Data Structures and Algorithms

Trong-Hop Do
University of Information Technology, HCM city

Complexity Analysis

Asymptotic Analysis

Given two algorithms for a task, how do we find out which one is better?

One naive way of doing this is – implement both the algorithms and run the two programs on your computer for different inputs and see which one takes less time.

Here are some running times for this example:

Linear Search running time in seconds on A: 0.2 * n

Binary Search running time in seconds on B: 1000*log(n)

n	Running time on A	Running time on B
10	2 sec	~ 1 h
100	20 sec	~ 1.8 h
10^6	~ 55.5 h	~ 5.5 h
10^9	~ 6.3 years	~ 8.3 h

Time Complexity

In the time complexity analysis we define the time as a function of the problem size and try to estimate the growth of the execution time with the growth in problem size.

Memory Space

- The space require by the program to save input data and results in memory (RAM).

Big-O-Notation

- Big O notation (with a capital letter O, not a zero), also called Landau's symbol, is a symbolism used in complexity theory, computer science, and mathematics to describe the basic behavior of functions. Basically, it tells you how fast a function grows or declines.
- Landau's symbol comes from the name of the German mathematician Edmund Landau who invented the notation.
- The letter O is used because the rate of growth of a function is also called its order.

Difference between complexity and computation time

- Computation time: The interval of solving a problem based on the embedded system architecture. (in the field of computer sciences)

- Complexity: The art of handling a problem based on the algorithm designed to solve a case.

OR

- The difficulty faced by the processor in solving a deployed case on it.

Functions defined in big o notation

$O(1)$	constant(slowest)
$O(\log(n))$	logarithmic
$O((\log(n))^c)$	polylogarithmic (same as $O(\log(n))$)
$O(n)$	linear
$O(n^2)$	quadratic
$O(n^c)$	polynomial
$O(c^n)$	exponential(fastest)

Understanding big o

- Efficiency covers lots of resources, including:
  1. CPU (time) usage (The most important)
  2. Memory usage
  3. Disk usage
  4. Network usage

Performance vs complexity

- 1. Performance: how much time/memory/disk/... is actually used when a program is run. This depends on the machine, compiler, etc. as well as the code.
- 2. Complexity: how do the resource requirements of a program or algorithm scale, i.e., what happens as the size of the problem being solved gets larger?

More about performance

- The time required by a function/procedure is proportional to the number of "basic operations" that it performs, like;

1. one arithmetic operation (e.g., +, *).
2. one assignment (e.g. x := 0)
3. one test (e.g., x = 0)
4. one read (of a primitive type: integer, float, character, Boolean)
5. one write (of a primitive type: integer, float, character, Boolean)

Regarding computing

We express complexity using big-O notation.

For a problem of size N:

A constant-time algorithm is "order 1": O(1)

A linear-time algorithm is "order N": O(N)

A quadratic-time algorithm is "order N squared": $O(N^2)$

Infinite Time algorithm is “Order infinity”: O(inf)

Finding complexity

Generally, we have 6 cases

1. Statements
2. If else
3. Loop
4. Nested loop
5. Function call
6. When

Statement

statement 1;

statement 2;

...

statement k;

The total time is found by adding the times for all statements:

total time = time(statement 1) + time(statement 2) + ... + time(statement k)

If each statement is "simple" (only involves basic operations) then the time for each statement is constant and the total time is also constant: O(1).

If Else

```
if (cond) then
        block 1 (statements)
else
        block 2 (statements)
end if;
```

Here, either block 1 will execute, or block 2 will execute. Therefore, the worst-case time is the slower of the two possibilities:

max(time(block 1), time(block 2))

If block 1 takes O(1) and block 2 takes O(N), the if-then-else statement would be O(N)

LOOP

```
for I in 1 .. N loop
    sequence of statements
end loop
```

The loop executes N times, so the sequence of statements also executes N times.

If we assume the statements are O(1), the total time for the for loop is N * O(1), which is O(N) overall.

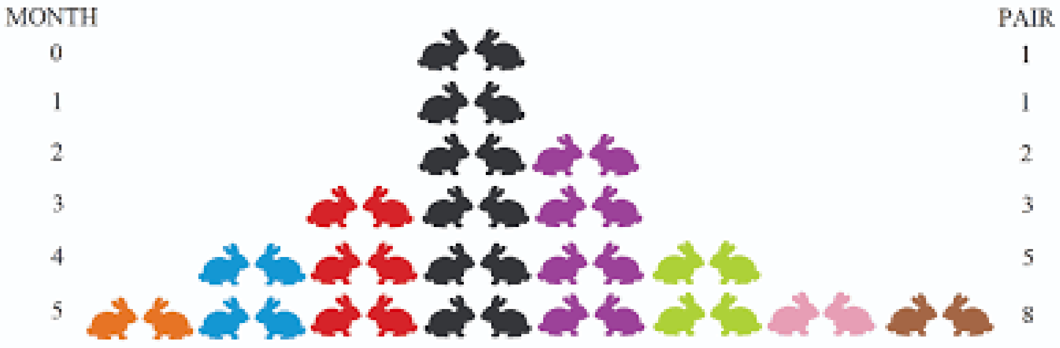
Nested LOOP

```
for I in 1 .. N loop
    for J in 1 .. M loop
        sequence of statements
    end loop;
end loop;
```

The statements in the inner loop execute a total of N * M times.
Thus, the complexity is O(N * M).

Recursive

Input: Some non-negative integer n
Output: The nth number in the Fibonacci Sequence
if $n \leq 1$ **then**
  return n
else
  return $F(n-1) + F(n-2)$;



For n > 1

$$T(n) = T(n-1) + T(n-2) + 1$$

When n = 0 and n = 1

$$T(0) = T(1) = 0$$

$$T(n) = 2^{n} {}^{*}\, T(0) + (2^{n}-1) = 2^{n} + 2^{n} - 1 = O(2^{n})$$

Function Calls

The behavior of function is same as statement if called once

Its behavior is statement in loop if it is called in loop

Its behavior is more like nested loop if it is called inside loop and it has an characteristic loop inside as well

When

- The behavior of such statement is not defined by time or cycles of processing
- It may occur the very next moment
- It might not occur even after the device is expired
- Such algorithms are limited by some thresholds or bounds, becomes O(N)
- Used in training and testing of Artificial Neural Networks and such

Actual complexity vs O(n)

- $an^2 + bn + c$
- $an + b$
- $an \log n + bn + c$

- n^2
- n
- $n \log n$

- Only care about the order of of the biggest term

O(1)

```
void print(int index,int values[]){
     cout<<value[index];
}
```

You can directly access the index element of the array, so no matter how big the array gets, the time it takes to access the element won’t change.

O(n)

```
void print(int index, int values[]){
	for(int x=0;x<index;x++){
		cout << value[index];
	}
}
```

As you increase the size of the array, the time for this algorithm will increase proportionally. An array of 50 will take twice the time as an array of 25 and half the time of an array of 100.

$O(n^2)$

```
int CompareAll (int array1[ ] ,int index1,int array2[ ],int index2){
        for (int x = 0; x < index1; x++)
        {
                bool isMin;
                for (int y = 0; y < index2; y++)
                {
                        if( array[x] > array[y])  isMin = false;
                }
                if(isMin)
                break;
        }
        return array[x];
}
```

If you add one element to array, you need to go through the entire rest of the array one additional time. As the input grows, the time required to run the algorithm will grow quite quickly.

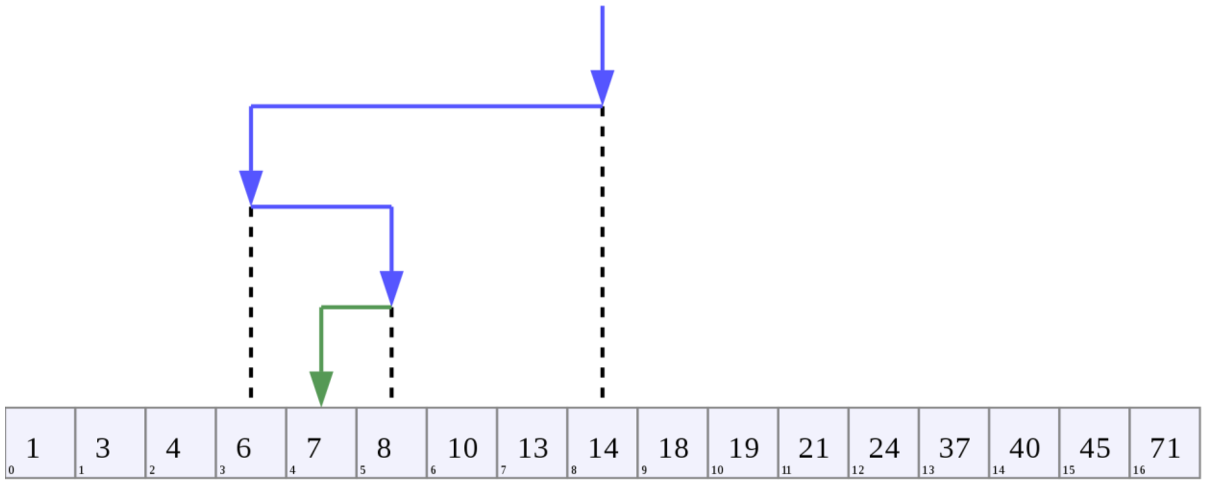
O(n!)

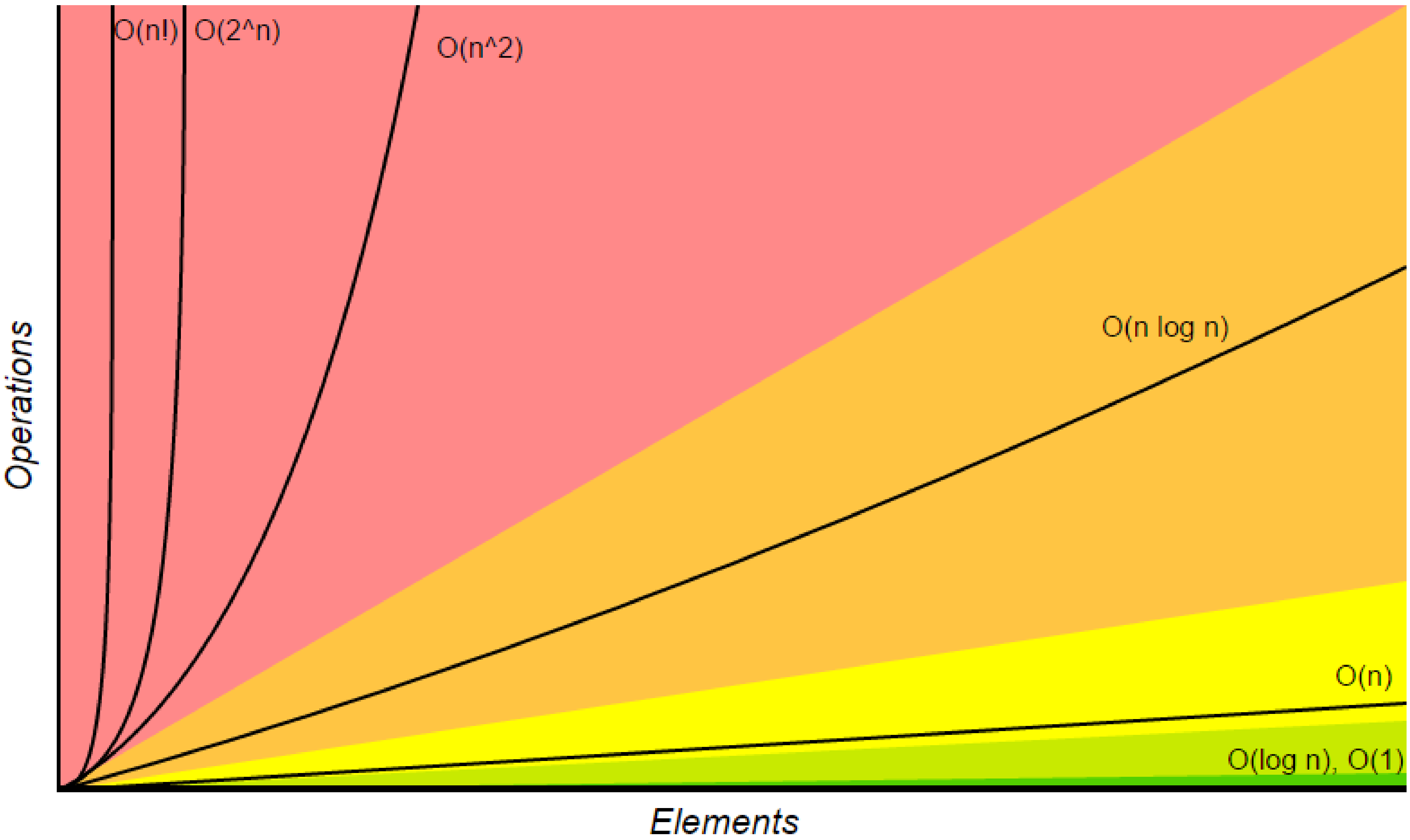
- Travelling salesman problem

in this problem as the number of cities increases the runtime also increases accordingly by a factor of cities factorial.

O(log(n))

```
function binary_search(A, n, T)
    L := 0
    R := n - 1
    while L ≤ R do
        m := floor((L + R) / 2)
        if A[m] < T then
            L := m + 1
        else if A[m] > T then
            R := m - 1
        else:
            return m
    return unsuccessful
```





O(n!)
O(2^n)
O(n^2)
O(n log n)
O(n)
O(log n), O(1)
Operations
Elements

We are having two different algorithms to find the smallest element in the array

```
int FindMin (int array[ ]){
        int x, curMin;  curMin =
array[0];
        for (x = 1; x < 10; x++)
                {       if( array[x] <
curMin)  curMin = array[x];}
        return curMin;
}
```

```
int CompareToAllNumbers (int array[ ]){
        bool is Min;  int x, y;
        for (int x = 0; x < 10; x++){
                isMin = true;
                for (int y = 0; y < 10; y++){
                        if( array[x] > array[y])
                                isMin = false;
                }
        if(isMin)  break;
        }
        return array[x];
}
```

Best ☺

- Uses array to store elements
- Compares a temporary variable in the array
- The Big O notation for it is O(n)

This algorithm is efficient and effective as its O is small as compare to the 2nd algorithm And hence it gives the best result in less time.

Worst ☹

- Uses array to store elements
- Compares the array with array itself
- The big O notation for it is $O(n^2)$

This algorithm is less effective than the other Beacause this algo has a high Big O and hence It is more time taking and less effective.

SCENARIOS

BEST
AVERAGE
WORST

Best case:
When the algorithm takes less
run time w.r.t the given inputs/data.
Best case also defines as Big Ω.

Average case:
When the algorithm takes
average run time w.r.t to the input /data.
This is also known as BigΘ.

Worst case:
When the algorithm takes higher runtime
as compared to the given inputs/data.
Worst case is also called as Big O.

Examples

```
1 for (i = 0; i < n; i++)
2     for (j = 0; j < n; j++)
3         a[i][j] = 0;
4 for (i = 0; i < n; i++)
5     a[i][j] = 1;
```

$T_3 = O(1)$
$T_2 = O(n)$
$T_{23} = O(n) \times O(1) = O(n)$
$T_1 = O(n)$
$T_{123} = O(n) \times O(n) = O(n^2)$
$T_5 = O(1)$
$T_4 = O(n)$
$T_{45} = O(n) \times O(1) = O(n)$
$T_{12345} = T_{123} + T_{45} = O(n^2) + O(n) = O(n^2 + n) = O(n^2)$

```
1 for (i = 0; i < n; i++)
2    for (j = 0; j < n; j++)
3.      if (i == j)
4.         a[i][j] = 1;
5       else
6          a[i][j] = 0;
```

$$T_4 = O(1)$$
$$T_6 = O(1)$$
$$T_3 = O(1)$$
$$T_{3456} = O(1)$$
$$T_2 = O(n)$$
$$T_{23456} = O(n) \times O(1) = O(n)$$
$$T_1 = O(n)$$
$$T_{12345} = O(n) \times O(n) = O(n^2)$$

```
1 for (i = 0; i < n; i++)
2     for (j = 0; j < n; j++)
3         a[i][j] = 0;
4 for (i = 0; i < n; i++)
5     a[i][j] = 1;
```

$$T_3 = O(1)$$
$$T_2 = O(n)$$
$$T_{23} = O(n) \times O(1) = O(n)$$
$$T_1 = O(n)$$
$$T_{123} = O(n) \times O(n) = O(n^2)$$
$$T_5 = O(1)$$
$$T_4 = O(n)$$
$$T_{45} = O(n) \times O(1) = O(n)$$
$$T_{12345} = T_{123} + T_{45} = O(n^2) + O(n) = O(n^2 + n) = O(n^2)$$

```
sum = 0
for (i = 0; i < n; i++)
   for (j = i + 1; j <= n; j++)
      for (k = 1; k < 10; k++)
         sum = sum + i * j * k;
```

```
sum = 0
for (i = 0; i < n; i++)
   for (j = i + 1; j <= n; j++)
      for (k = 1; k < m; k++) {
         x = 2 * y
         sum = sum + i * j * k;
      }
```

```
sum = 0
thisSum = 0
for (i = 0; i < n; i++) {
   thisSum += a[i];
   if (thisSum > sum)
      sum = thisSum;
   else
      thisSum = sum;
}
```