DigiPen
INSTITUTE OF TECHNOLOGY

Chapter 9 Algorithm Analysis

Java AP

Analysis of an algorithm is about understanding its performance. The aspect of the performance depends on the application and the algorithm. Usually we need to know how fast the algorithm will run. Sometimes we need to know how much memory it needs. Also we might be interested in the resources the algorithm requires. For example, if the algorithm gets its input from physical sensors, then the number of censors plays an important role in analyzing the algorithm.

When solving a problem we are faced with a choice among algorithms such as algorithms that are easy to understand, code, and debug. Or algorithm that makes efficient use of the computer resources.
The running time of an algorithm depends on factors such as:

1. The input of the algorithm
2. The quality of the code generated by the compiler used to create the object program
3. The nature and speed of the instructions on the machine used to execute the program
4. The time complexity of the algorithm.

Since the running time depends on the input and the size of the input, it should be defined as a function of the input. In other words, the running time of an algorithm for a given input is the total number of steps it executes, which is proportional to the actual execution time. For example, the natural input size for a sorting algorithm is the number of items to be sorted.
The running time is really a function of the particular input and not just the input size. Consequently, we can analyze the worst case, average case, and the best case.

Example: Assume that we have an array of random integers and that we wish to find out where ***x*** is in the array. When the **while** loop ends, ***n*** will be the position of ***x*** in the array.

```
/* Assume the array is declared and filled with integers */
int n = 0;
while (x != array[n])
{
  n++;
}
```

In order to analyze the above code, the following questions should be asked:
To find the position of ***x*** in the array:

- What is the *least* number of iterations?
- What is the *most* number of iterations?
- What is the *average* number of iterations?

Suppose the array was sorted. Now how would you answer these questions:

- What search method would you use?
- How would the search method affect these questions:
  - What is the *least* number of iterations?
  - What is the *most* number of iterations?
  - What is the *average* number of iterations?

The worst case analysis

As mentioned before, the running time of an algorithm depends on the input order. For example, searching in a sorted list is faster than searching in an unsorted list. In addition, when searching for an item, we might find the item at the very beginning, somewhere in the middle, at the end, or not finding the item. The cases listed in the previous sentence, constitute the best case, average case, and the worst case. Usually, we are interested in how an algorithm behaves in the worst case for the following reasons.

1. Worst case running time often represents a tight bound since worst performance is achieved by the input used in the analysis. In other words, it guarantees that the algorithm will never perform worse than the current case since it gives an upper bound on the performance where all the other cases must at least have the same performance if not a better performance.

2. Many algorithms have the same performance on the best case. For example, both the linear and the binary search performs one comparison on the best case. Consequently, analyzing the performance of the best case is not accurate or very informative.

3. The nature of many algorithms performs to their worst case most of the time. For example, when trying to detect a collision each frame at run time, most of the time the test is performed against all the visible object. Or, when searching for an item in a list, most of the time the item is not found.

4. The average case is usually more difficult to determine because it depends on the input meaning. Since assumptions are adopted in order to carry the average case analysis, the analysis might not be accurate when the assumption is incorrect. For example, the assumption could be taken that all inputs of a given size are equally likely to occur.

Big O Notation

"Big O notation" is used in Computer Science to describe the performance or complexity of an algorithm. Big O specifically describes the **worst-case** scenario, and can be used to describe the execution time required or the space used (e.g. in memory or on disk) by an algorithm. It is represented by an O symbol followed by the complexity between parentheses: $O(1)$, $O(n)$, $O(n^2)$, etc...

Note: n is usually the input's size.

The notation $\boldsymbol{O}()$, is called a number of things such as "Big Oh", "asymptotic O-notation ", "growth rate", "order of", "proportional to", etc.

An algorithm is made from a series of instructions within statements. Usually the cost of the instructions is measured in cycles. What is important to us in this section is the cost concept rather than the cost value. In other words, each statements could be assigned a constant cost. The algorithm total cost is determined by adding all the statements cost together. Of course, if statements are found within the loop, the cost of the statements should be multiplied by the number of loops. The next chapter will discuss the cost summation of a recursive algorithm.

Let's show an example, analyze it and specify its "Big O Notation".

Example:

```
public static void main(String[] args)
{
    int n = 10;
    int k = 0;
    int l = 0;
    int m = 0;

    for (int i=0; i<n; i++) // Will run n iterations
    {
        for (int j=0; j<=n; j++) // Will run n iterations
        {
                k++;   //assume the cost of k++ is 3 units
        }
    } // Complexity: n * n * 3 = 3n^2

    for (int i=0; i<n; i++)  // Will run n iterations
    {
        k++; //assume the cost of the for body statement is 9 units
        l++;
        m++;
    }  // Complexity: n * 9 = 9n

    k++;    //assume the cost of the statement is 3 units
    l++;    //assume the cost of the statement is 3 units
    m++;    //assume the cost of the statement is 3 units
} //Final Complexity: 3n^2 + 9n + 9
```

Explanation:

The running time of the algorithm is $T(n) = 3n^2 + 9n + 9$

- Case n= 10
  - Running time for $3n^2$: $\frac{3(10)^2}{3(10)^2 + 9(10) + 9} = 75.18\%$
  - Running time for 9n: $\frac{9(10)}{3(10)^2 + 9(10) + 9} = 22.55\%$
  - Running time for 9: $\frac{9}{3(10)^2 + 9(10) + 9} = 02.27\%$

- Case n= 100
  - Running time for $3n^2$: $\frac{3(100)^2}{3(100)^2 + 9(100) + 9} = 97.06\%$
  - Running time for 9n: $\frac{9(100)}{3(100)^2 + 9(100) + 9} = 02.91\%$
  - Running time for 9: $\frac{9}{3(100)^2 + 9(100) + 9} = 00.03\%$

- Case n= 1000
  - Running time for $3n^2$: $\frac{3(1000)^2}{3(1000)^2+9(1000)+9} = 99.700\%$
  - Running time for 9n: $\frac{3(1000)}{3(1000)^2+9(1000)+9} = 00.099\%$
  - Running time for 9: $\frac{9}{3(1000)^2+9(1000)+9} = 00.001\%$

We notice that when n is 100, the higher term $3n^2$ accounts for 97% of the algorithm running time. And when n is 1000, the higher term $3n^2$ accounts for 99.7% of the algorithm running time. Finally, when n is 10000, the higher term $3n^2$ accounts for 99.97% of the algorithm running time.

Now it is clear that the significance of the lower order terms is negligible when the size of the input is relatively large. And since we are interested in the worst case analysis, when we look at a function in terms of growth rate we will only consider the higher-order terms of the formula:

$$O(T(n)) = O(3n^2 + 9n + 9) = O(3n^2) = O(n^2)$$

Analysis of Simple Iterative Algorithms

Example:

```
public static boolean IsFirstValue5(int array[])
{
    if(array[0] == 5)
    {
        return true;
    }

    return false;
}
```

Explanation:

No matter how big the array is, the algorithm will keep a constant rate.
Best case scenario = Worst Case Scenario = $\boldsymbol{O(1)}$

Example:	`public static int IndexOf(int array[], int value)` `{` `  for(int i = 0; i < array.length; ++i)` `  {` `  if(array[i] == value)` `  {` `  return i;` `  }` `  }` `  return -1;` `}`
Explanation:	This algorithm returns the index of the first occurrence of the user specified value in the array. The length of the array does affect the algorithm's rate. Best case scenario: First element in the array is equal to **value** Worst case scenario: No element in the array is equal to **value**, which means that the loop has to go through all the elements before exiting the function. A matching integer could be found during any iteration of the for loop and the function would return early, but "Big Oh" notation will always assume the upper limit where the algorithm will perform the maximum number of iterations. That fact demonstrates how "Big Oh" favors the worst-case performance scenario; $\boldsymbol{O(N)}$

Example:	`public static void MyPrint(int n)` `{` `  for(int i = 0; i < n+50; ++i)` `  {` `  System.out.println(i);` `  }` `}`
Explanation:	This algorithm simply prints out the values from 0 to n+50. **n**'s value does affect the algorithm's speed. Both "best case scenario" and "worst case scenario" are equal. Although it is obvious that the complexity is n + 50, "Big Oh" notation is an approximation for large values of n so: $$\boldsymbol{O(N+50) = O(N)}$$

Example:

```
public static boolean Duplicates(int array[])
{
    for(int i = 0; i < array.length; ++i)
    {
        for(int j = 0; j < array.length; ++j)
        {
            if(i == j) /* Don't compare with self */
            {
                continue;
            }

            if(array[i] == array[j])
            {
                return true;
            }
        }
    }

    return false;
}
```

Explanation:

This algorithm checks if the array contains any duplicates.

The length of the array does affect the algorithm's speed.

Best case scenario: First and second elements in the array are equal.
Worst case scenario: No duplicates found, which means that the outer loop has to go through all the elements which calls the inner loop every single time.

Since every outer loop iteration will call the inner loop, the complexity is:

$$\boldsymbol{O}(\boldsymbol{N} * \boldsymbol{N}) = \boldsymbol{O}(\boldsymbol{N^2})$$

Example:

```
public static void PrintVlaues(int n)
{
    for(int i = 0; i < n*10; ++i)
    {
        for(int j = 0; j < (n+5)*3; ++j)
        {
           System.out.println(i * j);
        }
    }
}
```

Explanation:

This algorithm will print out weird values. Its purpose is to teach how to compute its complexity.

	The outer loop will iterate 10*n times. The inner loop will iterate (n+5)*3 times for every outer loop iteration. Algorithm complexity is: $(10n) * ((n + 5) * 3) = (10n) * (3n + 15) = 30n^2 + 150n$ "Big Oh" notation is an approximation for large values of n so: $$O(30N^2 + 150N) = O(N^2)$$

Analysis of Recursive Algorithms

- Recursion breaks a problem into several smaller problems.
- The smaller problems are exactly the same type as the original problem.
- A recursive solution solves a problem by solving a smaller instance of the same problem, where the known problem is solved by solving an even smaller instance of the same problem. The new problem will be so small that its solution will be either obvious or known. Finally, this solution will lead to the solution of the original problem.
- Not all recursive solutions are better than iterative solutions.
- Consider the problem of looking up a word in a dictionary. We can start looking at the beginning of the dictionary and looking at every word until a match.
  Using sequential search may be very slow sometimes, a faster way to perform the search is similar to the way in which we actually use a dictionary. We open the dictionary probably to a point near its middle, and by glancing at the page, we determine which half contains the desired word.
- A solution to the problem:

Pseudo Code

```
Search(Dictionary, Word)
{
      if (Dictionary is one page in size)
            Scan the page for the Word //Obvious solution - base case
      else
      {
            Open the dictionary to point near the middle
            Determine which half of Dictionary contains the Word
            if (Word is in the first half of the Dictionary)
                  Search(First half of Dictionary, Word)
            else
                  Search(Second half of Dictionary, Word)
      }
   }
```

- Writing a solution as a function allows several observations:
  1. One of the actions of the function is to call itself. The solution strategy is to split the **Dictionary** in half, then determine which half contains **Word**, and apply the same strategy to the half.
  2. At each successive call to **Search(Dictionary, Word)**, the size of the Dictionary is smaller. The function solves the search problem by solving another search problem that is identical in nature but smaller in size.

3. There is one search problem that we handle differently from all the others. When the Dictionary contains only a single page, another obvious method (sequential search) is used. Searching a one page dictionary is **the base case**, where recursive calls stops and the problem is solved directly.
4. The manner in which the size of the problem diminishes ensures that we will eventually reach the base case.

- When constructing a recursive solution, keep in mind the following four questions:
  1. How can we define the problem in terms of smaller problems of the same type?
  2. How does each recursive call diminish the size of the problem?
  3. What instance of the problem can serve as the base case?
  4. As the problem size diminishes, will we reach this base case?

The Factorial of N

- The iterative definition of Factorial(n)=n*(n-1)*(n-2)*…*1 for any integer n>0.
- Factorial(0)=1
- The factorial of a negative number is undefined.

The recursive definition of the factorial function is:

$$\text{Factorial(n)}=\begin{cases} 1 & if\ n=0 \\ n*Factorial(n-1) & if\ n>0 \end{cases}$$

Example:

- Factorial(4)=4*Factorial(3)
- Factorial(3)=3*Factorial(2)
- Factorial(2)=2*Factorial(1)
- Factorial(1)=1*Factorial(0)
- The base case is reached and the definition states that Factorial(0)=1 and the information to solve Factorial(4) is now available.
- Factorial(1)=1*Factorial(0)=1*1=1
- Factorial(2)=2*Factorial(1)=2*1=2
- Factorial(3)=3*Factorial(2)=3*2=6
- Factorial(4)=4*Factorial(3)=4*6=24

Iterative	

```
public static long Factorial_Iterative(int n)
{
    int i, result=1;

    for (i = 1; i <= n; ++i)    //Assuming n>0
    {
        result = result * i;
    }

    return result;
}
```

Recursive

```
public static long Factorial_Recursive(int n)
{
    if (n==0)
    {
        return 1;
    }
    else        //Assuming n>0
    {
        return n * Factorial_Recursive(n-1);
    }
}
```

Explanation

The complexity of the iterative and the recursive algorithms is linear **O(n)**.

Benchmark: clocked 5 times 10,000,000 calls/clock n = 10

```
Factorial_Iterative: 5 milliseconds
Factorial_Recursive: 1080 milliseconds

Factorial_Iterative: 5 milliseconds
Factorial_Recursive: 1114 milliseconds

Factorial_Iterative: 5 milliseconds
Factorial_Recursive: 1111 milliseconds

Factorial_Iterative: 4 milliseconds
Factorial_Recursive: 1112 milliseconds

Factorial_Iterative: 5 milliseconds
Factorial_Recursive: 1108 milliseconds
```

Even though both functions have the same complexity, we notice that the recursive algorithms takes more time than the iterative algorithm due to the overhead of the function calls.

Raising an integer to a power

The iterative definition of the function x^n

$$x^n = \begin{cases} 0 & if \quad x = 0, \quad n \neq 0 \\ not\ defined & if \quad x = 0, \quad n = 0 \\ 1 & if \quad x \neq 0, \quad n = 0 \\ x^n & if \quad x \neq 0, \quad n > 0 \end{cases}$$

Algorithm	`public static long Power_Iterative(long x, long n)` `{` `  //Assuming x is an integer and n is a non-negative integer` `  long result=1, i;` `  for (i = 1; i <= n; ++i)` `  {` `  result *= x;` `  }` `  return result;` `}`
Explanation	It is obvious that the algorithm will do **n** iterations, hence the complexity is linear **O(n)**.

The first recursive power approach will be using the following rule of exponents: $x^n = x * x^{n-1}$
By definition $x^0 = 1$ which is the base case
The recursive definition of the function power(x, n) is:

$$\text{Power(x, n)} = \begin{cases} x = 1, & n = 0 \\ x^n = x \times x^{n-1}, & n > 0 \end{cases}$$

Algorithm	`public static long Power_Recursive1(long x, long n)` `{` `  //Assuming x is an integer and n is a non-negative integer` `  if (n==0)` `  {` `  return 1;` `  }` `  else` `  {` `  return x * Power_Recursive1(x, n-1);` `  }` `}`

Explanation	To compute x^n, the function Power is called n+1 times; once from the initial call to PowerRecursive1(x, n) and n times recursively. The complexity is linear **O(n + 1) = O(n).**

The second recursive power approach will be using the rule of exponents: $x^n = (x^{\frac{n}{m}})^m$

If m=2 $x^n = (x^{\frac{n}{2}})^2 = \begin{cases} \left(x^{\frac{n}{2}}\right)\left(x^{\frac{n}{2}}\right) & if\ n\ is\ even \\ \left(x^{\frac{n}{2}}\right)\left(x^{\frac{n}{2}}\right)(x) & if\ n\ is\ odd \end{cases}$

Algorithm	``` public static long Power_Recursive2(long x, long n) { //Assuming x is an integer and n is a non-negative integer long halfPower; if (n==0) { return 1; } else { halfPower = Power_Recursive2(x,n/2); if (n%2==0) //check if n is even { return halfPower * halfPower; } else { return x * halfPower * halfPower; } } } ```
Explanation	The complexity of the second recursive algorithm is logarithmic since we are dividing the value of n by 2 every recursive call. $O(Log_2 N)$

Benchmark	clocked 5 times 10,000,000 calls/clock n = 11 x = 13 Power_Iterative: 1089 milliseconds Power_Recursive1: 1256 milliseconds Power_Recursive2:657 milliseconds Power_Iterative: 1081 milliseconds Power_Recursive1: 1235 milliseconds Power_Recursive2:862 milliseconds Power_Iterative: 1088 milliseconds Power_Recursive1: 1242 milliseconds Power_Recursive2:650 milliseconds Power_Iterative: 1087 milliseconds Power_Recursive1: 1250 milliseconds Power_Recursive2:652 milliseconds Power_Iterative: 1086 milliseconds Power_Recursive1: 1247 milliseconds Power_Recursive2:652 milliseconds

Fibonacci sequence

0 1 1 2 3 5 8 13 21 34 55 …is called the Fibonacci sequence.

Where:

- F0 = 0
- F1 = 1
- F2 = 1
- F3 = 2
- F4 = 3
- F5 = 5
- F6 = 8
- F7 = 13
- F8 = 21

The value of: $F_n = F_{n-1} + F_{n-2}$ except for $F_0=0$ and $F_1=1$.

$$F(n)=\begin{cases} 0 & n=0 \\ 1 & n=1 \\ F(n-1)+F(n-2) & n\geq 2 \end{cases}$$

Recursive Fibonacci Algorithm

```
public static long Fibonacci_Recursive(long n)
{
    if (n <= 2)
    {
        return 1;
    }
    else
    {
        return Fibonacci_Recursive(n - 1) + Fibonacci_Recursive(n - 2);
    }
}
```

Explanation

The homogeneous equation of Fibonacci is of the form: $F_n=F_{n-1} + F_{n-2}$

⇨ $F_n - F_{n-1} - F_{n-2} = 0$

Substituting F_n with x^n

⇨ $x^n - x^{n-1} - x^{n-2} = 0$

Common factor (x^{n-2})

⇨ $(x^{n-2})(x^2 - x^1 - 1) = 0$

Taking off the trivial solution x=0 we have a polynomial of the second degree where k = 2 in $x^k - x^{k-1} - 1 = 0$

⇨ $x^2 - x - 1 = 0$

⇨ Solving the 2nd degree equation, we get $\Delta=5$ and $r_1=1.61$ and $r_2=0.62$

The general solution is therefore of the form:

$t_n = c_1 r_1^n + c_2 r_2^n$

The initial condition of Fibonacci gives us:

$c_1 + c_2 = 0$ where n=0

$c_1 r_1 + c_2 r_2 = 1$ where n=1

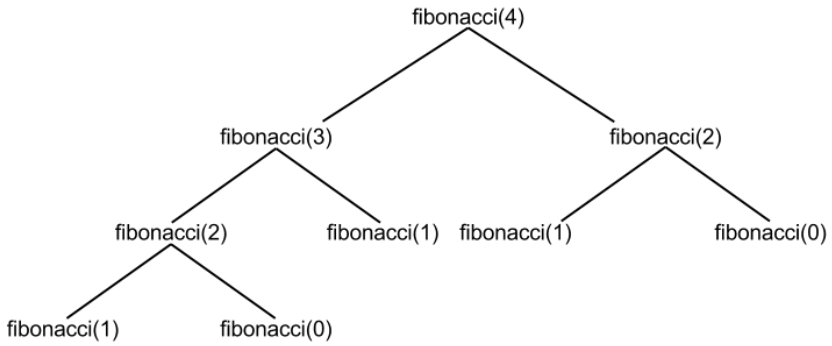
⇨ $c_1 = \frac{1}{\sqrt{5}}$ and $c_2 = -\frac{1}{\sqrt{5}}$

⇨ $t_n = \frac{1}{\sqrt{5}} (r_1^n - r_2^n)$

⇨ $t_n = \frac{1}{\sqrt{5}} ((1.61)^n - (0.62)^n)$

The complexity of the recursive Fibonacci is: $\boldsymbol{O}((\mathbf{1.61})^{\boldsymbol{n}})$

The algorithm Fibonacci_Recursive is very inefficient because it recalculates the same values many times. For instance, to calculate Fibonacci(4) we need the values of Fibonacci(3) and Fibonacci(2), but Fibonacci(3) also calls Fibonacci(2).



In order to avoid wasteful calculations, the problem is solved iteratively. We will cover three algorithms and test their speed.

Iterative Fibonacci Algorithms

```
public static long Fibonacci_Iterative1(long  n)
{
    long i = 1, j = 0, k;

    for (k=1; k<=n; ++k)
    {
        j += i;
        i = j - i;
    }

    return j;
}



public static long Fibonacci_Iterative2(long n)
{
    long i, previous=1, current =1, next=1;

    for(i=3; i<=n; i++)
    {
        next=current+previous;
        previous=current;
        current =next;
    }
    return next;
}
```

```
public static long Fibonacci_Iterative3(long n)
{
    long i=1, j=0, k=0, h=1, t;

    while(n > 0)
    {
        if (n % 2 == 1) // if n is odd
        {
            t = j * h;
            j = (i * h) + (j * k) + t;
            i = (i * k) + t;
        }

        t = h * h;
        h = (2 * k * h) + t;
        k = (k * k) + t;
        n = n / 2;
    }

    return j;
}
```

Explanation

The complexity of the first and second iterative versions is linear since we loop respectively **n** and **(n - 2)** times. Consequently, the complexity is: $\boldsymbol{O(n)}$

Due to the fact that we are dividing n by two as long as n is positive, the complexity of the third iterative version is logarithmic since we loop $\log_2 n$ times. Consequently, the complexity is $\boldsymbol{O(\log_2 n)}$

Classification of algorithms

Virtually all of the algorithms have running time proportional to one of the following functions.

- **1:** this running time is constant, where most instructions are executed once or at most a few times.

- **Log_2N:** The running time of the program is logarithmic. The program gets slightly slower when the size 'n' of the items grows. It occurs when large problems are solved by being transformed into smaller programs. For example, searching a sorted list using the binary search algorithm has a running time of Log2 since the problem is divided by two at every iteration.

- **N:** the running time is linear, when N doubles the running time doubles. For example, searching an unordered list sequentially has a linear running time.

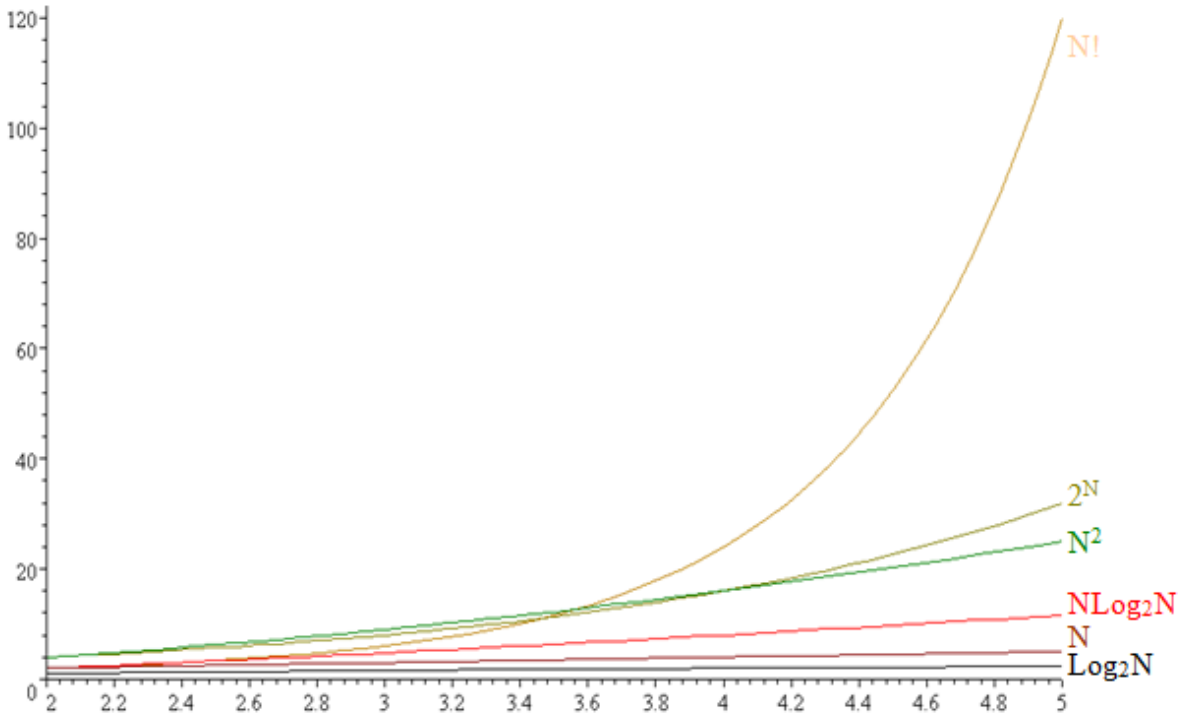
- **$NLog_2N$:** This running time is found for algorithms that solve a problem by breaking it up into smaller sub-problems, solving them independently and then combining the solutions. When N double, the running time is more than double. For example, the merge sort or the quick sort algorithms has a running time of $NLog_2N$, since it has to process all the input and break the input size into a smaller size.

- **N^2:** The running time of the algorithm is quadratic, it is practical to use for relatively small problems. It is found in algorithms that process all pairs of data usually in a double nested loop. For example, the exchange sort, bubble sort and insertion sort has a running time of N^2. When N is a thousand, the running time is a million.

- **N^3**: The running time of the algorithm is cubic, it is practical to use for relatively small problems. It is found in algorithms that process all triples of data usually in a triple nested loop. When N is a hundred , the running time is a million.

- **2^N:** The running time is exponential, such algorithms arise naturally as a brute force solution to problems when the algorithm needs to generate all possible subsets of a set of data. When N is 20, the running time is a million and when N doubles, the running time squares.

- **N!:** The running time is factorial, it is found when an algorithm needs to generate all possible subsets of a set of data.



Growth rate of the complexities