Algorithm Analysis

Contents

Introduction

An ***algorithm*** is a set of instructions to be followed (usually by a computer) to solve a computational problem. By computational problem, we mean a problem for which a solution has been designed and structured in the form of an algorithm which could be implemented using a suitable programming language, and processed by a given computer.

Generally there can be more than one algorithm to solve a given computational problem and these algorithms can be implemented using different programming languages on different platforms. In order to cross-compare different algorithms, we need to analyse each of them in terms of their ***space*** and ***time complexities***. *Space complexity* refers to the theoretical evaluation of how much space (memory load) is required by the algorithm if processed by the computer, and *time complexity* refers to the time (processing time) taken by the algorithm to complete (i.e. to solve the problem).

The aim of ***algorithm analysis*** is thus to assess the efficiency of an algorithm. Algorithm efficiency entails not only the time required by the computer to process the corresponding program, but also the memory resources required during its execution. However, a formal algorithm analysis is usually performed theoretically without taking into account the underlying architecture on which the corresponding program will be executed. This kind of formalism helps evaluate the performance of the algorithm at design time before taking into account programming language and hardware considerations.

Algorithm analysis is therefore a means by which the programmer may evaluate various algorithms aiming at solving the same problem in order to choose the optimal solution. An algorithm is said to be **optimal** if it is processed ***faster*** (*time complexity*) compared to its counterparts, and utilizes fewer memory resources (*space complexity*) compared to the others.

Once we have an algorithm that correctly solves a problem, both its time and space complexities should be evaluated in order to capture its efficiency compared to other algorithms designed to solve the same problem.

This lesson focuses on how to estimate the time and space complexities (requirements) of an algorithm.

Execution time of algorithms

One approach to compare the time efficiency of two algorithms that solve the same problem is to implement these algorithms in a programming language and run them to compare their time requirements.

The difficulties with this approach are that it is dependent upon the computer used, the programming language, the programmer's style, and the test data used. When we analyze algorithms, we should employ mathematical techniques that analyze algorithms independently of *specific implementations, computers, or data.*

To analyze algorithms we first count the number of **basic operations** in a particular solution to assess its efficiency.

Then, we will express the efficiency of algorithms using growth functions. **A basic operation** is an operation that takes a one-time unit to execute and is independent of the programming language used. One unit of time is simply the theoretical time taken by a basic operation to complete on a given computer. In practice, this time may vary from one computer to another, according to the performance of the processor.

An algorithm usually consists of basic operations that can be executed by the computer. Basic operations include among others:

- Assignment operations (e.g.: today = "Monday")
- Arithmetic operations (e.g.: salary = hoursWorked*hourlyRate)
- Comparison operations (e.g.: age == 20)

Some simple examples

Example 1: A single operation

```
count = count + 1;
```

takes one unit of time to complete. Its evaluation depends on the unit time (in micro/ milliseconds) it takes the computer to execute a basic operation.

If for a given computer, a unit time is for example 1 microsecond, then approximately 1 microsecond is required to complete the operation.

Example 2: A sequence of operations:

count = count + 1;(1) (Cost = 1 unit time)

sum = sum + count;(2) (Cost = 1 unit time)

Total cost = 1 + 1 = 2

Instructions (1) and (2) are both basic operations. As for the first example, it will take approximately 2 unit times to complete the sequence.

Example 3: A Simple If-Statement

Operation	**Cost**
if (n < 0)	1
absval = -n	1
else	
absval = n;	1

Cost

Total cost = 1+ 1 = 2

For this simple conditional statement, only one branch is taken after the condition has been evaluated.

Example 4: A Simple Loop

	Cost
i = 1;	1
sum = 0;	1
while (i <= n) {.	n+1
i = i + 1;	n
sum = sum + i;	n

Total Cost = 1 + 1 + (n+1) + n + n

The time required for this algorithm is 3n + 3

Remarks

(a) Loops: The running time of a loop is at most the running time of the statements inside that loop times the number of iterations.

(b) Nested Loops: The running time of a nested loop containing a statement in the innermost loop is the running time of the statements multiplied by the product of the size of all loops.

(c) Consecutive Statements: The running time is the sum of the running time of each statement.

(d) If/Else: The running time is that of the test instruction plus the larger of the running times of both branches' instructions.

In many cases, we can isolate a specific operation fundamental to the analysis of the algorithm, and then ignore other basic operations (that have a negligible influence on the overall time complexity or are the same for all algorithms being considered to solve a particular problem). Examples of such ignorable operations are initialization and incrementation of loop control variables. The chosen basic operation may, for instance, be very expensive compared to the others, or it may be the only operation that causes a growth of time required. So then we only count the chosen basic operations. Of course, we need to be careful to make the right choice of basic operations. In our simple loop of example 4, we can safely choose sum =sum + i as our basic operation, and use only this statement in our running time calculations.

Order-of-Magnitude Analysis and Big-O Notation

Normally, we measure an algorithm's time requirement as a function of its *problem size*. Problem size depends on the application. For example in a sorting algorithm, the problem size will be the number of elements we want to sort. The problem size for an algorithm that calculates the *nth* prime number will be determined by the value of *n*.

If the problem size of algorithm **A** (an arbitrary algorithm) is *n* then we can say, for example, that Algorithm **A requires** $5*n^2$ time units to solve a problem of size *n*. The most important thing to learn is how quickly an algorithm's time requirement grows as a function of the problem size. Algorithm **A** requires a running time that is proportional to n^2**.** . An algorithm's proportional time requirement is known as the ***growth rate***. We can compare the efficiency of the two algorithms by comparing their growth rates.

If Algorithm **A** requires time proportional to **f(n)**, Algorithm **A** is said to be **order f(n),** and it is denoted as**(f(n)).** The **function f(n)** is called the algorithm's **growth-rate function**. Since the capital O is used in the notation, this notation is known as the **Big O notation**.

If Algorithm **A** requires time proportional to $\mathbf{n^2}$, it is $\mathbf{O(n^2)}$**.**

Consider the formal definition of the Big-O function:

Definition:

Let *f* and *g* be nonnegative real-valued functions in the input size *n*. We say that *f*(*n*) is Big-O of *g*(*n*), written*f*(*n*) = O(*g*(*n*)), if there exists positive constants *c* and n_0 such that *f*(*n*) £*cg(n)* for all $n \geq n_0$.

Example:

An algorithm requires $n^2–3*n+10$ seconds to solve a problem size of n.

If constants c and n_0 exist such that

$c*n^2 > n^2–3*n+10$ for all $n \geq n_0$.

the algorithm is order n^2(In fact, c is 3 and n_0 is 2)

$3*n^2 > n^2–3*n+10$ for all $n \geq 2$.

Thus, the algorithm requires no more than $c*n^2$ time units for $n \geq n_0$, So it is O$\mathbf{(n^2)}$

Some useful properties of Growth-Rate Functions:

We can ignore constants in an algorithm's growth-rate function. That is if $f(n)$ is

$O(cg(n))$ for any constant $c>0$, then $f(n)$ is $O(g(n))$.

e.g.: If an algorithm is $O(5n^3)$, it is also $O(n^3)$.

We can ignore low-order terms in an algorithm's growth-rate function. That is, if $f(n)$ is $O(anx+bny+cnz)$ for the constants $a,b,c>0$, and $x>y>z>0$ then $f(n)$ is $O(anx)$, which is also $O(nx)$.

e.g.: if an algorithm is O(n3+4n2+3n), it is also $O(n^3)$.

If $f1(n)$ is $O(g1(n))$ and $f2(n)$ is $O(g2(n))$ then $f1(n)\ f2(n)$ is $O(g1(n)\ g2(n))$ and

$f1(n) + f2(n)$ is $O(g1(n)+ g2(n))$.

e.g.: If an algorithm is $O(n^3)$* $O(4n^2)$, it is also $O(n^3 *4n^2)$ which is $O(4n^5)$ therefore it is $O(n^5)$. Ifan the algorithm is $O(n^3) + O(4n^2)$, it is also $O(n^3 +4n^2)$ therefore it is $O(n^3)$.

Calculation Examples

Example 1:

If an algorithm takes 1 second to run with a problem size of 8, what is the time requirement?

(approximately) for that same algorithm with a problem size of 16?

If its order is:

O(1) T(n) = 1 second

If its order is:

$O(\log_2 n)$ $T(n) = (1*\log_2 16) / \log_2 8 = 4/3$ seconds

If its order is:

O(n) T(n) = (1*16) / 8 = 2 seconds

If its order is:

$O(n*\log_2 n)$ $T(n) = (1*16*\log_2 16) / 8*\log_2 8 = 8/3$ seconds

If its order is:

$O(n^2)$ $T(n) = (1*16^2) / 8^2 = 4$ seconds

If its order is:

$O(n^3)$ $T(n) = (1*16^3) / 8^3 = 8$ seconds

If its order is:

$O(2^n)$ $T(n) = (1*2^{16}) / 2^8 = 2^8$ seconds = 256 seconds

Example 2:

In the following code fragment,

```
i =1
sum = 0;
while (i <= n) {
    i = i + 1;
   sum = sum + i;
}
```

the basic operation is sum = sum + i, which has a cost of 1 unit-time and is executed *n* times.

T(n)= n

So, the growth-rate function for this algorithm is **O(n)**

Example 3:

In the following code fragment,

```
i=1;

sum = 0;

while (i <= n) {

  j=1;

  while (j <= n) {

      sum = sum + i; j = j + 1;

}

  i = i +1;

}
```

the basic operation is sum = sum + i, which has a cost of 1 unit- time and is executed *n*n*

times.

T(n) = n*n

So, the growth-rate function for this algorithm is **$O(n^2)$**

Example 4:

In the following code fragment,

```
for (i=1; i<=n; i++)
    for (j=1; j<=i; j++)
        for (k=1; k<=j; k++)
            x=x+1;
```

the basic operation is x=x+1, which has a cost of 1 and is executed n^3 times

$T(n) = n^3$

So, the growth-rate function for this algorithm is **$O(n^3)$**

What to analyze?

Consider a simple linear search where we are searching for a specific value (num) in an array of size *n*:

```
int i=0;
int place=0;
while (num != arr[i] &&i<n)
    i++;
if (i<n)
    place=i;
```

Say we choose the comparison num != arr[i]as the basic operation. If num happens to be equal to the first element in the array, then the comparison will be made once only. If num is not equal to any element of the array, the comparison will be performed *n+1* times. On average, if we know that num occurs exactly once in the array, the comparison will be performed *n/2* times. As we can see an algorithm can require different times to solve different problems of the same size.

In general, there are three types of algorithm analysis:

Worst-Case Analysis – This is the maximum amount of time that an algorithm requires to solve a problem of size *n*. This gives an upper bound for the time complexity of an algorithm. In the worst case, the linear search is *O(n)*. This is achieved when the search item is not in the list or it is the last item in the list.

Best-Case Analysis –This is the minimum amount of time that an algorithm requires to solve a problem of size *n*. In our search, the best case is when the search item is the first item in the list. The best-case time complexity is then *O(1)*.

Average-Case Analysis –This is the average amount of time that an algorithm requires to solve a problem of size *n*. The linear search is *O(n)* on average. We only focus in this course on the Worst-Case analysis of algorithms.

Space complexity

The analysis techniques used to measure space requirements are similar to those used to measure time requirements. The asymptotic analysis of the growth rate in the time requirement of an algorithm as a function of the input size also applies to the measurement of the space requirement of an algorithm. However, time requirements are measured in terms of basic operations on the data structure performed by the algorithm, whereas space requirements are determined by the data structure itself.

If we have an array with n integers and we want to keep the entire array in memory, the space requirements will be O(*n*). If we have a two-dimensional array, then it will be O(*n*2). However, if the two-dimensional array is not always filled up, and if we can find a way of only setting aside the memory positions as and when we need them, we may save a lot on memory. In this module you'll study various data structures to accomplish such memory savings and various algorithms that could accomplish the same task, but with different time complexities. As you get acquainted with these data structures, you should also learn how to determine the space complexity of each structure.

In the third-year level *Artificial Intelligence* module, COS3751, you will encounter algorithms that operate on implicit data structures, called search spaces. These data structures are huge and space complexity becomes an important issue when using them, even with today's cheap memory. Searchspaces are expanded dynamically; they are never generated and kept in memory in totality. Different orders of expansions yield different space complexities.

Big-O examples

```
//Code Fragment #1

for(int i = 0; i < 2n; i++)

  sum++;
```

The problem size here is the value of n. As the value of n gets bigger, the fragment will take longer to run. Rather than trying to calculate exactly how long it will take to run for a particular value of n, we are interested in how quickly the running time of this fragment increases as n increases. The amount of time a program takes to run depends on how many basic instructions must be executed. To obtain an estimate of this, we can examine the source code and count how many assignments, increments, or conditional tests the fragment would perform. There are four expressions in Fragment#1:

(a) the initialization i = 0

is performed ***1*** time regardless of *n*

(b) the test i< 2n

is performed ***2n + 1*** times – one more time for the final test when i = 2n

(c) the increment i++

is performed ***2n*** times

(d) the increment sum++

is performed ***2n*** times.

So for an input of size ***n***, the number of basic operations this fragment executes is:

6n + 2

The constants in the formula above are not relevant - we are interested in the performance of the fragment for large values of ***n*** so we simply ignore the constant ***2***.

So	→***6n*** **+ 2**	(ignoring 2)
	→***6n***	(ignoring the constant factor)
	→***n***	

Hence the Big-O Notation of Fragment#1 is ***O(n).***

This tells us, that the running time of the program is proportional to ***n***. That is if we double the value of ***n we can*** expect the running time to be approximately double as well. This gives us an indication of the scalability of the program – how well it will perform when the value of ***n*** is large.

This kind of analysis is quite exhausting. We can often analyze the running time of a program by determining the number of times selected statements are executed. We can usually get a good estimate of the running time by considering one type of statement such as some statement within a looping structure. Provided the loop is iterating sequentially in a linear fashion. Thus if we just consider sum++ we note that it runs ***2n*** times. Ignoring the constant - the Big O Notation of Fragment#1 is ***O(n)***. This works because Big O is just an estimate of the running time.

```
//Code Fragment #2

for(inti = 0; i< n; i++)

    for(int j = 0; j <i; j++)

        sum++; //Line 3
```

We begin by analyzing the number of times Line 3 is executed. When i = 1, Line 3 is executed once.When i = 2, Line 3 is executed twice. In general, Line 3 is executed exactly i times. Therefore, the total number of executions of Line 3 is:

1 + 2 + 3 + . . . + n - 1 times.

By mathematical proof, we know that:

1 + 2 + 3 + . . . + n = n(n + 1)/ 2

For mathematical buffs, this formula will be familiar. But if you have not seen this before, *do not worry* about the proof, we will not expect you to know this and there are easier ways of finding the Big O, which will be discussed shortly.

Applying the formula:

1 + 2 + 3 + . . . + n – 1= (n – 1)((n -1) + 1)/2 = (n – 1)n/2 = n2/2 - n/2

Omitting constants and insignificant terms the Big O Notation of Fragment#3 is expressed by ***O(n2).***

As you can see trying to count the exact number of times Line 3 is executed as a function of ***n*** gets tricky, especially for this fragment. The following rule may be applied:

The Multiplication Rule: The total running time of a statement inside a group of nested loops is the running time of the statements multiplied by the product of the sizes of all the for loops. We need only look at the maximum potential number of repetitions of each nested loop.

Applying the Multiplication rule to Fragment #2 we get:

The inner for loop *potentially* executes ***n*** times (the maximum value of i will be ***n-1***) and the outer loop executes ***n*** times. The statement sum++; therefore executes not many times less than ***n * n***, so the Big O Notation of Fragment#3 is expressed by $O(n^2)$.

```
//Fragment #3

for(int i = 0; i < n*n; i++)

    sum++;

for(int i = 0; i < n; i++)

    for(int j = 0; j < n*n; j++)

        sum++;
```

We consider only the number of times the sum++ expression within the loop is executed. The first sum expression is executed n^2 times. While the second sum++ expression is executed $n * n^2$ times. Thus the running time is approximately $n^2 + n^3$. As we only consider the dominating term - the Big O Notation of this fragment is expressed by $O(n^3)$.

Note that there were two sequential statements therefore we added n^2 to n^3. This problem illustrates another rule that one may apply when determining the Big O. You can combine sequences of statements by using the **Addition rule**, which states that **the running time of a sequence of statements is just the maximum of the running times of each statement.**

```
//Fragment #4

for(int i =0; i < n; i++)

   for(int j = 0; j < n*n; j++)

        for(int k = 0; k < j; k++)

            sum++;
```

We note that we have three nested for loops: the innermost, the inner and the outer for loops. The innermost loop potentially executes $n*n$ times (the maximum value of j will be $n*n-1$). The inner loop executes $n*n$ times and the outer for loop executes n times. The statement sum++; therefore executes not many times less than $n*n * n*n * n$, so the Big O Notation of Fragment #4 is expressed by $O(n^5)$.

```
//Fragment #5

for(inti = 0; i<2^10; i++)

    for(int j = 0; j < n; j++)

        sum++;
```

Note that the outer loop runs 2^{10}times while the inner loop runs ***n*** times. Hence the running time is:

$2^{10} * n$. The Big O Notation of this Fragment is expressed by ***O(n)***(ignoring constants).

```
//Fragment #6

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

    for(int i = n; i > 1; i = i/2)

        sum++;
```

The first loop runs ***n*** times while the second loop runs ***log n*** times. (Why? Suppose ***n*** is 32, thensum++; is executed 5 times. Note 25 = 32, therefore, log2 32 = 5. Thus the running time is $\log_2 n$, but we can ignore the base.) Hence the Big O Notation of Fragment #6 is expressed by ***O(n log n)*** (Multiplication Rule.)