PMDS605L - Data Structures and Algorithms

Dr. B.S.R.V. Prasad
Department of Mathematics
School of Advanced Sciences
Vellore Institute of Technology
Vellore

srvprasad.bh@gmail.com (Personal)
srvprasad.bh@vit.ac.in (Official)
+91-8220417476

Problem Development Steps

The following steps are involved in solving computational problems.

- Problem definition
- Development of a model
- Specification of an Algorithm
- Designing an Algorithm
- Checking the correctness of an Algorithm
- Analysis of an Algorithm
- Implementation of an Algorithm
- Program testing
- Documentation

Characteristics of Algorithms

The main characteristics of algorithms are as follows —

- Algorithms must have a unique name
- Algorithms should have explicitly defined set of inputs and outputs
- Algorithms are well-ordered with unambiguous operations
- Algorithms halt in a finite amount of time. Algorithms should not run for infinity, i.e., an algorithm must end at some point

Pseudocode

- **Pseudocode** gives a high-level description of an algorithm without the ambiguity associated with plain text but also without the need to know the syntax of a particular programming language.
- The running time can be estimated in a more general manner by using Pseudocode to represent the algorithm as a set of fundamental operations which can then be counted.

Difference between Algorithm and Pseudocode

- An algorithm is a formal definition with some specific characteristics that describes a process, which could be executed by a Turing-complete computer machine to perform a specific task.
- Generally, the word “algorithm” can be used to describe any high level task in computer science.
- Pseudocode is an informal and (often rudimentary) human readable description of an algorithm leaving many granular details of it.
- Writing a pseudocode has no restriction of styles and its only objective is to describe the high level steps of algorithm in a much realistic manner in natural language.

Difference between Algorithm and Pseudocode Insertion Sort

Algorithm: Insertion-Sort

```
Input: A list L of integers of length n
Output: A sorted list L1 containing those integers present in L
Step 1: Keep a sorted list L1 which starts off empty
Step 2: Perform Step 3 for each element in the original list L
Step 3: Insert it into the correct position in the sorted list L1.
Step 4: Return the sorted list
Step 5: Stop
```

Difference between Algorithm and Pseudocode Insertion Sort

Below is a pseudocode which describes how the high level abstract process for the Insertion-Sort algorithm:

```
for i <- 1 to length(A)
   x <- A[i]
   j <- i
   while j > 0 and A[j-1] > x
      A[j] <- A[j-1]
      j <- j - 1
   A[j] <- x
```

Algorithm Design

- The important aspects of algorithm design include creating an efficient algorithm to solve a problem in an efficient way using minimum time and space.
- To solve a problem, different approaches can be followed.
  - Few can be efficient with respect to time consumption
  - Few other approaches may be memory efficient
- Both time consumption and memory usage cannot be optimized simultaneously.
- If we require an algorithm to run in lesser time, we have to invest in more memory and if we require an algorithm to run with lesser memory, we need to have more time.

Analysis of Algorithms

- In theoretical analysis of algorithms, it is common to estimate their complexity in the asymptotic sense, i.e., to estimate the complexity function for arbitrarily large input.
- This process is known as “analysis of algorithms” and this term was coined by Donald Knuth.
- Algorithm analysis —
  - is an important part of computational complexity theory.
  - provides theoretical estimation for the required resources of an algorithm to solve a specific computational problem.
- Most algorithms are designed to work with inputs of arbitrary length.
- Analysis of algorithms is the determination of the amount of time and space resources required to execute it.

Analysis of Algorithms

- Usually, the efficiency or running time of an algorithm is stated as a function relating the input length to
  - the number of steps, known as **time complexity**
  - volume of memory, known as **space complexity**

The Need for Analysis

- One computational problem can be solved by different algorithms.
- How to choose a better algorithm for a particular problem?
- By considering an algorithm for a specific problem, we can begin to develop *pattern recognition* so that similar types of problems can be solved by the help of this algorithm.

The Need for Analysis

- Algorithms are often quite different from one another, though the objective of these algorithms are the same.
  - For example, we know that a set of numbers can be sorted using different algorithms.
- Number of comparisons performed by one algorithm may vary with others for the same input.
- Hence, time complexity of those algorithms may differ.
- At the same time, we need to calculate the memory space required by each algorithm.

The Need for Analysis

- As stated earlier analysis of algorithm is the process of analyzing the problem-solving capability of the algorithm in terms of the time and size required.
- However, the main concern of analysis of algorithms is the required time or performance.
- Generally, we perform the following types of analysis —

  Worst-case — The maximum number of steps taken on any instance of size n.

  Average case — An average number of steps taken on any instance of size n.

  Best-case — The minimum number of steps taken on any instance of size n.

  Amortized — A sequence of operations applied to the input of size a averaged over time.

The Need for Analysis
Time Complexity

Worst-case time complexity: For ‘n’ input size, the worst-case time complexity can be defined as the maximum amount of time needed by an algorithm to complete its execution. Thus, it is nothing but a function defined by the maximum number of steps performed on an instance having an input size of n.

Average case time complexity: For ‘n’ input size, the average-case time complexity can be defined as the average amount of time needed by an algorithm to complete its execution. Thus, it is nothing but a function defined by the average number of steps performed on an instance having an input size of n.

The Need for Analysis
Time Complexity

Best case time complexity: For ‘n’ input size, the best-case time complexity can be defined as the minimum amount of time needed by an algorithm to complete its execution. Thus, it is nothing but a function defined by the minimum number of steps performed on an instance having an input size of n.

Asymptotic Analysis of Algorithms

- Execution time of an algorithm depends on the instruction set, processor speed, disk I/O speed, etc.
- Hence, we estimate the efficiency of an algorithm asymptotically.
- The word **asymptotic** means approaching a value or curve arbitrarily closely (i.e., as some sort of limit is taken).
- **Asymptotic analysis** is a technique of representing limiting behavior.
- This methodology has applications across science and can be used to analyze the performance of an algorithm for some large data set.

Example of asymptotic Analysis:

- Consider the function $f(n) = n^2 + 3n$.
- While, analysing the asymptotic nature, the term $3n$ becomes insignificant compared to n^2 when n is very large.
- The function $f(n)$ is said to be asymptotically equivalent to n^2 as $n \to \infty$, and here is written symbolically as $f(n) \approx n^2$.

Asymptotic Notations

- Asymptotic notations are used to write fastest and slowest possible running time for an algorithm.
- These are also referred to as ‘best case’ and ‘worst case’ scenarios respectively.
- In asymptotic notations, we derive the complexity concerning the size of the input (example in terms of *n*).
- Asymptotic notations are important because:
  1. They give simple characteristics of an algorithm’s efficiency.
  2. They allow the comparisons of the performances of various algorithms.

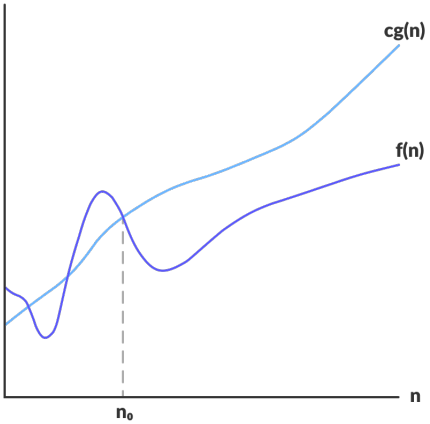
Asymptotic Notations

- Time function of an algorithm is represented by $T(n)$, where n is the input size.
- Different types of asymptotic notations are used to represent the complexity of an algorithm.
- Following asymptotic notations are used to calculate the running time complexity of an algorithm.
  - O — Big Oh
  - Ω — Big omega
  - Θ — Big theta
  - o — Little Oh
  - ω — Little omega

O: Asymptotic Upper Bound

- 'O' (Big Oh) is the most commonly used notation.
- Big Oh notation represents the upper bound of the running time of an algorithm and thus, gives the worst-case complexity of an algorithm.
- A function $f(n)$ can be represented in the order of $g(n)$ that is $O(g(n))$, if there exists a value of positive integer n_o and a positive constant c such that $f(n) \leq cg(n)$ for all $n \geq n_o$
- In other words, the function $g(n)$ is an upper bound for function $f(n)$, and $g(n)$ grows faster than $f(n)$.

O: Asymptotic Upper Bound



f(n) = O(g(n))

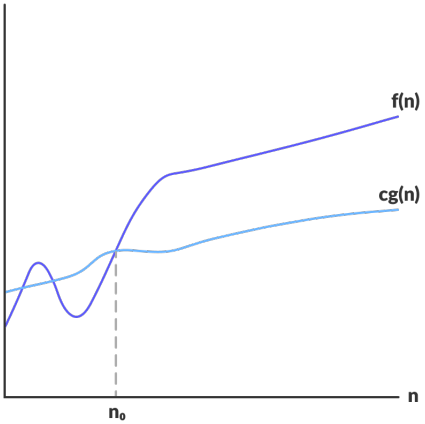
O: Asymptotic Upper Bound
Example

- Let us consider a given function, $f(n) = 4n^3 + 10n^2 + 5n + 1$.
- Considering $g(n) = n^3$, we have $f(n) \leq 5g(n)$ for all $n > 2$.
- Hence, the complexity of $f(n)$ can be represented as $O(g(n))$, i.e. $O(n^3)$.

Ω: Asymptotic Lower Bound

- Big Omega notation represents the lower bound of the running time of an algorithm and thus, provides the best case complexity of an algorithm.
- We say that $f(n) = \Omega(g(n))$ if there exists constant c and n_o such that $f(n) \geq cg(n)$ for all $n \geq n_o$.
- In other words, the function g is a lower bound for the function f after a certain value of n.

Ω: Asymptotic Lower Bound



$f(n) = \Omega(g(n))$

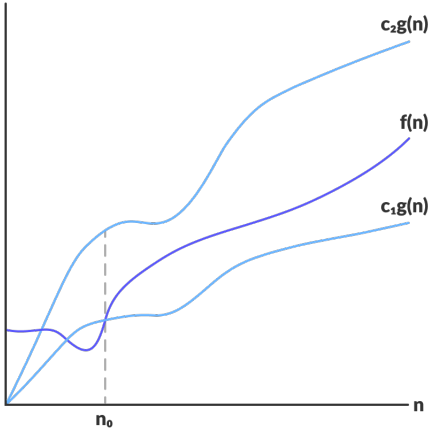
Ω: Asymptotic Lower Bound
Example

- Let us consider a given function, $f(n) = 4n^3 + 10n^2 + 5n + 1$.
- Considering $g(n) = n^3$, we have $f(n) \geq 4g(n)$ for all $n > 0$.
- Hence, the complexity of $f(n)$ can be represented as $\Omega(g(n))$, i.e. $\Omega(n^3)$.

Θ: Asymptotic Tight Bound

- Theta notation encloses the function from above and below.
- Since it represents the upper and the lower bound of the running time of an algorithm, it is used for analyzing the average-case complexity of an algorithm.
- We say that $f(n) = \Theta(g(n))$ when there exist constants c_1, c_2 and n_o that $c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all $n \geq n_o$.
- This means function g is a tight bound for function f.

Θ: Asymptotic Tight Bound



f(n) = Θ(g(n))

Θ: Asymptotic Tight Bound
Example

- Let us consider a given function, $f(n) = 4n^3 + 10n^2 + 5n + 1$.
- Considering $g(n) = n^3$, we have $4g(n) \leq f(n) \leq 5g(n)$ for all $n > 2$.
- Hence, the complexity of $f(n)$ can be represented as $\Theta(g(n))$, i.e. $\Theta(n^3)$.

$o-$notation

- The asymptotic upper bound provided by $O-$notation may or may not be asymptotically tight.
- The bound $2n^2 = O(n^2)$ is asymptotically tight, but the bound $2n = O(n^2)$ is not.
- So, we use $o-$notation to denote an upper bound that is not asymptotically tight.
- We formally define $o(g(n))$ (little-oh of g of n) as the set $f(n) = o(g(n))$ for any positive constant $c > 0$ and there exists a value $n_0 > 0$, such that $0 \leq f(n) \leq c.g(n)$.
- Intuitively, in the $o-$notation, the function $f(n)$ becomes insignificant relative to $g(n)$ as n approaches infinity; that is,

$$\lim_{n \to \infty} \left(\frac{f(n)}{g(n)} \right) = 0.$$

o—notation

- Example:
  Let us consider the same function, $f(n) = 4n^3 + 10n^2 + 5n + 1$
  Considering $g(n) = n^4$,

$$\lim_{n \to \infty} \left(\frac{4n^3 + 10n^2 + 5n + 1}{n^4} \right) = 0.$$

  Hence, the complexity of $f(n)$ can be represented as $o(g(n))$, i.e. $o(n^4)$.

$\omega-$notation

- We use $\omega-$notation to denote a lower bound that is not asymptotically tight.
- Formally, we define $\omega(g(n))$ (little-omega of g of n) as the set $f(n) = \omega(g(n))$ for any positive constant $c > 0$ and there exists a value $n_0 > 0$, such that $0 \leqslant cg(n) < f(n)$.
- The relation $f(n) = \omega(g(n))$ implies that the following limit exists

$$\lim_{n\to\infty} \left(\frac{f(n)}{g(n)}\right) = \infty$$

  That is, $f(n)$ becomes arbitrarily large relative to g(n) as n approaches infinity.
- So, $\frac{n^2}{2} = \omega(n)$, but $\frac{n^2}{2} \neq \omega(n^2)$.

$\omega-$notation

- Example:
  Let us consider same function, $f(n) = 4n^3 + 10n^2 + 5n + 1$
  Considering $g(n) = n^2$,

$$\lim_{n\to\infty}\left(\frac{4.n^3 + 10.n^2 + 5.n + 1}{n^2}\right) = \infty$$

  Hence, the complexity of $f(n)$ can be represented as $o(g(n))$, i.e. $\omega(n^2)$.

Typical Complexities of an Algorithm

- If f corresponds to the function whose size is the same as that of input data of an algorithm then $O(f)$ notation represents the complexity of an algorithm.
- The complexity of the asymptotic computation $O(f)$ determines in which order the resources such as CPU time, memory, etc. are consumed by the algorithm that is articulated as a function of the size of the input data.
- The complexity can be found in any form such as
  - constant,
  - logarithmic,
  - linear,
  - $n \log(n)$,
  - quadratic,
  - cubic,
  - exponential, etc.
- To make it even more precise, we often call the complexity of an algorithm as “running time”.

$O(1)$ — Constant Time

- $O(1)$ describes algorithms that take the same amount of time to compute regardless of the input size.
- For instance, if a function takes the same time to process ten elements and 1 million items, then we say that it has a constant growth rate or $O(1)$.
- Examples:
  - Find if a number is even or odd.
  - Check if an item on an array is null.
  - Print the first element from a list.
  - Find a value on a map.

$O(n)$ — Linear Time

- Linear running time algorithms imply that the program visits every element from the input.
- Linear time complexity $O(n)$ means that the algorithms take proportionally longer to complete as the input grows.
- Linear time algorithms are widespread. Examples:
  - Get the max/min value in an array.
  - Find a given element in a collection.
  - Print all the values in a collection.

$O(n^2)$ — Quadratic Time

- A function with a quadratic time complexity has a growth rate of n^2 i.e., if the input is size n, it will do n^2 operations.
- For example if $n = 2$ then it will perform four operations. If the input size is 8, the algorithm will take 64 operations, and so on.
- Examples:
  - Check if a collection has duplicated values.
  - Sorting items in a collection using bubble sort, insertion sort, or selection sort.
  - Find all possible ordered pairs in an array.

$O(n^c)$ — Polynomial Time

- Polynomial running time is represented as $O(n^c)$, when $n > 1$.
- As we already saw, two inner loops almost translate to $O(n^2)$ since it has to go through the array twice in most cases.
- If the number of iterations increases then the algorithm complexity increases in polynomial time i.e., quadratic, cubic, $n^c (c \geq 4)$ etc.
- Polynomial running time algorithms take longer time to compute as the input grows fast.
- As a reason we want to stay away from polynomial running time algorithms.
- However these algorithms are not the worst.
- Example: Multiplication of two matrices.

$O(\log n)$ — Logarithmic Time

- Logarithmic time complexities usually apply to algorithms that divide problems in half every time.
- For instance, let's say that we want to look for a phone number in a telephone dictionary.
- As you know, this book has every word sorted alphabetically, we can find a number in two ways:

  Algorithm A Start on the first page of the book and go sequentially till you find the number

  Algorithm B
  1. Open the book in the middle and check for the entry
  2. If the work you are looking for is alphabetically more significant, then look to the right. Otherwise, look in the left half
  3. Divide the remainder in half again, and repeat Step #2 until you find the number you are looking for.

$O(\log n)$ — Logarithmic Time

- Logarithmic time complexities usually apply to algorithms that divide problems in half every time.
- For instance, let's say that we want to look for a phone number in a telephone dictionary.
- As you know, this book has every word sorted alphabetically, we can find a number in two ways:

  Algorithm A Start on the first page of the book and go sequentially till you find the number $O(n)$

  Algorithm B
  1. Open the book in the middle and check for the entry
  2. If the work you are looking for is alphabetically more significant, then look to the right. Otherwise, look in the left half
  3. Divide the remainder in half again, and repeat Step #2 until you find the number you are looking for. $O(\log n)$

$O(n \log n)$ — Linearithmic Time

- Linearithmic time complexity is slightly slower than a linear algorithm.
- However, it's still much better than a quadratic alogrithm
- Examples;
  - Efficient sorting algorithms like merge sort, quicksort etc.

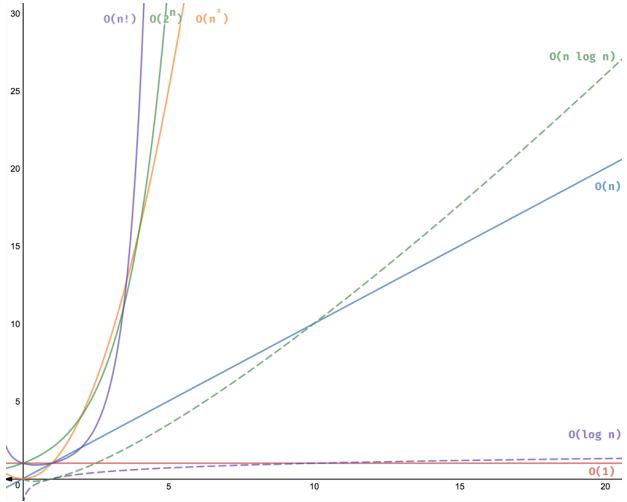
$O(2^n)$ — Exponential (base 2) Time

- Exponential (base 2) running time means that the calculations performed by an algorithm double every time as the input grows.
- Examples:
  - Power Set: finding all the subsect on a set.
  - Fibanacci Sequence.
  - Travelling salesman problem using dynamic programming.

$O(n!)$ — Factorial Time

- Factorial time algorithm takes $n!$ factorial time for an input size of n.
- As factorial of a number increases rapidly as n increases, one must stay away from the factorial time algorithms.
- Examples:
  - Permutations of a string.
  - Solving the travelling salesman problem with a brute-force search.

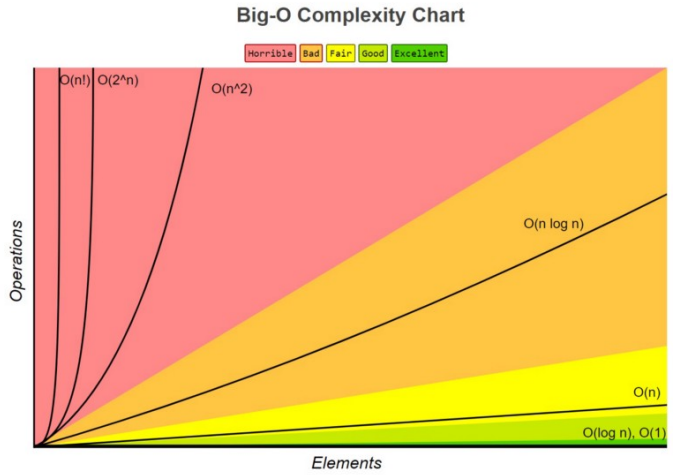
Graph representing the running complexities



Summary

Big O Notation	Name	Example(s)
O(1)	Constant	# Odd or Even number, # Look-up table (on average)
O(log n)	Logarithmic	# Finding element on sorted array with binary search
O(n)	Linear	# Find max element in unsorted array, # Duplicate elements in array with Hash Map
O(n log n)	Linearithmic	# Sorting elements in array with merge sort
$O(n^2)$	Quadratic	# Duplicate elements in array **(naïve)**, # Sorting array with bubble sort
$O(n^3)$	Cubic	# 3 variables equation solver
$O(2^n)$	Exponential	# Find all subsets
O(n!)	Factorial	# Find all permutations of a given set/string

Graph representing the running complexities



Exercise

Exercise

An image is represented by a 2D array of pixels. If you use a nested for loop to iterate through every pixel (that is, you have a for loop going through all the columns, then another for loop inside to go through all the rows), what is the time complexity of the algorithm when the image is considered as the input?

Exercise

Exercise

An image is represented by a 2D array of pixels. If you use a nested for loop to iterate through every pixel (that is, you have a for loop going through all the columns, then another for loop inside to go through all the rows), what is the time complexity of the algorithm when the image is considered as the input?

At first we think that the complexity of above algorithm is $O(n^2)$ as we are using nested loop.

Exercise

Exercise

An image is represented by a 2D array of pixels. If you use a nested for loop to iterate through every pixel (that is, you have a for loop going through all the columns, then another for loop inside to go through all the rows), what is the time complexity of the algorithm when the image is considered as the input?

At first we think that the complexity of above algorithm is $O(n^2)$ as we are using nested loop.

However, n is supposed to be the input size and since the image array is input, and every pixel was iterated through only once, the correct answer is $O(n)$

Determining Complexity Odeer
Few Rules...

1. $O(1)$ has the least complexity
   - Often called “constant time”, if you can create an algorithm to solve the problem in $O(1)$, you are probably at your best.
   - In some scenarios, the complexity may go beyond $O(1)$, then we can analyze them by finding its $O(1/g(n))$ counterpart. For example, $O(1/n)$ is more complex than $O(1/n^2)$.
2. $O(\log(n))$ is more complex than $O(1)$, but less complex than polynomials
   - As complexity is often related to divide and conquer algorithms, $O(\log(n))$ is generally a good complexity you can reach for sorting algorithms.
   - $O(\log(n))$ is less complex than $O(\sqrt{n})$, because the square root function can be considered a polynomial, where the exponent is 0.5.
3. Complexity of polynomials increases as the exponent increases
   - For example, $O(n^5)$ is more complex than $O(n^4)$.

Determining Complexity Odeer
Few Rules...

4. Exponentials have greater complexity than polynomials as long as the coefficients are positive multiples of n
   - $O(2^n)$ is more complex than $O(n^{99})$, but $O(2^n)(n < 0)$ is actually less complex than $O(1)$.
   - We generally take 2 as base for exponentials and logarithms because things tends to be binary in Computer Science, but exponents can be changed by changing the coefficients.
   - If not specified, the base for logarithms is assumed to be 2.
5. Factorials have greater complexity than exponentials
   - As factorial is the analytic continuation of the Gamma function, we have factorials have greater complexity than exponentials.
   - Factorials and exponentials have the same number of multiplications, but the numbers that get multiplied grow for factorials, while remaining constant for exponentials.
6. Multiplying terms
   - When multiplying, the complexity will be greater than the original, but no more than the equivalence of multiplying something that is more complex.
   - For example, $O(n * \log(n))$ is more complex than $O(n)$ but less complex than $O(n^2)$, because $O(n^2) = O(n * n)$ and n is more complex than $\log(n)$.

Exercise Problem

Exercise

Rank the following complexities:

$$(\sqrt{2})^{\log n} \quad n \quad n^{100} \quad n^2 \quad \sqrt{n} \quad n \quad n^3 \quad n \log n \quad 4^{\log n} \quad 2^{100 \log n} \quad 2^{\log^{1.001} n}$$

Exercise Problem

Exercise

$$(\sqrt{2})^{\log n}, \quad \sqrt{n}, \quad 2^{\log^{1.001} n}, \quad n, \quad n \log n, \quad 4^{\log n}, \quad n^2, \quad n^3, \quad 2^{100 \log n}, \quad n^{100}$$

Exercise Problem

Exercise

Consider the following example, where an integer i in a for loop running from i equals 1 to n to print the word "Hello"?

```
A()
{
    int i;
    for (i=1 to n)
        printf("Hello");
}
```

What is the complexity of the above algorithm?

Exercise Problem

Exercise

Consider the following example, where an integer i in a for loop running from i equals 1 to n to print the word "Hello"?

```
A()
{
    int i;
    for (i=1 to n)
        printf("Hello");
}
```

What is the complexity of the above algorithm?

Since *i* equals 1 to *n*, so the above program will print Hello, *n* number of times. Thus, the complexity will be $O(n)$.

Exercise Problem

Exercise

What is the complexity of the following problem?

```
A()
{
    int i, j:
    for (i=1 to n)
        for (j=1 to n)
            printf("Hello");
}
```

Exercise Problem

Exercise

What is the complexity of the following problem?

```
A()
{
    int i, j:
    for (i=1 to n)
        for (j=1 to n)
            printf("Hello");
}
```

The outer loop runs n times and the inner loop runs n for each instance of the inner loop. Thus, the time complexity is $O(n^2)$

Exercise Problem

Exercise

What is the complexity of the following?

```
int a = 0, b = 0;
for (i = 0; i < N; i++) {
    a = a + rand();
}
for (j = 0; j < M; j++) {
    b = b + rand();
}
```

Exercise Problem

Exercise

What is the complexity of the following?

```
int a = 0, b = 0;
for (i = 0; i < N; i++) {
    a = a + rand();
}
for (j = 0; j < M; j++) {
    b = b + rand();
}
```

$O(N + M)$

The first loop is $O(N)$ and the second loop is $O(M)$. Since we don't know which is bigger, we say this is $O(N + M)$. This can also be written as $O(max(N, M))$.

Exercise Problem

Exercise

What is the complexity of the following?

```
int a = 0;
for (i = 0; i < N; i++) {
    for (j = N; j > i; j--) {
        a = a + i + j;
    }
}
```

Exercise Problem

Exercise

What is the complexity of the following?

```
int a = 0;
for (i = 0; i < N; i++) {
    for (j = N; j > i; j--) {
        a = a + i + j;
    }
}
```

$O(N^2)$

The above code runs total no of times

$N + (N-1) + (N-2) + \ldots + 1 + 0 = \frac{N(N+1)}{2} = \frac{1}{2}N^2 + \frac{1}{2}N = O(N^2)$ times.

Exercise Problem

Exercise

What is the complexity of the following?

```
int i, j, k = 0;
for (i = n / 2; i <= n; i++) {
    for (j = 2; j <= n; j = j * 2) {
        k = k + n / 2;
    }
}
```

Exercise Problem

Exercise

What is the complexity of the following?

```
int i, j, k = 0;
for (i = n / 2; i <= n; i++) {
    for (j = 2; j <= n; j = j * 2) {
        k = k + n / 2;
    }
}
```

j keeps doubling till it is less than or equal to n. Number of times, we can double a number till it is less than n would be $\log(n)$.
For example if $n = 16$ then $j = 2, 4, 8, 16$ and if $n = 32$, then $j = 2, 4, 8, 16, 32$.
So, j would run for $O(\log n)$ steps and i runs for $n/2$ steps.
So, total steps = $O(n/2 * \log(n)) = O(n \log n)$