ST. CHARLES COMMUNITY COLLEGE

CPT-281 - Introduction to Data Structures with C++

Module 1

Introduction to the Big-O Theory

Dayu Wang

- **Data structures and algorithms are closely-related to each other.**

  → You cannot talk about data structures without talking about algorithms.

  → You cannot talk about algorithms without talking about data structures.

- **What are data structures?**

  → Data structure is the arrangement of data in computer's memory.

  For example, array (vector) is a data structure.

- **What are algorithms?**

  → Algorithms are step-by-step procedures for processing data in data structures.

  For example, accessing, insertion, deletion, searching, sorting, etc.

- **Learning data structures (in this class) includes:**

  → Understanding the mechanisms to organize data in various containers.

  Vector, linked list, stack, queue, binary search tree, heap, map...

  → Using C++ programming language to implement some basic data structures.

  → [Important] Using the data structures learned to solve problems.

- **What kinds of problems data structures can solve?**

→ **[Important] Algorithm problems**

There must be algorithm problems in your job interview for positions of software developer.

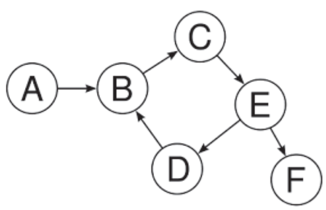
You need to design optimal algorithms, not just correct algorithms, to get the job.

→ **Real-world data storage (system implementation)**

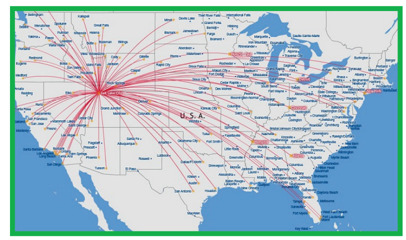
1) **Undo/redo system: history activities should be stored in what kind of data structure?**

2) **Transaction processing system (TPS): what data structure is optimal to store transactions so that insertion, deletion, searching, and sorting can be done fast?**

→ **Real-world modeling**



Graph ↔ Airline routes



- **What is efficiency?**

→ **The use of as few resources as possible to accomplish a goal.**

→ **Algorithm efficiency is measured in terms of its performance (time) and memory usage (space).**

→ **The less time an algorithm takes and the less memory it requires, the better its efficiency.**

- **How can we know the performance of an algorithm?**

→ **We can measure the time the algorithm takes using the CPU clock:**

```
long long unsigned int measure_execution_time() {
    // Record the time before running the program.
    std::clock_t time_start = std::clock();
    // Sample program -> Calculate "1 + 2 + 3 + ... + 1,000,000 = ?"
    unsigned int sum = 0;
    for (unsigned int current = 1; current <= 1000000; current++) {
        sum += current;
    }
    // Record the time after running the program.
    std::clock_t time_end = std::clock();
    // Return the execution time in clock ticks.
    return (long long unsigned int)time_end - (long long unsigned int)time_start;
}
```

- **What is the main problem with this solution?**

  → We measured the actual algorithm execution time.

  → However, the actual algorithm execution time depends on too many factors, not just the quality of the algorithm:

    1) CPU properties
    2) Programming languages
    3) Different compilers
    4) Quality of algorithms

  → We want to rule out "other factors", comparing the quality of algorithms regardless of CPUs, programming languages, and so on.

- **We can get inspired from the example below.**

  → Do you know which vehicle has the smallest size without measuring their actual dimensions?

Mini Van	Jeep	Compact	SUV	Intermediate	Standard

Vehicles are categorized into groups, and groups are comparable.

We actually compare groups instead of comparing specific vehicles.

- **The Big-O Theory**

  → Idea of Big-O theory is to categorize algorithms into different groups.

  Groups are comparable; one group is better than another.

  → To compare two algorithms (that can solve the same problem), people can identify which groups the algorithms belong to.

    1) If they belong to different groups, then the algorithm residing in a "better" group is better.
    2) If they belong to the same group, then the two algorithms have no big difference in quality.

  → The Big-O theory provides a useful way to evaluate the performance of algorithms only based on the quality of the algorithms.

- **How to analyze the performance of an algorithm?**

  [Fact] If the array is larger, then it needs more time to process data.

  → Algorithm performance is analyzed based on the size of input.

  Size of input is normally denoted as n.

    1) If input is an array/string, then n is the length of the array/string.
    2) If input is an integer, then n is the magnitude of the integer.

  [Important] Assume n is very large ($n \to \infty$).

- **How many times the statement will be executed?**

→ **[Example 1]**

```
for (int i = 0; i < 4; i++) {
    /* Statement */
}
```

- **The statement is executed when `i == 0`, `i == 1`, `i == 2` and `i == 3`.**
- **The statement will be executed 4 times.**
- **What will be the answer if I change the "4" to "10", to "100", and to "30000"?**

```
for (int i = 0; i < n; i++) {
    /* Statement */
}
```

- **The statement will be executed n times.**
- $T(n) = n$
- $T(n)$ **is a time function.**
- $T(n)$ **is a function of the size of input.**
- **In this example,** $T(n)$ **and** n **has a proportional relationship with coefficient 1.**
- $T(n)$ **is not Big-O (we will discuss Big-O later).**

- **How many times the statement will be executed?**

→ **[Example 2]**

```
for (int i = 0; i < 3 * n; i++) {
    /* Statement */
}
```

- **If `n == 1`, then `3 * n == 3`, the statement will be executed 3 times.**
- **If `n == 2`, then `3 * n == 6`, the statement will be executed 6 times.**
- **If `n == 3`, then `3 * n == 9`, the statement will be executed 9 times.**

  ...
- $T(n) = 3n$
- **In this example,** $T(n)$ **and** n **has a proportional relationship with coefficient 3.**

```
for (int i = 0; i < n; i++) {
    /* Statement */
    /* Statement */
    /* Statement */
}
```

- $T(n) = 3n$ **(3 statements in each iteration, `n` iterations)**

- **How many times the statement will be executed?**

→ **[Example 3]**

```
for (int i = 0; i < 3; i++) {
    for (int j = 0; j < 4; j++) {
        /* Statement */
    }
}
```

- **When `i == 0`, `j == 0`, 1, 2, and 3, statement will be executed 4 times.**
- **When `i == 1`, `j == 0`, 1, 2, and 3, statement will be executed 4 times.**
- **When `i == 2`, `j == 0`, 1, 2, and 3, statement will be executed 4 times.**
- **Totally, the statement will be executed 12 times ($3 \times 4 = 12$).**

```
for (int i = 0; i < m; i++) {
    for (int j = 0; j < n; j++) {
        /* Statement */
    }
}
```

- $T(m, n) = mn$
- **[Tip] Consider multiplication for nested loops.**

- **How many times the statement will be executed?**

→ **[Example 4]**

```
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        /* Statement */
    }
}
```

- **Using the "multiplication tip", $T(n) = n^2$.**
- **In this example, $T(n)$ and n has a quadratic relationship (not linear).**

→ **[Exercise] How many times the statement will be executed?**

A

```
for (int i = 3; i < n; i++) {
    for (int j = 5; j < n; j++) { /* Statement */ }
}
```

B

```
for (int i = 3; i < n; i++) {
    for (int j = 5; j < n; j++) {
        for (int k = 4; k < n; k++) { /* Statement */ }
    }
}
```

- **How many times the statement will be executed?**

→ **[Example 5]**

```
for (int i = 0; i < n; i++) {
    for (int j = i + 1; j < n; j++) {
        /* Statement */
    }
}
```

- **When `i == 0`, the inner loop is `(j = 1; j < n; j++)`, the statement will be executed n – 1 times.**
- **When `i == 1`, the inner loop is `(j = 2; j < n; j++)`, the statement will be executed n – 2 times.**
- **When `i == 2`, the inner loop is `(j = 3; j < n; j++)`, the statement will be executed n – 3 times.**

  ...
- **When `i == n – 2`, the inner loop is `(j = n – 1; j < n; j++)`, the statement will be executed 1 time.**
- **When `i == n – 1` (largest value of `i`), the inner loop is `(j = n; j < n; j++)`, the statement will be executed 0 times.**
- $T(n) = (n-1) + (n-2) + (n-3) + \cdots + 3 + 2 + 1 + 0$

- $T(n) = (n-1) + (n-2) + (n-3) + \cdots + 3 + 2 + 1 + 0$

→ **What is the explicit expression of $T(n)$?**

- **The story of Gauss**

→ **First of all, the "+ 0" at the end of $T(n)$ can be ignored.**

$$T(n) = (n-1) + (n-2) + (n-3) + \cdots + 3 + 2 + 1$$

→ **Then, let's write $T(n)$ twice, one in forwards, one in backwards.**

$$T(n) = (n-1) + (n-2) + (n-3) + \cdots + 3 + 2 + 1$$

$$T(n) = 1 + 2 + 3 + \cdots + (n-3) + (n-2) + (n-1)$$

→ **Then, add the two equations (blue and red) together, meaning add the left side together and add the right side together.**

$$2T(n) = \underbrace{n + n + n + \cdots + n + n + n}$$

There are n – 1 "n"s adding together.

→ **Finally, using the definition of multiplication to get $T(n)$.**

$$2T(n) = n(n-1)$$

$$T(n) = \frac{n(n-1)}{2} = \frac{1}{2}n^2 - \frac{1}{2}n$$ **(still a quadratic relationship)**

- **How many times the statement will be executed?**

→ **[Example 6]**

```
for (int i = 0; i < n; i++) {
    for (int j = n - 1; j >= 1; j--) {
        /* Statement */
        /* Statement */
    }
}

for (int k = 2; k < n - 2; k++) {
    /* Statement */
}

/* Statement */

for (int c = 32; c < 64; c += 2) {
    /* Statement */
}
```

$T_1(n) = 2n(n-1)$

$T_2(n) = n - 4$

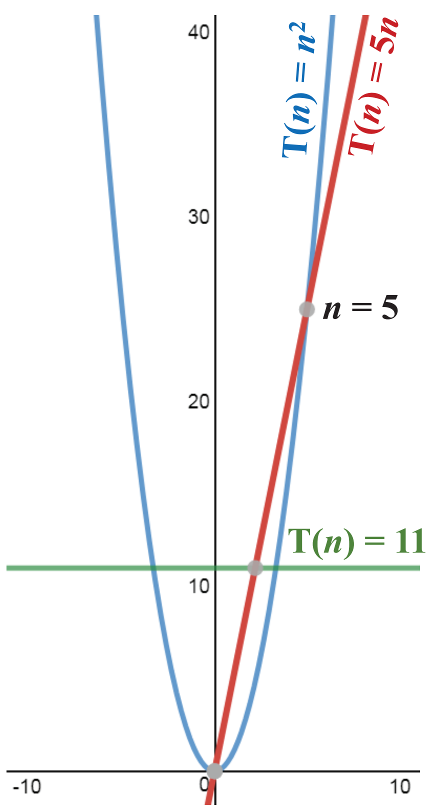
$T_3(n) = 1$

$T_4(n) = \frac{64-32}{2} = 16$

- $T(n) = T_1(n) + T_2(n) + T_3(n) + T_4(n)$
  $= 2n(n-1) + (n-4) + 1 + 16$
  $= 2n^2 - n + 13$ **(a polynomial in terms of n)**

- **How to use Big-O to categorize different algorithms?**

→ **[Example]** $T(n) = n^2 + 5n + 11$



- **For $n < 5$, $5n$ increases faster than n^2.**
- **For $n > 5$, n^2 increases faster than $5n$.**
- **In the Big-O notation, we always assume that n is very large ($n \to \infty$).**
- **When n is very large, the time of execution is mainly determined by term n^2.**
- **We say that n^2 is the dominating term that determines the time of execution.**
- **Big-O notation:** $T(n) = O(n^2)$

→ **Mathematical definition of Big-O:**

There exists two positive constants, n_0 and c, and a function $f(n)$, such that $cf(n) \geq T(n)$ is true for all $n > n_0$, then $T(n) = O(f(n))$.

→ **In this example ($T(n) = n^2 + 5n + 11$), to prove that $T(n) = O(n^2)$, simply let $f(n) = n^2$, $c = 3$, and $n_0 = 100$.**

- **Properties of the Big-O Notation**

→ **Only the highest order term dominates (other terms can be ignored).**

$$O(n^2 + 3000000000n + 1000000000000000000000000000000000) = O(n^2)$$

→ **Constant coefficient should be ignored.**

$$O(100n^2) = O(n^2)$$

$$O(0.0000001n^2) = O(n^2)$$

→ **[Exercise] Identify the higher-order term.**

1) $3n$ and $\frac{3}{2}\sqrt{n}$

2) $2\sqrt{n}$ and $3\log n$

3) 2π and $\sqrt{2}e$

→ **[Exercise] Write the $T(n)$ and Big-O for the following piece of code:**

```
for (int j = n; j > 0; j /= 2) {
    /* Statement */
}
```

C

- **In computer science, if the base of logarithm does not matter, the base can be omitted.**

- **Comparison of Performance (from Quickest to Slowest)**

Big-O Notation	Name	$f(50)$	$f(100)$
$O(1)$	Constant	1	1
$O(\log n)$	Logarithmic	5.64	6.64
$O(n)$	Linear	50	100
$O(n\log n)$	Log-Linear	282	664
$O(n^2)$	Quadratic	2500	10,000
$O(n^3)$	Cubic	12,500	100,000
$O(2^n)$	Exponential	1.126×10^{15}	1.27×10^{30}
$O(n!)$	Factorial	3.0×10^{64}	9.3×10^{157}

→ **Comparing algorithms**

1) **Find the Big-O of each algorithm.**
2) **The algorithm that has the quickest Big-O is the best.**
3) **If two (or more) algorithms have the same Big-O, then there is no big difference in the quality of the algorithms.**

- **Best Case and Worst Case**

→ **Sometimes, the Big-O of an algorithm depends on the input data:**

```
// Linear search
int search(const vector<int>& arr, int target) {
    for (int i = 0; i < arr.size(); i++) {
        if (arr.at(i) == target) { return i; }
    }
    return -1;
}
```

What is the Big-O of linear search?

1) **Best case (the first element is the target):** $O(1)$

2) **Worst case (the last element is the target):** $O(n)$

→ **By default, the Big-O of an algorithm is always the worst case Big-O.**

- **Average Case**

→ **Average case of $T(n)$ considers each possibility with its probability.**

- **Average case Big-O of linear search:**

$$T(n) = 1 \times \frac{1}{n} + 2 \times \frac{1}{n} + 3 \times \frac{1}{n} + \cdots + n \times \frac{1}{n}$$

$$= \frac{1}{n}(1 + 2 + 3 + \cdots + n)$$

$$= \frac{1}{n} \cdot \frac{n(n+1)}{2} = \frac{n+1}{2} = \frac{1}{2}n + \frac{1}{2} = O(n)$$

→ **Time complexity of linear search:**

1) **Best case:** $O(1)$

2) **Worst case:** $O(n)$

3) **Average case:** $O(n)$