*** internal and external documentation).

  - **Structured organization:** The system should be divided into compact modules, each of which is responsible for a specific, well-defined task.

  - **Efficiency:** The system should make optimal use of **time** and **space** resources.

INTRODUCTION (CONT.)

- As a **computer scientist** you should:

  - Have a detailed knowledge of **algorithms** and **data storage techniques**
    - Do not re-invent the wheel!

  - Apply these techniques when designing software

  - Choose the most appropriate algorithms and tailor them to the application you are creating

4

EFFICIENCY

- A system to be **efficient** needs to make optimal use of

  - Storage space

  - Programming effort

  - Computer time

  - *Let's look at each one of them…*

EFFICIENCY – STORAGE SPACE

- **Storage space** is the amount of memory required to store data
  - If the list is too large to be kept in high-speed memory, you need to find other options
  - One approach for sorting
    - Divide lists into sub-lists
    - Sort them internally within high-speed memory
    - Merge the sorted sub-lists externally

EFFICIENCY – PROGRAMMING EFFORT

- Several factors to consider:
  - An **algorithm** is going to be running only a few times
    - No need for a programmer to spend days/weeks investigating sophisticated algorithms
  - Should you use **recursion**?
    - Consider **readability**
    - Consider if easy to implement
    - Some programming languages do **not** have recursion (FORTRAN, COBOL)
  - **Simplicity** and **correctness** are <u>essential</u>

7

EFFICIENCY – COMPUTER TIME

- One **computer** may be **faster** than another
  - You should use the **same computer** to **test** different algorithms
- Some **languages** are better suited for certain algorithms.
- Some **compilers** generate better machine code than others
- Some **programmers** write better programs than others

8

TIME EFFICIENCY

- Although the efficient use of both **time** and **space** is important, **inexpensive memory** has **reduced** the significance of **space efficiency**. Thus, we will focus primarily on **time efficiency**.

- How can we measure time efficiency?

COMPUTATIONAL COMPLEXITY

- Same problem can be solve with algorithms that differ in efficiency.

- Differences may be immaterial for processing a small number of data items, but can grow with the amount of data.

- **Computational complexity** indicates how costly it is to apply an algorithm:

  - The cost can be measured in a variety of ways.

  - We will focus on the relationship between **input size** and **execution time**.

EXECUTION TIME

- How do we measure the efficiency of algorithms in terms of execution time?
  - Use a systematic and quantitative way to evaluate comparatively
  - Example: Sorting
    - Determine a function $f(n)$, where n is the size of the data
    - How many comparisons to sort the data?

11

Efficiency of Algorithms

- Efficiency can be measured for
  - **Best case**
    - Searching a key in a list:
      - The key is immediately found
  - **Worst case**
    - Searching a key in a list:
      - The key is at the end of the list or there is no key
  - **Average case**
    - Difficult to determine

12

Efficiency of Algorithms (cont.)

- How do we analyze a particular algorithm?
  - We **count the number of operations** the algorithm executes
  - We do **NOT** focus on the actual computer time to execute the algorithm
    - Why not?
      - A particular algorithm can be implemented on a variety of computers and the speed of the computer can affect the execution time
      - **But** the number of operations performed by an algorithm would be the same

EXAMPLE 1

- How many times is each statement executed?

```cpp
cout << "Enter two numbers: ";      // 1 time
cin >> num1 >> num2;

if (num 1 > num2)
     largest = num1;
else
     largest = num2;

cout << "The largest number is"
    << largest << endl;
```

EXAMPLE 1 (CONT.)

- How many times is each statement executed?

```
cout << "Enter two numbers: ";      // 1 time
cin >> num1 >> num2;                 // 1 time

if (num 1 > num2)
     largest = num1;
else
     largest = num2;

cout << "The largest number is"
    << largest << endl;
```

EXAMPLE 1 (CONT.)

○ How many times is each statement executed?

```
cout << "Enter two numbers: ";      // 1 time
cin >> num1 >> num2;                 // 1 time

if (num 1 > num2)              // 1 time
     largest = num1;
else
     largest = num2;

cout << "The largest number is"
    << largest << endl;
```

EXAMPLE 1 (CONT.)

o How many times is each statement executed?

```
cout << "Enter two numbers: ";      // 1 time
cin >> num1 >> num2;                // 1 time

if (num 1 > num2)            // 1 time
    largest = num1;                  // 0 to 1 time
else
    largest = num2;

cout << "The largest number is"
    << largest << endl;
```

EXAMPLE 1 (CONT.)

- How many times is each statement executed?

```
cout << "Enter two numbers: ";      // 1 time
cin >> num1 >> num2;                 // 1 time

if (num 1 > num2)              // 1 time
    largest = num1;                  // 0 to 1 time
else
    largest = num2;                  // 0 to 1 time

cout << "The largest number is"
    << largest << endl;
```

EXAMPLE 1 (CONT.)

- How many times is each statement executed?

```
cout << "Enter two numbers: ";        // 1 time
cin >> num1 >> num2;                   // 1 time

if (num 1 > num2)              // 1 time
     largest = num1;                   // 0 to 1 time
else
     largest = num2;                   // 0 to 1 time

cout << "The largest number is"
   << largest << endl;                 // 1 time
```

19

EXAMPLE 1 (CONT.)

○ How many times is each statement executed?

```
cout << "Enter two numbers: ";      // 1 time
cin >> num1 >> num2;                 // 1 time

if (num 1 > num2)              // 1 time
    largest = num1;                  // 0 to 1 time
else
    largest = num2;                  // 0 to 1 time

cout << "The largest number is"
    << largest << endl;          // 1 time
```

What is the total?

EXAMPLE 1 (CONT.)

- How many times is each statement executed?

```cpp
cout << "Enter two numbers: ";      // 1 time
cin >> num1 >> num2;                // 1 time

if (num 1 > num2)           // 1 time
     largest = num1;                // 0 to 1 time
else
     largest = num2;                // 0 to 1 time

cout << "The largest number is"
   << largest << endl;            // 1 time
```

Total = 1 + 1 + 1 + 1 + 1 = 5

EXAMPLE 2

- How many times is each statement executed?

```
int num = 0;
int count = 0;
int sum = 0;
cout << "Enter 5 integers: ";
while (count < 3)
{
    cin >> num;
    sum += num;
    ++count;
}
cout << "The sum is"
    << sum << endl;
```

EXAMPLE 2 (CONT.)

○ How many times is each statement executed?

```
int num = 0;                        // 1 time
int count = 0;                      // 1 time
int sum = 0;                        // 1 time
cout << "Enter 5 integers: ";       // 1 time
while (count < 3)
{
     cin >> num;
     sum += num;
     ++count;
}
cout << "The sum is"
   << sum << endl;
```

EXAMPLE 2 (CONT.)

- How many times is each statement executed?

```
int num = 0;                          // 1 time
int count = 0;                        // 1 time
int sum = 0;                          // 1 time
cout << "Enter 5 integers: ";      // 1 time
while (count < 3)              // 4 times
{
      cin >> num;
      sum += num;
      ++count;
}
cout << "The sum is"
    << sum << endl;
```

EXAMPLE 2 (CONT.)

- How many times is each statement executed?

```
int num = 0;                           // 1 time
int count = 0;                         // 1 time
int sum = 0;                           // 1 time
cout << "Enter 5 integers: ";      // 1 time
while (count < 3)              // 4 times
{
     cin >> num;              // 3 times
     sum += num;              // 3 times
     ++count;                 // 3 times
}
cout << "The sum is"
    << sum << endl;
```

EXAMPLE 2 (CONT.)

- How many times is each statement executed?

```
int num = 0;                            // 1 time
int count = 0;                          // 1 time
int sum = 0;                            // 1 time
cout << "Enter 5 integers: ";       // 1 time
while (count < 3)                // 4 times
{
     cin >> num;             // 3 times
     sum += num;             // 3 times
     ++count;                // 3 times
}
cout << "The sum is"
    << sum << endl;                    // 1 time
```

EXAMPLE 2 (CONT.)

- How many times is each statement executed?

```
int num = 0;                          // 1 time
int count = 0;                        // 1 time
int sum = 0;                          // 1 time
cout << "Enter 5 integers: ";        // 1 time
while (count < 3)            // 4 times
{
    cin >> num;             // 3 times
    sum += num;             // 3 times
    ++count;                // 3 times
}
cout << "The sum is"
    << sum << endl;                   // 1 time
```

Total = 18

EXAMPLE 2 (CONT.)

- How many times is each statement executed?

```
    int num = 0;                        // 1 time
    int count = 0;                      // 1 time
    int sum = 0;                        // 1 time
    cout << "Enter 5 integers: ";       // 1 time
    while (count < 3)              // 4 times
    {
        cin >> num;            // 3 times
        sum += num;            // 3 times
        ++count;               // 3 times
    }
    cout << "The sum is"
       << sum << endl;                  // 1 time
```

Execution time = 1 + 1 + 1 + 1 + (3 + 1) + (3 + 3 + 3) + 1

EXAMPLE 2 (CONT.)

Assume we do not know that the loop will go around 3 times, and we know that it will go around n times:

```
while (count < n)      // executes n + 1 times
{
        cin >> num;
        sum += num;    // each executes n times
        ++count;
}
```

```
Execution time = 1 + 1 + 1 + 1 +  (n + 1) + (n + n + n) + 1
```

EXAMPLE 2 (CONT.)

We can simplify our expression:

$$1 + 1 + 1 + 1 + (n + 1) + (n + n + n) + 1$$

$$5 + (n + 1) + (n + n + n)$$

$$5 + n + 1 + (n + n + n)$$

$$6 + n + (n + n + n) = 6 + 4n$$

The execution time for this algorithm is $= 6 + 4n$

PRACTICAL SHORTCUTS

- Ignore **lesser terms** (find the **dominant term**)
  - $n^3 + 5n^2 + n$ → n^3

- Ignore **coefficients**
  - $5n^2$ → n^2

- Ignore **bases of logarithms**
  - $\log_{10} n$ → $\log n$
  - $\log_2 n$ → $\log n$ *(same)*

BIG O-NOTATION

- **Big O-notation**:
  - Mathematical measuring tool to quantitatively evaluate algorithms
  - Also called **Big Oh** notation, **Landau** notation, **Bachmann-Landau** notation, and **asymptotic** notation.
  - Formal definition:

$$f(x) = O(g(x)) \ \text{ as } x \rightarrow \infty$$

  - Categorizes algorithms with respect to **execution time**

    → *in terms of **growth rate**, __not__ speed*

BACK TO EXAMPLE 2 (CONT.)

Going back to example 2, we can simplify as follows:

Ignore lesser terms $\rightarrow$ $6 + 4n \equiv 4n$
Ignore coefficients $\rightarrow$ $4n \equiv n$

The algorithm's complexity is O (n)

COMMON GROWTH-RATE FUNCTIONS

- **O(1)** - Constant
  - Time required is constant
  - Independent of problem size
- **O(log n)** - Logarithmic
  - Time requirement increases slowly as the size increases
  - The base does not affect the growth rate
- **O(n)** - Linear
  - Increases directly with size
- **O(n log n)** - Log-linear
  - Time requirement increases more rapidly than a linear algorithm
  - Typical algorithms divide the problem into smaller problem that are solved separately

Common Growth-Rate Functions

- **$O(n^2)$** - Quadratic
  - Time requirement increases rapidly with the size of the problem
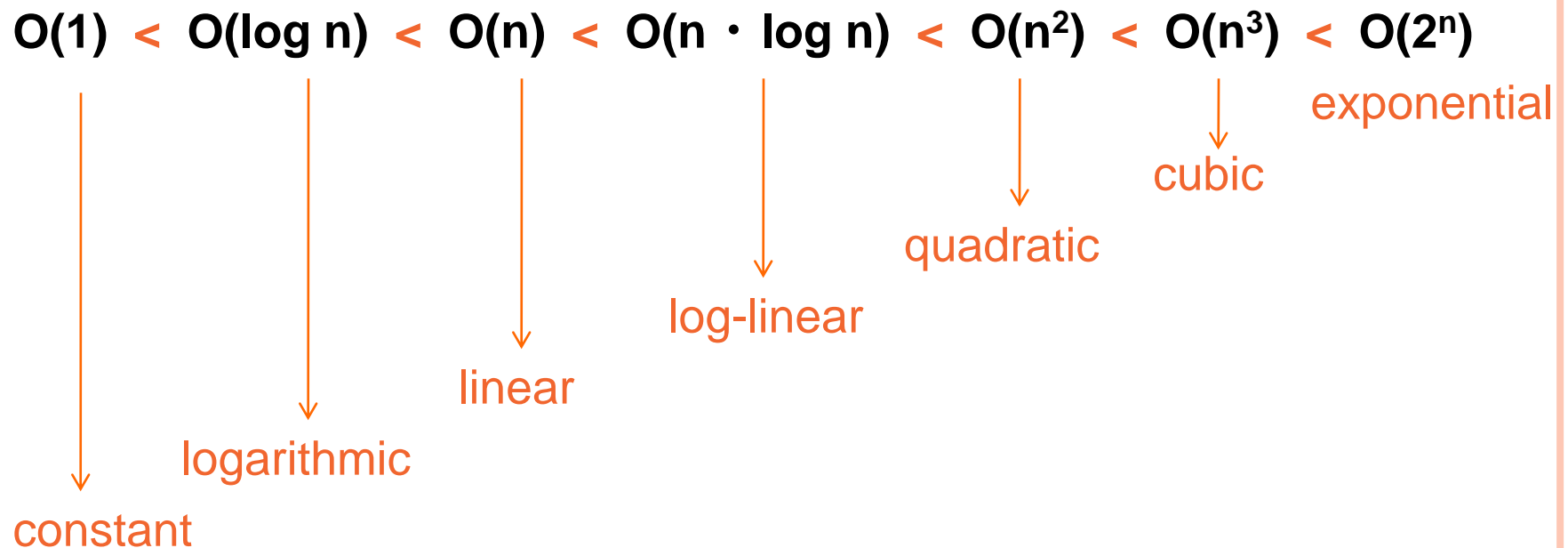 - Practical for small problems only
- **$O(n^3)$** - Cubic
  - Time requirement increases more rapidly than the quadratic algorithm
  - Practical for small problems only
- **$O(2^n)$** - Exponential
  - Time requirement increases too rapidly to be practical

Order of Growth

$O(1)$ $<$ $O(\log n)$ $<$ $O(n)$ $<$ $O(n \cdot \log n)$ $<$ $O(n^2)$ $<$ $O(n^3)$ $<$ $O(2^n)$

exponential

cubic

quadratic

log-linear

linear

logarithmic

constant

REDUCING AN EXPRESSION TO BIG-O

- The number of logical operations in an algorithm as a function of n can be reduced to a simplified **O-notation**:

$$n + 4n^2 + 4n$$

What is the dominant term?   → $4n^2$

What is the O-notation?   → $O(n^2)$

EFFICIENCY OF BINARY SEARCH

- We have seen that **Binary Search** cuts the list in **half** after each **comparison**.

- Assume you have a list of 32 items.

- After comparing the **middle element**, if the middle element is **not** the item we are looking for, only one half of the list will be considered.

Efficiency Of Binary Search (cont.)

- Start with 32 items in the list.
- Roughly, we cut the list in half after each comparison
  - Assume the item we are looking for is not in the list.

32

16

8

4

2

1

Efficiency Of Binary Search (cont.)

- Start with 32 items in the list.
- Roughly, we cut the list in half after each comparison
  - Assume the item we are looking for is not in the list.

32		2^5
16		2^4
8	equivalent to	2^3
4		2^2
2		2^1
1		2^0

EFFICIENCY OF BINARY SEARCH (CONT.)

- Start with 32 items in the list.
- Roughly, we cut the list in half after each comparison
  - Assume the item we are looking for is not in the list.

32	2^5		$\log_2 32 = 5$
16	2^4		$\log_2 16 = 4$
8	2^3	in logarithmic notation	$\log_2 8 = 3$
4	2^2		$\log_2 4 = 2$
2	2^1		$\log_2 2 = 1$
1	2^0		$\log_2 1 = 0$

EFFICIENCY OF BINARY SEARCH (CONT.)

- Start with 32 items in the list.
- Roughly, we cut the list in half after each comparison
  - Assume the item we are looking for is not in the list.

$$\log_2 32 = 5$$

$$\log_2 16 = 4$$

$$\log_2 8 = 3$$

$$\log_2 4 = 2$$

$$\log_2 2 = 1$$

$$\log_2 1 = 0$$

Therefore, the **growth rate** is **logarithmic**:

$$O\ (\log_2 n)$$

But we **omit the base** and we have:

$$O\ (\log n)$$

EFFICIENCY OF BINARY SEARCH (CONT.)

- The **running time** of **binary search** for worst and average case is **O (log n)**

- This is better than **linear search**, which has worst and average case **O (n)**

- BUT, for Binary search to work, we need to assume that all **elements are in order**.

O-Notation (end)

44