Tutorial 06: Complexity Analysis

CSCI2520 - DATA STRUCTURES AND APPLICATIONS

TUTOR: ZHENG CHENGUANG

Outlines

- Big-O Notation
- Big-O Simplification Rules
- Big-O Formal Definition
- Determining Computational Complexity from Code Structure
- Binary Search

Big-O Notation

- The relationship between the problem size N and the performance of an algorithm as N becomes large is called the ***computational complexity*** (or time complexity) of the algorithm.
- The ***big-O notation*** is used to provide a *quantitative insight* as to how changes in the problem size N affect the algorithmic performance as N becomes *large*.

Big-O Simplification Rules

- *Eliminate any term whose contribution to the total becomes **in**significant as* N *becomes large.*
  - When a formula involves a summation of several terms, the *fastest growing term* alone will *control* the running time of the algorithm for large N.
  - Example: $N^3 + 1000N^2 + N = O(N^3)$
- *Eliminate any **constant factors**.*
  - What we want to capture in computational complexity is how changes in N affect the algorithmic performance.
  - Example: $0.1N^3 = O(N^3)$

Exercise 1

- Order the following functions by ascending order of growth rates.
  1. $0.0001N^3 + N$
  2. $2N^{1/2} + \log N + 1$
  3. $3N^{3/2} - 2N^{1/2}$
  4. $N + 5\log N + 100$
  5. $2N^2 + 9N\log N + 1$

Exercise 1 Solution

- Order the following functions by ascending order of growth rates.
  - $2N^{1/2} + \log N + 1 = O(N^{1/2})$
  - $N + 5\log N + 100 = O(N)$
  - $3N^{3/2} - 2N^{1/2} = O(N^{3/2})$
  - $2N^2 + 9N\log N + 1 = O(N^2)$
  - $0.0001N^3 + N = O(N^3)$

  The answer is 2 4 3 5 1

Big-O Formal Definition

- Definition: $t(N) = O(f(N))$ if and only if
  - there are positive constants n_0 and c
  - for every value of $N \geq n_0$, the following condition holds:

$$t(N) \leq c \times f(N)$$

- As long as N is "*large enough*", $t(N)$ is always bounded by a constant multiple of $f(N)$.

Exercise 2

- What are the c to make $t(N) \leq c \times f(N)$ when $n_0 = 1$?
  - $2N^{1/2} + \log N + 1 = O(N^{1/2})$
  - $N + 5\log N + 100 = O(N)$
  - $3N^{3/2} - 2N^{1/2} = O(N^{3/2})$
  - $2N^2 + 9N\log N + 1 = O(N^2)$
  - $0.0001N^3 + N = O(N^3)$

Exercise 2 Solution

- What are the c to make $t(N) \le c \times f(N)$ when $n_0 = 1$?
  - $2N^{1/2} + \log N + 1 \le 2N^{1/2} + N^{1/2} + N^{1/2} = 4 \times N^{1/2}$
  - $N + 5\log N + 100 \le N + 5N + 100N = 106 \times N$
  - $3N^{3/2} - 2N^{1/2} \le 3 \times N^{3/2}$
  - $2N^2 + 9N\log N + 1 \le 2N^2 + 9N^2 + N^2 = 12 \times N^2$
  - $0.0001N^3 + N \le 0.0001N^3 + N^3 = 1.0001 \times N^3$
- Do you still remember big-O simplification rules?
- Using larger n_0, c can be reduced. The requirement of $n_0 = 1$ here is just for illustration.

Determining Computational Complexity from Code Structure

- In general, we can determine the time complexity simply by finding the piece of the code that is executed ***most often***.
- However, if an expression or statement involves ***function calls***, it must be accounted *separately*.

Exercise 3 Binary Search

- A binary search algorithm finds the position of a target value within a ***sorted*** array.
- Compare the target value to the value of the ***middle*** element of the sorted array.
  - If the target value is equal to the middle element's value, then the position is returned and the search is finished.
  - If the target value is less than the middle element's value, then the search continues on the lower half of the array
  - If the target value is greater than the middle element's value, then the search continues on the upper half of the array.
  - In each iteration, half of the elements are eliminated.

Exercise 3 Binary Search

- Input
  - target value: 10
  - input array: (1 2 3 4 5 6 7 8 10 11 12 13 24 25 26)
- First iteration:
  - (1 2 3 4 5 6 7 **8** <u>10 11 12 13 24 25 26)</u>
  - go to upper half since 10 > 8

Exercise 3 Binary Search

- Second iteration:
  – (10 11 12 **13** 24 25 26)
  – go to lower half since 10 < 13
- Third iteration:
  – (10 **11** 12)
  – go to lower half since 10 < 11
- Fourth iteration:
  – (**10**) get it!

Exercise 3 Binary Search

- Finish the implementation of the following function.

```
int BinarySearch(int array[], int n, int tarVal)
```

- Hints
  - You may try to implement the following function first.
  - What if the array is of even length (e.g. 2)? What should be the end case?

```
int BinarySearch(int array[], int tarVal, int iMin, int iMax)
```

- Can you think of this problem recursively?
- Question: what happens if there is no target value?

Exercise 3 Binary Search

```
int BinarySearch(int array[], int tarVal, int iMin, int iMax){
    if (iMin > iMax)
        return -1;
    int iMid = (iMin + iMax) / 2;
    if (array[iMid] == tarVal)
        return iMid;
    else{
        if (array[iMid] > tarVal) iMax = iMid - 1;
        else iMin = iMid + 1;
        return BinarySearch(array, tarVal, iMin, iMax);
    }
}
int BinarySearch(int array[], int n, int tarVal){
    return BinarySearch(array, tarVal, 0, n - 1);
}
```

Two *tests* to stop or continue

Two *end cases* to terminate the recursion

A *recursive call* to continue recursion

Exercise 3 Binary Search

- What is the complexity of binary search in worst case?
  - Hint: what are worst cases?

```
int BinarySearch(int array[], int tarVal, int iMin, int iMax){
    if (iMin > iMax) return -1;
    int iMid = (iMin + iMax) / 2;
    if (array[iMid] == tarVal) return iMid;
    else{
        if (array[iMid] > tarVal) iMax = iMid - 1;
        else iMin = iMid + 1;
        return BinarySearch(array, tarVal, iMin, iMax);
    }
}
```

O(1)

O(?)

Exercise 3 Binary Search

- The time spend at each level is O(1).
- There are totally k levels where $2^k = N$, i.e.,

$$k = \log_2 N$$

- Hence, the time complexity of binary search in worst case is

$$O(\log_2 N)$$

- We usually omit the logarithmic base in writing complexities. That is, we usually write:

$$O(logN)$$

Exercise 4

- What is the time complexity of following functions?

```
void function1(String s) {
        for (int i = 0; i < StringLength(s); ++i) {
                ChangeToDigit(s[i]);
         }
}
// the time complexity of ChangeToDigit() is O(1).
// the time complexity of StringLength() is O(n), where n is the length of the String.
```

$$\mathrm{O}(N^2)$$

Exercise 4

- What is the time complexity of following functions?

```
int function2(int x) {
        if (x == 1 || x == 0) return 1;
        else
          return function2(x / 2) + function2(x / 2);
}
```

$$\mathrm{T(n)} = 2\mathrm{T}\left(\frac{\mathrm{n}}{2}\right) = 2^2 T\left(\frac{n}{2^2}\right) = \ \ldots. = 2^k T(1) \ \ (k = \log_2 n)$$

$$O(n)$$