**Time complex :**

there are k nested loops, the time complexity is O(n).

For example, the time complexity of the following code is O(n):

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

// code

}

And the time complexity of the following code is O(n^2):

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

{

for (int j = 1; j <= n; j++) {

// code

}

}

In the following examples, the code inside the loop is executed 3n, n+5 and dn/2e times, but the

time complexity of each code is O(n).

for (int i = 1; i <= 3\*n; i++) {

// code

}

for (int i = 1; i <= n+5; i++) {

// code

}

for (int i = 1; i <= n; i += 2) {

// code

}

As another example, the time complexity of the following code is O(n

2

):

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

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

// code

}

} For example, the following code consists of three phases with time complexities

O(n), O(n

2

) and O(n). Thus, the total time complexity is O(n

2

).

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

// code

}

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

for (int j = 1; j <= n; j++) {

// code

}

}

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

// code

}

Sometimes the time complexity depends on several factors. In this case, the time

complexity formula contains several variables.

For example, the time complexity of the following code is O(nm):

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

for (int j = 1; j <= m; j++) {

// code

}

}

Recursion

The time complexity of a recursive function depends on the number of times

the function is called and the time complexity of a single call. The total time

complexity is the product of these values.

For example, consider the following function:

void f(int n) {

if (n == 1) return;

f(n-1);

}

The call f(n) causes n function calls, and the time complexity of each call is O(1).

Thus, the total time complexity is O(n).

As another example, consider the following function:

void g(int n) {

if (n == 1) return;

g(n-1);

g(n-1);

}

In this case each function call generates two other calls, except for n = 1. Let us

see what happens when g is called with parameter n. The following table shows

the function calls produced by this single call:

function call number of calls

g(n) 1

g(n−1) 2

g(n−2) 4

··· ···

g(1) 2n−1

Based on this, the time complexity is

1+2+4+··· +2

n−1 = 2

n −1 = O(2n

).

Complexity classes The following list contains common time complexities of algorithms: O(1) The running time of a constant-time algorithm does not depend on the input size. A typical constant-time algorithm is a direct formula that calculates the answer. O(logn) A logarithmic algorithm often halves the input size at each step. The running time of such an algorithm is logarithmic, because log2 n equals the number of times n must be divided by 2 to get 1. O( p n) A square root algorithm is slower than O(logn) but faster than O(n). A special property of square roots is that p n = n/ p n, so the square root p n lies, in some sense, in the middle of the input. O(n) A linear algorithm goes through the input a constant number of times. This is often the best possible time complexity, because it is usually necessary to access each input element at least once before reporting the answer. O(nlogn) This time complexity often indicates that the algorithm sorts the input, because the time complexity of efficient sorting algorithms is O(nlogn). Another possibility is that the algorithm uses a data structure where each operation takes O(logn) time. O(n 2 ) A quadratic algorithm often contains two nested loops. It is possible to go through all pairs of the input elements in O(n 2 ) time. O(n 3 ) A cubic algorithm often contains three nested loops. It is possible to go through all triplets of the input

elements in O(n 3 ) time. O(2n ) This time complexity often indicates that the algorithm iterates through all subsets of the input elements. For example, the subsets of {1,2,3} are ;, {1}, {2}, {3}, {1,2}, {1,3}, {2,3} and {1,2,3}. O(n!) This time complexity often indicates that the algorithm iterates through all permutations of the input elements. For example, the permutations of {1,2,3} are (1,2,3), (1,3,2), (2,1,3), (2,3,1), (3,1,2) and (3,2,1).

input size required time complexity

n ≤ 10 O(n!)

n ≤ 20 O(2n

)

n ≤ 500 O(n

3

)

n ≤ 5000 O(n

2

)

n ≤ 106 O(nlogn) or O(n)

n is large O(1) or O(logn)