Analysis of Algorithm Complexity

Group 7

University of Infomation Technology

June 26, 2021

Faculty Of Computer Science

What is algorithm complexity?

Concept

- An algorithm built must be accompanied by a quantitative evaluation.
- For comparison with other related algorithms.
- There are *two* approaches, corresponding to each evaluation criterion
  - **Time efficiency**, also called **time complexity**.
  - **Space efficiency**, also called **space complexity**.

Time efficiency

- The most intuitive way to quantify the efficiency of an algorithm.
- Under the same operating conditions, the algorithm that gives the earliest results will be the best.

Time efficiency

```
int Fib1(int n) {
    int a, b, c;
    if(n <= 1)
        return n;
    a = 0; b = 1; c = 0;
    for(int k = 2; k <= n; ++k) {
        c = a + b;
        a = b;
        b = c;
    }
    return c;
}
```

Time efficiency


```
int Fib2(int n){
    if(n <= 1)
        return n;
    return Fib2(n - 1)
                + Fib2(n - 2);
}
```

Time efficiency

N	Fib1	Fib2
40	400 *ns*	1048 μs
60	61 *ns*	1s
80	81 *ns*	18 minutes
100	101 *ns*	13 days
120	121 *ns*	36 years
160	161 *ns*	$3.8 * 10^7$ years
200	201 *ns*	$4 * 10^{13}$ years

Table 1: Comparison of execution time of two above algorithms for Fibonacci

Space efficiency

- Based on system resource consumption.
- Based on the using data structure.
- Space efficiency is typically not of as much concern.

Space efficiency

First algorithm	Second algorithm
$temp \leftarrow a;$	$a \leftarrow a + b;$
$a \leftarrow b;$	$b \leftarrow a - b;$
$b \leftarrow temp;$	$a \leftarrow a - b;$

Table 2: Two algorithms for converting the values of two variables.

Why do we need to evaluate algorithms?

- Help us choose the most suitable algorithm.
- Simple but still general, can be applied to many different problems..
- Saving time and not be limited by hardwares, languages, compiler, input data, ...

Notations

Decision parameter and basic operation

- The size of the list
- The polynomial's degree or the number of its coefficients
- Number b of bits in the n's binary representation.

$$b = \log_2 n + 1$$

Decision parameter and basic operation

- For measuring of an algorithm's efficiency, we would like to have a metric that does not depend on these extraneous factors.
- One possible approach is to count *the number of times* each of the algorithm's operations is executed.

$\Rightarrow$ *Basic operation:* the operation contributing the most to the total running time.

- Typically arithmetic operations: $+, -, *, /, \ldots$ The time order to perform these operations is $/, *, (+, -)$.

Decision parameter and basic operation

INSERTION-SORT(A)	*cost*	*times*
for $j \leftarrow 2$ **to** $length[A]$	c_1	n
do $key \leftarrow A[j]$	c_2	$n-1$
▷ Insert $A[j]$ into the sorted		
▷ sequence $A[1 \mathinner{..} j-1]$.	0	$n-1$
$i \leftarrow j-1$	c_4	$n-1$
while $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
do $A[i+1] \leftarrow A[i]$	c_6	$\sum_{j=2}^{n}(t_j - 1)$
$i \leftarrow i-1$	c_7	$\sum_{j=2}^{n}(t_j - 1)$
$A[i+1] \leftarrow key$	c_8	$n-1$

Figure 1: Number of basic operations in Insertion Sort

Decision parameter and basic operation

Problem	**Decision param.**	**Basic op**
Find a number k in an array of size n	Size of array n	Comparison
Matrix multiplication	Number of dimensions or elements	Multiplication
Is n a prime number?	n'size $=$ number of digits (binary representation)	Division

Decision parameter and basic operation

- Let c_{op} be the execution time of an algorithm's basic operation.
- Let $C(n)$ be the number of times this operation needs to be executed for this algorithm.

$\Rightarrow$ We can estimate the running time $T(n)$ of a program implementing this algorithm:

$$T(n) \approx c_{op} C(n)$$

Assuming that: $C(n) = \frac{1}{2}n(n-1)$

(?) How much longer will the algorithm run if we double its input size?

$$C(n) = \frac{1}{2}n(n-1) = \frac{1}{2}n^2 - \frac{1}{2}n \approx \frac{1}{2}n^2$$

$$\Rightarrow \frac{T(2n)}{T(n)} \approx \frac{c_{op}C(2n)}{c_{op}C(n)} \approx \frac{\frac{1}{2}(2n)^2}{\frac{1}{2}n^2} = 4$$

- A difference in running times on small inputs is not what really distinguishes efficient algorithms from inefficient ones.
- In fact, with n being a small value, most algorithms give the same time. Only when $n \to \infty$ that the difference becomes more precisely.

Orders of Growth

Group 7

n	$log_2 n$	n	$nlog_2 n$	n^2	n^3	$2n$	$n!$
10	3.3	10^1	$3.3 \cdot 10^1$	10^2	10^3	10^3	10^6
10^2	6.6	10^2	$6.6 \cdot 10^2$	10^4	10^6	$1.3 \cdot 10^{30}$	10^{157}
10^3	10	10^3	$1.0 \cdot 10^4$	10^6	10^9		
10^4	13	10^4	$1.3 \cdot 10^5$	10^8	10^{12}		
10^5	17	10^5	$1.7 \cdot 10^6$	10^{10}	10^{15}		
10^6	20	10^5	$2.0 \cdot 10^7$	10^{12}	10^{18}		

Table 3: An example of orders of growth

Worst-case

The worst case(The lowest efficiency)

$$C_{worst}(n) = n$$

- The algorithm runs **the longest** among all possible inputs of that size.
- Way to determine: analyze the algorithm to see what kind of inputs yield **the largest** value of the basic operation's count $C_{worst}(n)$ among all possible inputs of size n.

Best-case

The best case (The most efficient)

$$C_{best}(n) = 1$$

- The algorithm runs **the fastest** among all possible inputs of that size.
- Way to determine: analyze the algorithm to see what kind of inputs yield **the smallest** value of the basic operation's count $C_{best}(n)$ among all possible inputs of size n.

Average-case

Average case = average

- Execute the algorithm many time with the same input size n (use some distribution function to create these inputs randomly).
- Get the total runtime and divide by number of executions.

Notations

Some functions will be use in this section:

- $t(n)$ will be an algorithm's running time .
  (usually indicated by its basic operation count $C(n)$).
- $g(n)$ will be some simple function to compare the count with $t(n)$

- $O(g(n))$ is the set of all functions with a **lower or same** order of growth as $g(n)$ (within a constant multiple and $n \to \infty$).
- Example:

$$n \in O(n^2) \tag{1}$$

$$100n + 5 \in O(n^2) \tag{2}$$

$$\frac{1}{2}n(n-1) \in O(n^2) \tag{3}$$

Mathematical definition:

$$t(n) \in O(g(n)) \iff t(n) \leqslant cg(n) \quad (\exists c > 0, n \geqslant n_0)$$

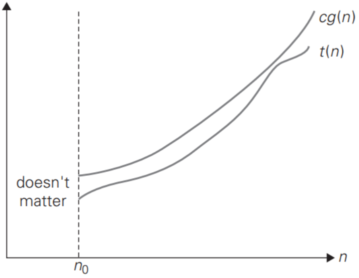


Figure 2: Big-O notation: $t(n) \in O(g(n))$

Prove that: $100n + 5 \in O(n^2)$

$$100n + 4 \leqslant 100n + n \quad (\forall n \geqslant 5)$$

$$= 101n \leqslant 100n^2$$

Complete the proof with $c = 101, n_0 = 5$.

The definition gives us a lot of freedom in choosing specific values for constants c and n_0.

Example: $100n + 5 \leqslant 100n + 5n \quad (\forall n \geqslant 1)$

Big-Omega (Ω)

- $\Omega(g(n))$ stands for the set of all functions with a **higher or same** order of growth as $g(n)$ (within a constant multiple and $n \to \infty$).
- Example:

$$n^3 \in \Omega(n^2) \tag{1}$$

$$\frac{1}{2}n(n-1) \in \Omega(n^2) \tag{2}$$

but

$$100n + 5 \notin \Omega(n^2) \tag{3}$$

Mathematical definition:

$$t(n) \in \Omega(g(n)) \iff t(n) \geqslant cg(n) \quad (\exists c > 0, n \geqslant n_0)$$

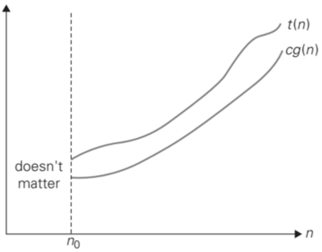


Figure 3: Big-Ω notation: $t(n) \in \Omega(g(n))$

Big-Omega (Ω)

Here is an example of the formal proof:

$$\begin{aligned} & n^3 \in \Omega(g(n)) \\ \iff \; & t(n) \geqslant cg(n) \quad (\forall n \geqslant 0) \end{aligned}$$

We can select $c = 1, n_0 = 0$

Big-Theta (Θ)

- $\Theta(g(n))$ is the set of all functions with a **same** order of growth as $g(n)$ (within a constant multiple and $n \to \infty$).
- Thus, every quadratic function "$an^2 + bn + c$", $\forall a > 0$ is in $\Theta(n^2)$

Big-Theta (Θ)

Mathematical definition:

$$t(n) \in \Theta(g(n))$$
$$\iff c_2 g(n) \leqslant t(n) \leqslant c_1 g(n) \quad (\exists c_1, c_2 > 0, n \geqslant n_0)$$

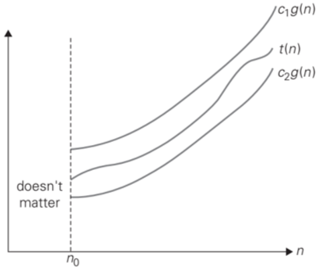


Figure 4: Big-Θ notation: $t(n) \in \Theta(g(n))$

Prove that: $\frac{1}{2}n(n-1) \in \Theta(n^2)$

- Prove the right inequality (the upper bound):

$$\frac{1}{2}n(n-1) = \frac{1}{2}n^2 - \frac{1}{2}n \leqslant \frac{1}{2}n^2 \quad (\forall n \geqslant 0) \tag{1}$$

- Prove the left inequality (the lower bound):

$$\frac{1}{2}n(n-1) = \frac{1}{2}n^2 - \frac{1}{2}n \geqslant \frac{1}{2}n^2 - \frac{1}{2}n\frac{1}{2}n = \frac{1}{4}n^2 \quad (\forall n \geqslant 2) \tag{2}$$

- Hence, we can select: $c_2 = \frac{1}{4}, c_1 = \frac{1}{2}$ và $n_0 = 2$

$$\lim_{n\to\infty} \frac{t(n)}{g(n)} = \begin{cases} 0 \\ c \\ \infty \end{cases}$$

With:

- $lim = 0$: $t(n)$ has a **smaller** order of growth than $g(n)$.
- $lim = c$: $t(n)$ has the **same** order of growth as $g(n)$.
- $lim = \infty$: $t(n)$ has a **larger** order of growth than $g(n)$.

How to compute algorithm complexity?

Common time complexities

- Constant time: $O(1)$
- Logarithmic time: $O(\log_2 n)$
- Linear time: $O(n)$
- Polynomial time: $O(P(n))$
- Exponential time: $O(2^n)$

Common time complexities

An algorithm is said to be

- $O(1)$ does not depend on the size of the input.
- $O(n)$ means that the running time increases at most linearly with the size of the input.
- $O(P(n))$ if its running time is upper bounded by a polynomial expression in the size of the input for the algorithm.
- $O(2^n)$ if running time is bounded by $O(2^{n^k})$ for some constant k.

Constant time, $O(1)$:

```
age = int(input())             # O(1)
visitors = 0                   # O(1)
if age < 17:
    status = "Not allowed"     # O(1)
else:
    status = "Welcome!"        # O(1)
    visitors += 1              # O(1)
```

Common time complexities

- Logarithmic time, $O(logn)$: as the ratio of the number of operations to the size of the input decreases and tends to *zero* when n increases.
- commonly found in operations on binary trees or when using binary search.

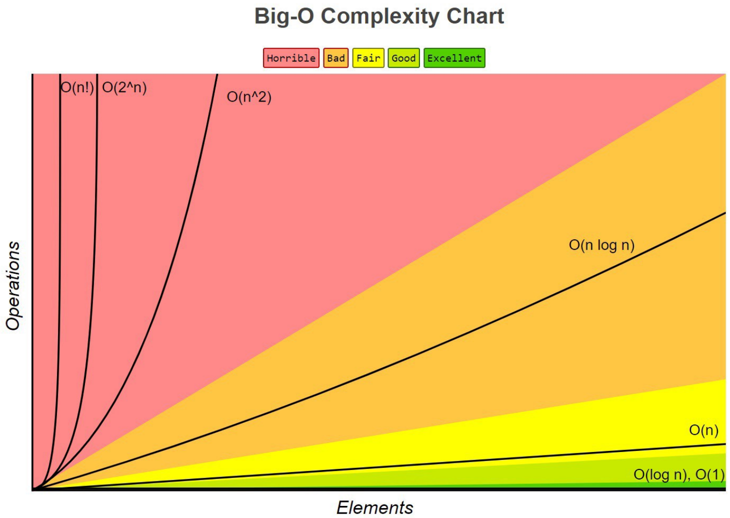
Linear time, $O(n)$:

```
total = 0                  # O(1)
for i in range(n)
    total += i             # O(n)
print(total)               # O(1)
```

Polynomial time, $O(P(n))$:

```
x = 0                              # O(1)
for i in range(n):
    for j in range(n):
        for k in range(n):
            x += 2                 # O(n^3)
```

Common time complexities



Rules

- Drop constance:

$$T(n) = O(c \cdot f(n)) = O(f(n)) \quad c = const, c \geqslant 0 \tag{1}$$

- Get max:

$$O(f(n)) + O(g(n)) = O(max(f(n), g(n)) \tag{2}$$

- Multiplication $T(n) = O(f(n))$
  If we excute $k(n)$ times with $k(n) = O(g(n))$, then the complexity will be $O(g(n) \cdot f(n))$.

Examples


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

Examples

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

- Number of excution times of $p = p * x / j$ is $n(n+1)/2$.
- Time complexity of this 7 lines of code is:

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

Examples

```python
def function():
    num1 = 50000
    n = num1 / 1000
    m = 2
    for i in range(n):
        for j in range(m):
            print("This is 1st string")
    num2 = 10000
    x = num2 / 1000
    for i in range(x):
        for j in range(x):
            print("This is 2nd string")
            print("This is 3rd string")
    return "Returned string"
```

Examples

- Line 2, 3, 4, 7, 8, 9, 12, 13, 14: each line costs $O(1)$ so number of times in total is **9**.
- Line 5, 6: $n * m$
- Line 10, 11: $x * x = x^2$
- Apply addition rule: $9 + n * m + x^2$
- Apply drop constance rule: $n * m + x^2$
- Apply get max rule: $\max(n * m, x^2)$ is the time complexity of above function.

Overview

Overview

- Algorithm complexity is used to evaluate algorithms from which to choose the most optimal algorithm.
- The higher the complexity of the algorithm, the more time it will take.
- Evaluating an algorithm before putting it into practice that will save time and money.
- There are 3 simple rules need to remember, those are **drop constance, multiplication and get max**.

Group 7

References

References

[1] Dinh Ba Tien Tran Dan Thu.
Nhap mon lap trinh.
NXB Khoa hoc va Ky thuat, 2018.

[2] Do phuc tap cua thua toan.

[3] Time complexity.

[4] Tran Van Tuan.
Do phuc tap cua thua toan.

[5] Nguyen Yen Bao.
Time complexity.

[6] How to find time complexity of an algorithm.

[7] Philips Tel.
Determining the complexity of algorithm (the basic part).