

# Running Time/Complexity Analysis of Algorithms

## Intro to Algorithm Analysis

- We want to know how can we measure the “goodness” of a program/algorithm?
- We have some ways to measure:
  - Running time (the shorter, the better)
  - Memory utilization (the less the better)
  - Amount of code (?)
  - Etc.
- The most commonly used performance measure is the **running time**.
- To measure running time, we can use a stop watch (i.e., use real time as measure). However, there are some problems associated:
  - The same program can have different running time on different computers
  - Different inputs can result in different running time - hard to find their relationship
- So, we need to rely on an more objective measure: count the number of instructions executed by a program for a given input size.
- However, this measure is not practical. In practice, we will count the number of “primitive operations” executed by a program for a given input size.
- Algorithms make repeated steps towards the solution. The **primitive operation** is a step in the algorithm.

```
1  for (int i = 0; i < N; i++) {  
2      S1;  
3      S2;  
4      ...  
5      SN  
6  }
```

The primitive operation of this algorithm consists of the statement S1; S2; ...; SN.

- Principles of Algorithm Analysis
  - Algorithm analysis consists of
    - \* Determine frequency (=count) of primitive operations
    - \* Characterize the frequency as a function of the input size
  - The algorithm analysis must
    - \* Take into account all possible inputs (good ones and bad ones)
    - \* Be independent of hardware/software environment
    - \* Be independent from the programming language
    - \* Give a good estimate that is proportional to the actual running time of the algorithm

- Good inputs, Bad inputs, and Average cases
  - Input data can affect the running time of algorithms
  - The best case are not studied because we cannot count on luck.
  - The worst case gives us an upper bound
    - \* The worst case analysis provides an upper bound on the running time of an algorithm.
    - \* The analysis is easier to do compare to average case analysis.
  - The average case is what we would expect.
    - \* Take the average running time over all possible inputs of the same size
    - \* The analysis depends on input distribution
    - \* The analysis is harder to do because it uses probability techniques.
- Techniques used in Algorithm Analysis
  - There are two main techniques used in Algorithm Analysis:
    - \* Loop analysis
    - \* Recurrence relations
  - A program spends the most amount of time in loops. One of the technique used in algorithm analysis is loop analysis.
  - Some algorithms are recursive. The running time complexity of recursive algorithms are often expressions as recurrence relations. Another technique is solving recurrence relations.

#### Example

$$C(n) = 2 \times C(n/2) + 1$$

## Intro and the Big-Oh Notation

- Consider the following program fragment, how many times is the loop body executed?

```

1  double sum = 0
2  for (int i = 0; i < n; i++) {
3      sum += array[i];
4  }
```

**Solution 1.**  $n$



```

1  double sum = 0
2  for (int i = 0; i < n; i = i + 2) {
3      sum += array[i];
4  }
```

**Solution 2.**  $\frac{n}{2}$



```

1  double sum = 0
2  for (int i = 0; i < n; i++) {
3      for (int j = 0; j < n; ++j) {
4          sum += array[i] * array[j];
5      }
6  }

```

**Solution 3.**  $n \times n = n^2$  □

- The running time in terms of the input size ( $n$ ) can be a general mathematical function. However, we are interested in the **order** of the growth function (but not the exact function).
- **Definition 1 (Approximate Definition of Order, Similar/ $\sim$ ).** Given 2 functions  $f(x)$  and  $g(x)$ , we say  $f(x) \sim g(x)$  if  $\frac{f(x)}{g(x)} = 1$  when  $n \rightarrow \infty$ .

**Remark.** In running time analysis, we can ignore the less significant terms.

**Definition 2 (Precise Definition of Order,  $\mathcal{O}$  notation).** Given two functions  $f(n)$  and  $g(n)$ . The function  $f(n)$  is  $\mathcal{O}(g(n))$  (order of  $g(n)$ ) if  $\exists c, n_0$  s.t.  $f(n) \leq cg(n) \forall n \geq n_0$

**Remark.** A function  $f(n)$  is Big-Oh of  $g(n)$  if  $f(n) \leq cg(n)$  for large values of  $n$ . That is,  $f(n)$  is dominant by some multiple of  $g(n)$  when  $n$  is large.

**Example**  $f(n) = 2n + 10$  is  $\mathcal{O}(n)$ .

**Proof 4.** For  $n > 10$ , we have  $2n + 10 < 3n$ . Therefore, we found  $c = 3$  and  $n_0 = 10$  for which  $f(n) \leq cg(n)$  when  $n \geq n_0$ . ■

However, we know that  $f(n) = n^2$  is not  $\mathcal{O}(n)$ . Picking  $n = c + 1$ , the condition  $n \leq c$  will never be satisfied.

- Big-Oh and Growth rate
  - The Big-Oh notation gives an upper bound on the growth rate of a function  $f(n)$  that represents the run time complexity of some algorithm
  - If  $f(n)$  is  $\mathcal{O}(g(n))$ , then the growth rate of  $f(n)$  is no more than the growth rate of  $g(n)$ .
  - In algorithm analysis, we use  $\mathcal{O}(g(n))$  to rank (=categorize) functions by their growth rate.

**Example**  $2n + 4$ ,  $7n + 9$ ,  $10000n + 999$  are all  $\mathcal{O}(n)$ , so in algorithm analysis we consider all these functions grow at the same rate.

- Common Running Times:

$\mathcal{O}(1)$	Constant Time
$\mathcal{O}(\log(n))$	Logarithmic
$\mathcal{O}(n)$	Linear
$\mathcal{O}(n \log(n))$	Log Linear
$\mathcal{O}(n^2)$	Quadratic

## Useful Formula in Algorithm Analysis

- Triangular Sums:

$$1 + 2 + 3 + 4 + \dots + N = \frac{N(N+1)}{2} \approx \frac{N^2}{2}.$$

- Geometric Sums:

$$1 + 2 + 4 + 8 + \dots + N(= 2^n) = 2N - 1 \approx 2N$$

$$1 + \frac{1}{2} + \frac{1}{4} + \dots + \frac{1}{N}(= \frac{1}{2^n}) = 2 - \frac{1}{N} \approx 2$$

- Harmonic Sum:

$$1 + \frac{1}{2} + \frac{1}{3} + \dots + \frac{1}{N} \approx \ln(N)$$

- Sterling's Approximation:

$$\log(1) + \log(2) + \log(3) + \dots + \log(N) = \log(N!) \approx N \log(N)$$

## Loop Analysis

```
1  for (int i = 0; i < 10; i++) {  
2      doPrimitive();  
3  }
```

- The loop is executed 10 times for any input size.
- Running time = 10 doPrimitive() operations.
- Run time complexity =  $\mathcal{O}(1) \implies$  constant time.

```
1  n = input size; // (e.g.: # elements in an array)  
2  for (int i = 0; i < n; i++) {  
3      doPrimitive();  
4  }
```

- The loop is executed  $n$  times for an input size of  $n$ .
- Running time =  $n$  doPrimitive() operations.
- Run time complexity =  $\mathcal{O}(n) \implies$  linear

**Remark.**  $n$  or  $N$  will always denote the input size in algorithm analysis.

```
1  int sum = 0;  
2  for (int i = 0; i < 5*n; i = i + 4) {  
3      sum = sum + 1;  
4  }
```

- The loop is executed  $\frac{5}{4}n$  times for an input size of  $n$ .

- Running time =  $\frac{5}{4}n$  doPrimitive() operations.

- Run time complexity =  $\mathcal{O}(n) \implies$  linear

```

1  int sum = 0;
2  for (int i = n; i > 0; i = i - 4) {
3      sum = sum + 1;
4  }
```

- The loop is executed  $\frac{1}{4}n$  times for an input size of  $n$ .
- Running time =  $\frac{1}{4}n$  doPrimitive() operations.
- Run time complexity =  $\mathcal{O}(n) \implies$  linear

```

1  int sum = 0;
2  for (int i = 1; i <= n; i = 2*i) {
3      sum++;
4  }
```

- $i$  will take the following numbers in the loop

1   2   4   8   16   32   ...

- Loop will exists when  $i > n$ :

1   2   4   8   16   32   ...    $n$

- Suppose  $2^{k-1} \leq n \leq 2^k$

1   2   4   8   16   32   ...    $2^{k-1}$     $n$     $2^k$

- Iterations:  $k \approx \log n$ . So,  $\mathcal{O}(\log n)$ . ( $n \approx 2^k \iff k \approx \log(n)$ )

```

1  int sum = 0;
2  for (int i = n; i >= 1; i = i/2) {
3      sum++;
4  }
```

- $i$  will take the following numbers in the loop

$n$     $n/2$     $n/4$    ...   1

- Loop will exists when  $i < 1$ .
- Suppose  $n/2^k < 1 < n/2^{k-1}$ .

$n$     $n/2$     $n/4$    ...    $n/2^{k-1}$    1    $n/2^k$

- Iterations  $k \approx \log n$ . So,  $\mathcal{O}(\log n)$  ( $n/2^k \approx 1 \implies n \approx 2^k \implies k \approx \log n$ )

```

1  int sum = 0;
2  for (int i = 0; i < n; i++) {
3      for (int j = 0; j <= i; j++) {
4          sum++;
5      }
6  }

```

$i$	0	2	3	4	...	$n-1$
$j$	*	*	*	*	...	*
		*	*	*	...	*
			*	*	...	*
				*	...	*
					$\vdots$	*
						*

- We sum up those starts. That is adding from 1 to  $n$ .
- Iterations =  $\frac{n(n+1)}{2}$ . So,  $\mathcal{O}(n^2)$ .

```

1  int sum = 0;
2  for (int i = n; i > 0; i = i/2) {
3      for (int j = 0; j < i; j++) {
4          sum++;
5      }
6  }

```

$i$	$n$	$n/2$	$n/4$	$n/8$	...	1
$j$	*	*	*	*	...	*
	*	*	*	*	...	
	*	*	*	*		
	*	*	*			
	*	*				
	*					

- In total, we have  $\log n$   $i$ 's. We add up those starts. That is,  $n + n/2 + n/4 + n/8 + \dots + 1$ .
- By Geometric sum, we have Iteration =  $n(1 + 1/2 + 1/4 + \dots + 1/n) \approx n(2) = 2n$ .
- So,  $\mathcal{O}(n)$ .

```

1  int sum = 0;
2  for (int i = 1; i <= n; i++) {
3      for (int j = 0; j < n; j = j + i) {
4          sum++;
5      }
6  }

```

$i$	1	2	3	4	...	$n$
$j$	0	0	0	0	...	0
	1	2	3	4	...	
	2	4	6	8		
	3	$\vdots$	$\vdots$	$\vdots$		
	4	$n-1$	$n-1$	$n-1$		
	$\vdots$					
	$n-1$					

- When  $i = 1$ , we iterate  $j$  for  $n$  times. When  $i = 2$ , we iterate  $j$  for  $n/2$  times. For an arbitrary  $i$ , we iterate  $j$  for  $n/i$  times.
- So, iteration  $= n + n/2 + n/3 + \dots + n/n = n(1 + 1/2 + 1/3 + \dots + 1/n) \approx n \log(n)$  by the harmonic series. So,  $\mathcal{O}(n \log n)$ .

## Analysis of Recursive Algorithms

```

1  public static void recurse(int n) {
2      if (n == 0) {
3          doPrimitive();
4      } else {
5          doPrimitive();
6          recurse(n-1);
7      }
8  }
```

- Let  $C(n) = \#$  of times that `doPrimitive()` is executed when input  $= n$ .
- $C(0) = 1$  because when  $n = 0$ , we only execute `doPrimitive()` one time and terminate. This is the base case.
- $C(n) = 1 + C(n-1)$  for  $n > 0$ . This is because `recurse(n)` will invoke `recurse(n-1)`, and by definition, the  $\#$  times that `doPrimitive()` is executed when input  $= n-1$  is:  $C(n-1)$
- To solve the recursive relation, we will use the technique **telescoping**.

$$\begin{aligned}
 C(n) &= 1 + C(n-1) \\
 &= 1 + 1 + C(n-2) \\
 &= 1 + 1 + 1 + C(n-3) \\
 &= \underbrace{1 + 1 + 1 + \dots + 1}_{n \text{ times}} + C(0) \\
 &= n + 1
 \end{aligned}$$

- So,  $\mathcal{O}(n)$

```

1  public static void recurse(int n) {
2      if (n == 0) {
3          doPrimitive();
4      } else {
5          for (int i = 0; i < n; i++) {
6              doPrimitive();
7          }
8          recurse(n-1);
9      }
10 }

```

- Let  $C(n)$  = # of times that `doPrimitive()` is executed when input =  $n$ .
- $C(0) = 1$  as the base case;  $C(n) = n + C(n - 1)$  for  $n > 0$  because `recurse(n)` will invoke `recurse(n-1)`, and by definition, the # times that `doPrimitive()` is executed when input =  $n - 1$  is:  $C(n - 1)$ .
- Telescoping:

$$\begin{aligned}
 C(n) &= n + C(n - 1) \\
 &= n + (n - 1) + C(n - 2) \\
 &= n + (n - 1) + (n - 2) + C(n - 3) \\
 &= n + (n - 1) + (n - 2) + \dots + 1 + C(0) \\
 &= \frac{n(n + 1)}{2} + 1
 \end{aligned}$$

- So,  $\mathcal{O}(n^2)$ .

```

1  public static void recurse(int[] A, int a, int b) {
2      if (b-a <= 1) {
3          doPrimitive();
4      } else {
5          doPrimitive();
6          recurse(A, a, (a+b)/2); // First half of array
7          recurse(A, (a+b)/2, b); // 2nd half of array
8      }
9  }

```

- Let  $C(n)$  = # of times that `doPrimitive()` is executed when input =  $n$ .
- $C(1) = 1$  because if  $b - a \leq 1$ , it executes 1 `doPrimitive()`.
- $C(n) = 1 + C(n/2) + C(n/2) = 1 + 2C(n/2)$  for  $n > 0$  because: `recurse(n)` will invoke `recurse()` twice with input size  $n/2$ , and by definition, the # times that `doPrimitive()` is executed when input =  $n/2$  is:  $C(n/2)$ .



- Telescoping:

$$\begin{aligned}
 C(n) &= 1 + 2C(n/2) \\
 &= 1 + 2 + 4C(n/4) \\
 &= 1 + 2^1 + 2^2 + \dots + 2^k * C(n/2^k)
 \end{aligned}$$

We want  $C(n/2^k)$  to eventually be  $C(1)$ . So, we have  $1 = n/2^k \implies n = 2^k \implies k = \log n$ . So,

$$\begin{aligned}
 C(n) &= 1 + 2^1 + 2^2 + \dots + 2^k C(1) \\
 &= 1 + 2^1 + 2^2 + \dots + 2^k \\
 2C(n) &= 2^1 + 2^2 + 2^3 + \dots + 2^{k+1} \\
 C(n) = 2C(n) - C(n) &= 2^{k+1} - 1 \\
 &= 2^{\log n + 1} - 1 \\
 &= 2 \cdot 2^{\log n} - 1 \\
 &= 2n - 1
 \end{aligned}$$

- So,  $\mathcal{O}(n)$ .