**Introduction to Algorithm Analysis**

Dr. Byun

12/23/2023

When we talk about "algorithm analysis", we generally refer to two efficiencies of **time efficiency** and **space efficiency**.

Time efficiency, also called time complexity, indicates how fast an algorithm in question runs. Space efficiency, also called space complexity, refers to the amount of memory units required by the algorithm in addition to the space needed for its input and output. In the past, space efficiency was an important topic in algorithm analysis. However, because memory prices have become cheap, most computer scientists **focus on time efficiency**.

In time efficiency, we usually say, "This program is faster than others." Therefore, there must be "**criteria for how fast**". To do this, we use the concept of "**Orders of Growth**" or categories of time complexities.

Refer to the following two sample programs.

Table 1. Two sample programs in C++.

| Program 1 | Program 2 |
| --- | --- |
| 1. #include <iostream> 2. using namespace std; 3. int main() { 4. int i, n; 5. cout << "Enter a number: "; 6. cin >> n; 7. for (i=0; i < n; i++) { 8. cout << "\*"; 9. } 10. cout << "\n"; 11. return 0; 12. } | 1. #include <iostream> 2. using namespace std; 3. int main() { 4. int i, j, n; 5. cout << "Enter a number: "; 6. cin >> n; 7. for (i=0; i < n; i++) { 8. for (j=0; j < n; j++) { 9. cout << "\*"; 10. } 11. cout << "\n"; 12. } 13. return 0; 14. } |

The first program reads a positive integer number *n* from a user and prints the \* symbol *n* times horizontally. The second program prints the \* symbol *n2* times in two dimensions. For example, the programs display the following results for the input number *5*:

| Program 1’s result for input number 5 | Program 2’s result for input number 5 |
| --- | --- |
| \*\*\*\*\* | \*\*\*\*\*  \*\*\*\*\*  \*\*\*\*\*  \*\*\*\*\*  \*\*\*\*\* |

The question here is which program is faster? Running both programs in the same environment on the same computer will obviously make Program 1 much faster. If the input value is as small as 5, the difference between the two programs is small, but if the input value is 1000 or 10,000 or more, the execution time difference between the two programs becomes clear.

More formally, we say that the time efficiency of Program 1 is linear (= *n*) time complexity and Program 2 has quadratic (= *n2*) time complexity. Based on this observation, we can say that the **algorithm analysis is simply to identify the category of time complexity** for the execution time of an algorithm. Generally, **algorithm analysis uses eight popular time efficiency classes** as shown below.

Table 2. Common time efficiency classes.

| Name | Sample Function | Example Algorithms |
| --- | --- | --- |
| Constant time | 1 | Assuming that we know the size of an array, adding a number to the end of the array will take constant time regardless of the size of the array. |
| Logarithmic time | log n | Binary search in a sorted array with n numbers. |
| Linear time | n | Calculate the sum of n numbers in an array. |
| Linearithmic time | n\*log n | Merge sort |
| Quadratic time | n2 | Bubble sort |
| Cubic time | n3 | Multiplication of two n×n matrices |
| Exponential time | 2n | Tower of Hanoi problem with n disks |
| Factorial time | n! | TSP (Traveling Salesman Problem) using brute-force approach |

Here, the time complexity at the top of the table is faster than at the bottom of the table. So, if your program takes linear time and your classmate’s program takes quadratic time, then you can say that your program is faster than your classmate’s one. Because the eight categories are important, you must **remember the sequence of them** for this course. By the way, let's look briefly at why the time efficiency of Program 1 in Table 1 is linear. As you can see, the *cout* statement in the line number 10 is in a loop that executes *n* times. Similarly, the line number 11 in Program 2 is inside a nested loop. Therefore, the time complexity of the program is quadratic.

Meanwhile, when we **express the time complexity** of a program, we typically use **Big-Oh (O), Big-Theta (), or Big-Omega ()** notations. So, we can represent Program 1’s time complexity as O(n), (n), or (n). These notations will be discussed in the class in detail later.

**Basic Operation**

When computer scientists analyze the time efficiency of algorithms, they typically use two different approaches:

* **Empirical analysis**: After implementing an algorithm in a programming language, run the program to measure the actual computing time spent.
* **Theoretical analysis**: Using the theoretical analysis framework, computer scientists analyze an algorithm to find its time complexity.

Because there are many issues measuring a real program on a computer, this class uses theoretical analysis. For the theoretical analysis, you must first identify the “**basic operation”** from pseudocode. A basic operation is **the operation contributing the most to the total running time**. Once you identify it, you will count the number of times the basic operation is executed. As an example, let’s consider the following pseudocode which adds numbers in an array.

1. Algorithm *SequentialAdd* (A[0..n-1])
2. // Input: An array A with *n* numbers from the index 0 to n-1
3. // Output: Sum of the numbers in the array A
4. sum ← 0 // assignment the value 0 to the variable “sum”
5. i ← 0 // assignment the value 0 to the variable “i”
6. while ( i < n ) do
7. sum ← sum + A[i]
8. i ← i + 1
9. return sum

There are four operations in the pseudocode, the comparison operation (<) in line number 6, the addition operation (+) in line numbers 7 and 8, and the return operation in line number 9. To simplify our discussion of time complexity in the class, we **exclude the assignment operation** (←) in line numbers 4, 5, 7, and 8.

From the definition of the basic operation (= the operation contributing the most to the total running time), we should identify **which operation runs most frequently among all operations in a pseudocode**. To identify it in the sample pseudocode, we count the actual execution number of each operation for the array sizes 1, 2, and general number *n*.

Table 3. Number of operations executed in pseudocode *SequentialAdd*.

| Operation | Line # | Array size n = 1 | Array size n = 2 | Array size = *n* |
| --- | --- | --- | --- | --- |
| < | 6 | 2 | 3 | n+1 |
| + | 7 | 1 | 2 | n |
| + | 8 | 1 | 2 | n |
| return | 9 | 1 | 1 | 1 |

As you can see in Table 3, the comparison operation (<) in line number 6 is performed twice if the input array size is 1. Meanwhile, the addition and return operations in the line number 7, 8, and 9 are performed only once for the array size 1. Based on the results, we can say that **the comparison operation (<) in line number 6 is the basic operation of the whole pseudocode** because it executes n+1 (= most frequent) for the general number *n*.

Understanding the concept of "basic operation" in algorithmic analysis is very important throughout the whole semester. So you should know the exact meaning. Here is a summary of the “basic operation”:

1. "Operation" and "Basic Operation" are different concepts. "Operation" means any operation in pseudocode such as +, -, \*, /,%, AND, OR, NOT,>, <, =, and so on. Thus, the pseudocode *SequentialAdd* has four operations (<, +, +, and return). "Basic operation" is the most frequently performed operation among all operations in a pseudocode (= algorithm). Therefore, there is only one basic operation in the pseudocode. Basic operation of the pseudocode *SequentialAdd* is the comparison operation (<) in line number 6.
2. In the class, we exclude the assignment operation (←) of pseudocode. Therefore, it should not be considered a basic operation in the algorithms covered in the class.

Meanwhile, when we analyze the time complexity of an algorithm, we consider the general number ***n* to be a very large number** such as 5000, 10,000, 1 million or more. Thus, the comparison operation (<) of line number 6 and the addition operation (+) of line numbers 7 and 8 have a difference of only 1 and the difference is very small when compared with a large number *n*. For this reason, we can say that the basic operation of the pseudocode is the addition operation in line number 7 or 8 instead of the comparison operation in line number 6.

**Exercise**

For the following pseudocode *Display\_2D\_Stars*, identify the basic operation.

1. Algorithm Display\_2D\_Stars (n)
2. // Input: A positive integer number n
3. // Output: Display the \* symbol in two dimensions.
4. i ← 0
5. while ( i < n ) do
6. j ← 0
7. while ( j < n ) do
8. write '\*'
9. j ← j + 1
10. write '\n'
11. i ← i + 1
12. return 0

**Solution**

There are seven operations (<, <, write, +, write, +, and return) in the pseudocode. Among them, the < operation in the line number 7 is the basic operation. However, we can say that the write operation or + operation in the line number 8 and 9 can be the basic operation because the execution difference is small compared to the < operation in the line number 7.

**Worst-Case, Best-Case, and Average-Case Efficiencies**

The sample pseudocodes we have studied in the document (= *SequentialAdd* and *Display\_2D\_Stars*) have time complexity independent of the input values. However, some algorithms have different execution times depending on the characteristics of the particular input. For example, consider the sequential search algorithm below.

1. Algorithm *SequentialSearch*(A[0..n − 1], K)
2. // Searches for a key value K in an array using sequential search
3. // Input: An array A[0..n − 1] and a search key K
4. // Output: The index of the first element in A that matches K
5. // or −1 if there are no matching elements
6. i ←0
7. while ((i < n) AND (A[i] K)) do
8. i ←i + 1
9. if i < n
10. return i
11. else
12. return −1

The algorithm has total seven operations:

* Comparison operation (<) in the line number 7
* Comparison operation () in the line number 7
* AND operation in the line number 7
* Addition operation (+) in the line number 8
* Comparison operation (<) in the line number 9
* Return operation in the line number 10
* Return operation in the line number 12

Of these, any operation in line number 7 or the addition operation in line number 8 can be the basic operation. Now, let’s choose the comparison operation () of line number 7 the basic operation. Clearly, the running time of this algorithm can be quite different depending on the K value and input array A.

In the **worst case**, when there are no matching numbers or the first matching number happens to be the last one on the array, the algorithm makes the largest number of key comparisons. One example of this case will be K is 55 and array A has {100, 5, 89, 70, 20, 64, 77} in which the array size *n* is 7.

The **best case** input for sequential search is a list with the first number being the search key *K*. For example, if K is 100 and array A has {100, 5, 89, 70, 20, 64, 77}, the comparison operation () will be executed only 1 (or constant time).

As in the pseudocode, some algorithms need to analyze execution time by taking into account the best and worst cases depending on the nature of the input. The **worst-case efficiency** of an algorithm is its efficiency for the worst-case input of size n. The worst-case analysis provides very important information about an algorithm’s efficiency by bounding its running time from above. In other words, it guarantees that for any instance of size n, the running time will not exceed the worst-case running time. The **best-case efficiency** of the algorithm is the efficiency for the best input of size n.

Meanwhile, there is one more analysis called the **average-case efficiency** which finds the time efficiency for "typical" or "random" inputs. **The average-case efficiency cannot be obtained by taking the average of the worst-case and the best-case efficiencies.** Typically, investigation of the average-case efficiency is considerably more difficult than investigation of the worst-case and best-case efficiencies.Thus, the average-case efficiency is not our main concern in this class, and we will mostly quote known results about the average-case efficiency of algorithms under discussion.

**Pseudo-code analysis with "for" loops in textbooks**

Before we finish this document, I want to talk about the analysis of pseudocode with a for loop. The following example calculates an average of *n* numbers in an array.

1. Algorithm *Average* (A[0..n-1])
2. // Input: An array A with *n* numbers from the index 0 to n-1
3. // Output: Average of the numbers in the array A
4. sum ← A[0]
5. for i ← 1 to n - 1 do
6. sum ← sum + A[i]
7. avg ← sum / n
8. return avg

This algorithm uses a for loop to get the sum of the array A. The for loops used in our textbook differ from common programming languages. For example, the C ++ language uses a for loop with a comparison operation (<) and an index increment operation (++) as shown below

for (int i = 1; i < n - 1; i++)

However, the “for” loop used in the pseudocode simply shows the index and its range as shown below

for i ← 1 to n - 1 do

Therefore, when you analyze a pseudocode using a for loop in the class, you must **assume that there is only one operation in the for loop**. So, the pseudocode *Average* has four operations: for loop operation in line number 5, addition operation (+) in line number 6, division operation (/) in line number 7, and return operation in line number 8. Among them, either the for loop operation or the addition operation can be the basic operation of the pseudocode.