# EE2331 Data Structures and Algorithms:

## Analysis of Algorithms

# Analysis of Algorithms

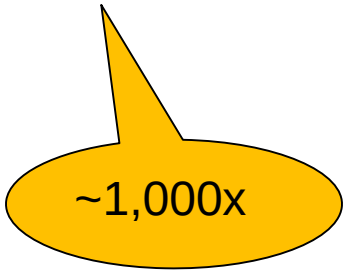
- Often several different algorithms are available to solve the same problem. These algorithms may not run with same efficiency.
  - May be impractical for large input size
  - May run extremely slow for particular inputs
- We want to know the efficiency and complexity of algorithms so as to compare them and make a wise choice.
- The complexity growth rate is far more important than the actual execution time during analysis.

# Algorithms Analysis Example

- Find the sum of 1 + 2 + 3 + 4 + … + 998 + 999

- Method 1:
  - 1 + 2 = 3
  - 3 + 3 = 6
  - 6 + 4 = 10
  - …
  - 498,501 + 999 = 499,500

998 addition




- Method 2:
  - (1 + 999) x 999 / 2
  - = 499,500

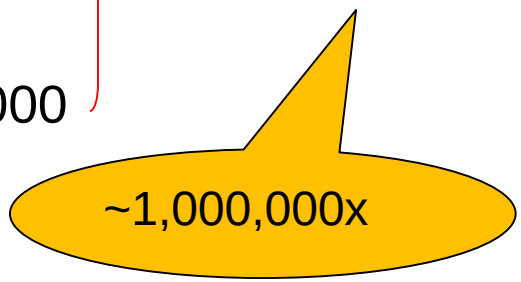
1 addition, 1 multiplication, 1 division

# Algorithms Analysis Example

- Find the sum of 1 + 2 + 3 + 4 + … + 999,999

- Method 1:
  - 1 + 2 = 3
  - 3 + 3 = 6
  - 6 + 4 = 10
  - …
  - 498,998,500,001 + 999,999 = 499,999,500,000

999,998 addition!




- Method 2:
  - (1 + 999,999) x 999,999 / 2
  - = 499,999,500,000

Still 1 addition, 1 multiplication, 1 division! (independent of the input size)

# Algorithms Analysis Example

Method 1:

```
int sumOfSeries(int n) {
    int i, sum = 0;
    for (i = 1; i < n; i++)
        sum += i;
    return sum;
}
```

*n* - 1 addition

**Which one is better?**

Method 2:

```
int sumOfSeries(int n) {
    return (1 + n) * n / 2;
}
```

1 addition, 1 multiplication, 1 division

# Complexity of Algorithms

# Two Types of Complexity

- Complexity
  - Time Complexity (time requirement)
  - Space Complexity (memory space requirement)
- Usually the problems have a natural “size” called ***n***
  - i.e. the amount of data to be processed
- Describe the resources used as a function of *n*
  - i.e. the amount of time taken, the amount of memory required
- Usually interested in **average case** and **worse case**
  - Average case: the amount of time/memory expected to take on typical input data
  - Worse case: the amount of time/memory would take on the worst possible input

# Time/Space Complexity

- Fixed part (ignored)
  - Time for declaring variables, assigning values
  - Space for instruction, constant, variables
- Variable part
  - Loops that run according to the input size
  - Memory allocation according to the input size
- The time requirement $T(P)$ of program $P$ is
  - $T(P) = C + Tp(n)$
  - $C$ is the fixed part
  - $Tp(n)$ is the variable part depends on problem size $n$
- To analyze time complexity, concentrate solely on the estimation of $Tp(n)$
- Space complexity $S(P)$ can be estimated in the same way

# An Example Program

```
#include <iostream>

int main(int argc, char *argv[]) {

    int i, n, sum = 0;

    cin >> n;

    for(i = 0; i < n; i++)
        sum += i;

    return 0;
}
```

Constant time, $C_1$

Constant time, $C_2$

Variable time, depends on $n$
$= C_3 \times n$

Constant time, $C_4$

Total execution time
$= C_1 + C_2 + C_3 \times n + C_4$
$\approx C_3 \times n$ (if $n$ is very large)

# Analysis

- The exact value of $C_i$ is not important, but the order of magnitude is important
- Usually $C_i$ is a very small number
- $n * C_i$ would be a very significant number if $n$ is a very large number
- e.g. suppose $C_i$ is 1ms
  - If $n$ is 1, execution time is 1ms
  - If $n$ is 10, execution time is 10ms
  - If $n$ is 1 million, execution time is 1,000s
- $C_i$ is machine dependent
- To simplify our analysis, simply count how many times each instruction is executed in the algorithm.

# Count the No. of Operations

```
int i, n, sum = 0;
```

This instruction being executed once

```
cin >> n;
```

This instruction being executed once

```
for(i = 0; i < n; i++)
    sum += i;
```

This block being executed *n* times

Total execution
= 1 + 1 + *n*
= *n* + 2
≈ *n* (if *n* is very large)

# Count the No. of Operations

```
//Code A
sum += i;
```

This instruction being executed once

```
//Code B
for(i = 0; i < n; i++)
        sum += i;
```

This code being executed $n$ times

```
//Code C
for(i = 0; i < n; i++)
    for(j = 0; j < n; j++)
        sum += i * j;
```

This code being executed $n^2$ times!

# Asymptotic Complexity

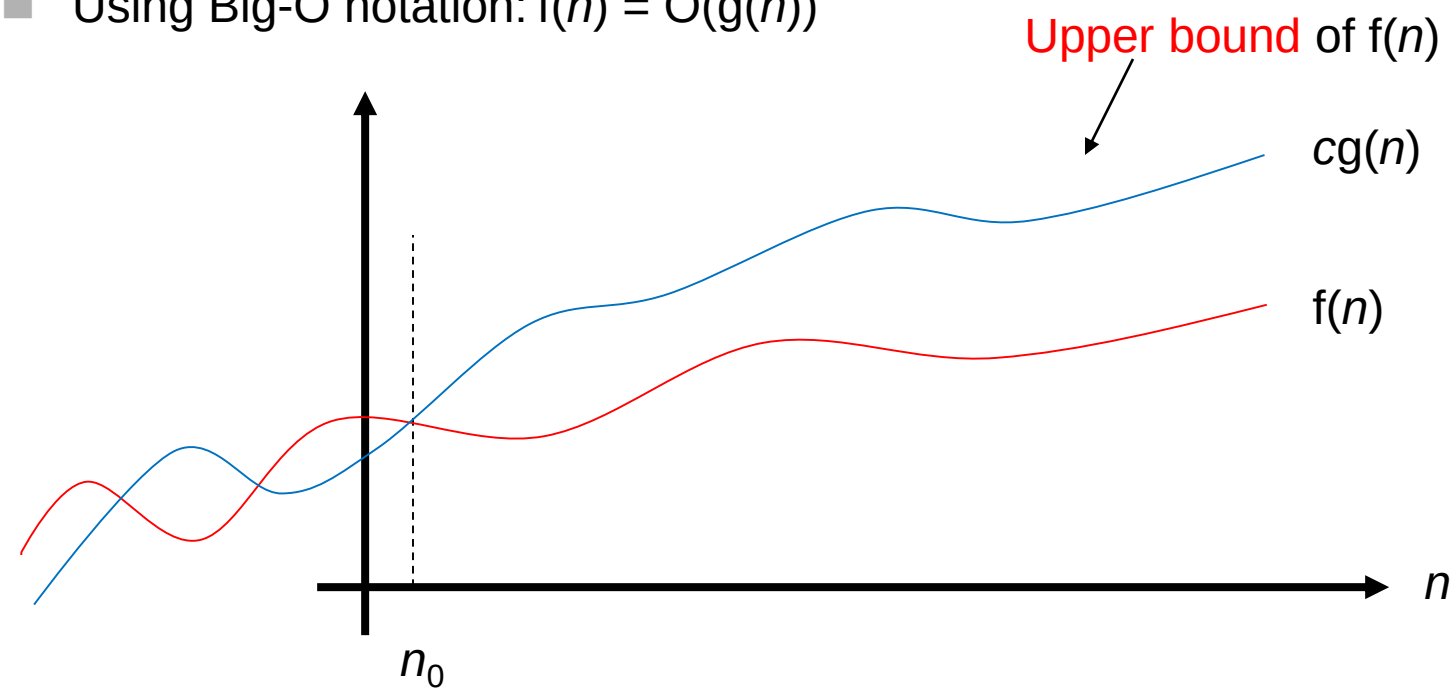
- Asymptotic complexity is a way of expressing the main component of the cost of an algorithm.
- For example, when analyzing some algorithm, one might find that the time (or the number of steps) it takes to complete a problem of size n is given by

  $T(n) = 4n^2 - 2n + 2$
- If we ignore constants (which makes sense because those depend on the particular hardware the program is run on) and slower growing terms (i.e. 2n), we could say T(n) grows at the order of $n^2$ and write:

  $T(n) = O(n^2)$
- The letter O is used because the rate of growth of a function is also called its *Order*. Basically, it tells you how fast a function grows or declines.

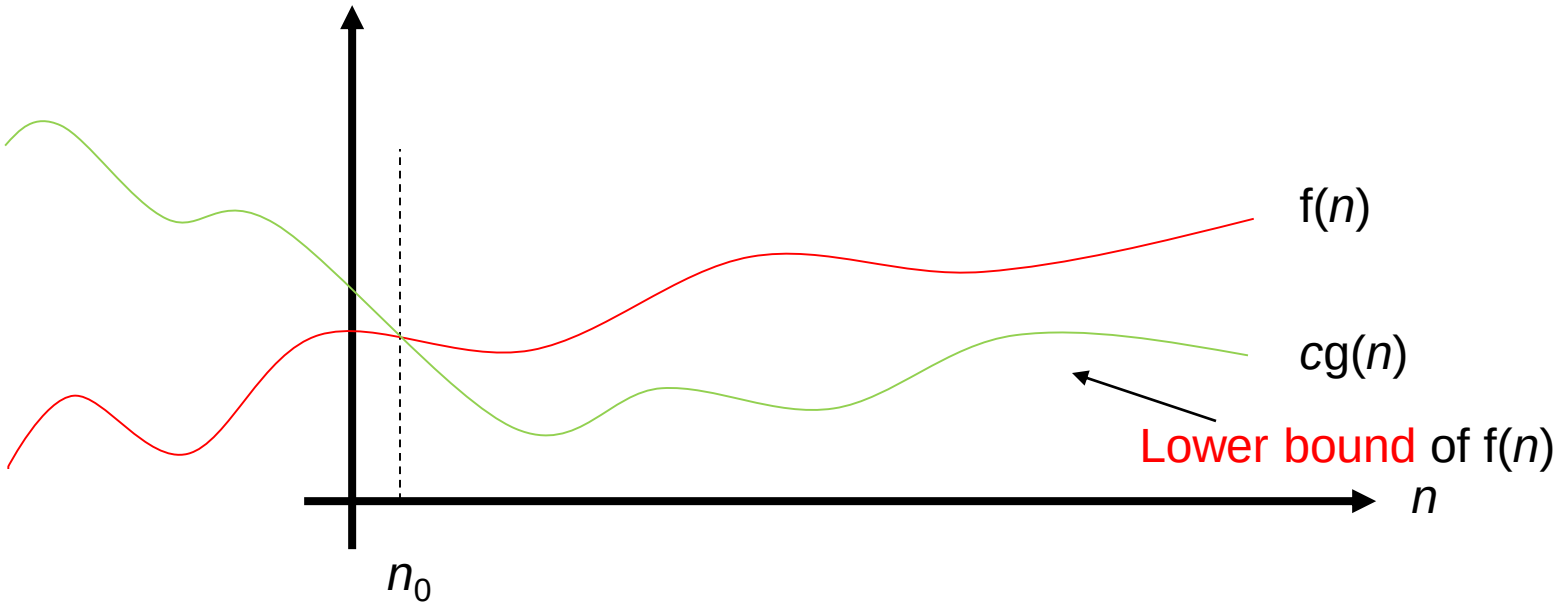
# Asymptotic Notation O

- Big-O notation defines an upper bound of an algorithm's running time.
- We say that a function f(n) is of the order of g(n), iff there exists constant $c > 0$ and $n_0$ such that $f(n) \leq cg(n)$ for all $n \geq n_0$
- In other words, f(n) is at most a constant times of g(n) for sufficiently large of values of n
- Using Big-O notation: f(*n*) = O(g(*n*))



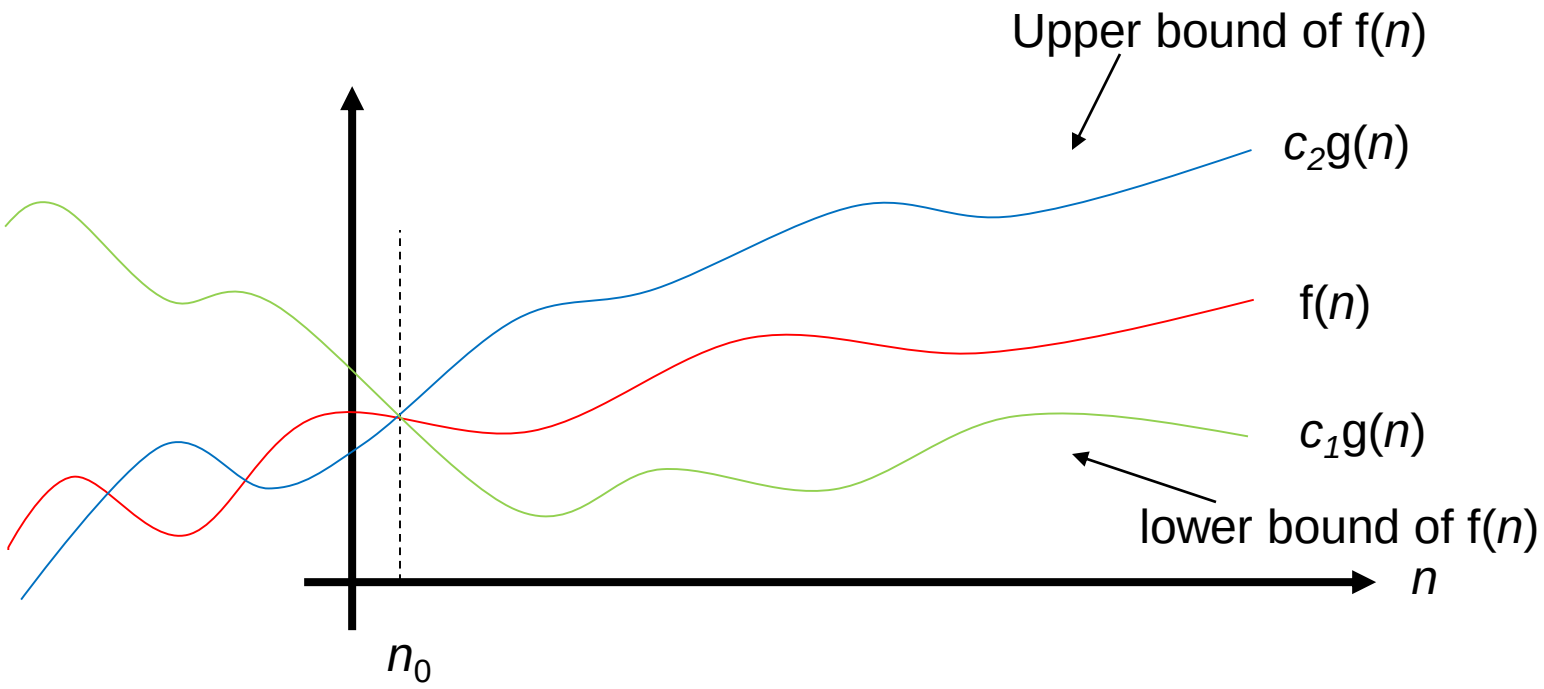
# Asymptotic Notation Ω

- Big-Omega notation defines a lower bound of an algorithm's running time.
- f($n$) = Ω(g($n$)) iff there exists constant $c > 0$ and $n_0$ such that f($n$) ≥ $c$g($n$) for all $n \geq n_0$



# Asymptotic Notation Θ

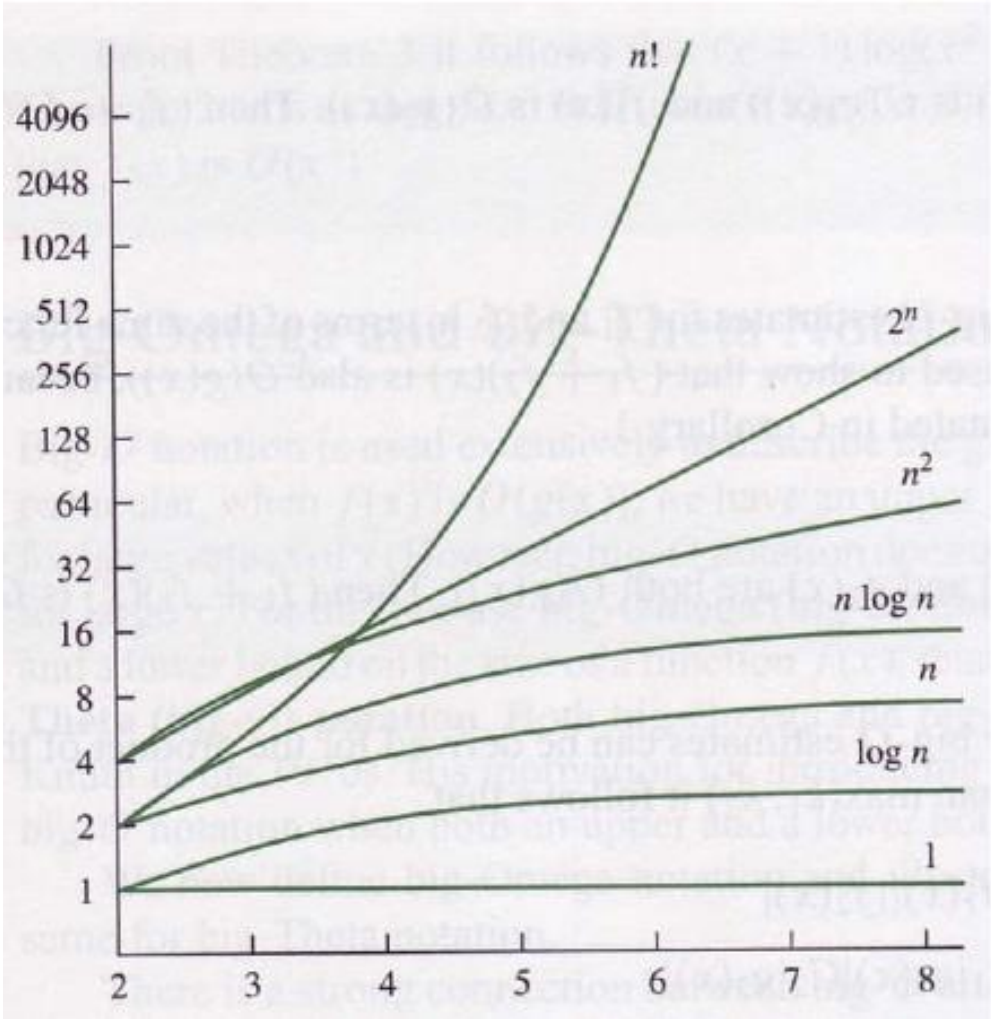
- Big-Theta notation defines an exact bound of an algorithm's running time.
- f($n$) = Θ(g($n$)) iff there exists constant $c_1 > 0$, $c_2 > 0$ and $n_0$ such that $c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all $n$, $n \geq n_0$



# Important Complexity Classes

- O(1): Constant time
- O($\log_2 n$): Logarithmic time
- O($n$): Linear time
- O($n \log_2 n$): Log-linear time
- O($n^2$): Quadratic time
- O($n^3$): Cubic time
- O($n^k$): Polynomial time
- O($2^n$): Exponential time

# Important Complexity Classes




Factorial time

Exponential time

Quadratic time

Log-linear time

Linear time

Logarithmic time

Constant time

Increasing complexity

Becomes not feasible when *n* grows larger

# Practical Problem Sizes

- Practical problem sizes that can be handled by algorithms of different complexity classes.

| Complexity class | | Maximum n | Example applications |
|---|---|---|---|
| constant time | O(1) | Unlimited | random number generation, hashing |
| logarithmic | O(log n) | Effectively unlimited | binary search |
| linear | O(n) | $n < 10^{10}$ | sum of a list, sequential search, vector product |
| log-linear | O(n log n) | $n < 10^{8}$ | fast sorting algorithms |
| quadratic | $O(n^2)$ | $n < 10^{5}$ | 2D matrix addition, insertion sort |
| cubic | $O(n^3)$ | $n < 10^{3}$ | 2D matrix multiplication |
| exponential | $O(a^n)$ for a > 1 | $n < 36$ for $2^n$ | traveling salesman, placement and routing in VLSI, many other optimization problems |
| Factorial | O(n!) | $n < 15$ | enumerate the permutations of n objects |

# Big-O of This Code?

```
int i, n, sum = 0, product = 1;
cin >> n;
for(i = 0; i < n; i++)
    sum += i;
for(i = 1; i < n; i++)
    product *= i;
printf("%d %d", sum, product);
```

1 time

1 time

*n* times

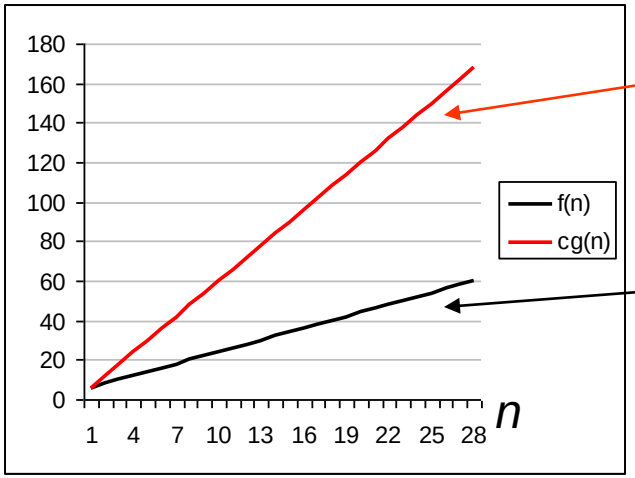
(*n* – 1) times

1 time

Total execution
= 1 + 1 + *n* + (*n*-1) + 1
= 2*n* + 2

# Big-O Notation

- If total execution is $2n + 2$
- $f(n) = 2n + 2$
- $f(n) = 2n + 2 \leq 2n + 2n$ (for $n \geq 1$)
- $f(n) = 2n + 2 \leq 4n$ (for $n \geq 1$)
- $f(n) = 2n + 2 \leq cg(n)$ ($c = 4$, $g(n) = n$, $n_0 = 1$)
- $f(n) = O(g(n))$
- So, time complexity of the above code = $O(n)$




*cg*(*n*): upper bound of f(*n*)

*f*(*n*): always smaller than *cg*(*n*) for $n \geq n_0$

# Why Using Big-O

- We concern about the bounds on the performance of algorithms, rather than giving exact speeds.
- The actual running time depends upon our processor speed, the condition of our processor cache, etc. It's all very complicated in the concrete details, and moreover not relevant to the essence of the algorithm.
- Big-O provides a simple bounded approximation of the complexity order.
  - **Prove** the running time is always less then some **upper bound** *for sufficiently large input size*
  - **Derive** the **average** running time for a *random* input
  - Count the no. of iterations (loops) and function calls
  - Don’t Count variable declaration and other constant time items

# Big O Notation Examples

- If total execution is $2n^2 + 6n + 4$
- = $O(n^2)$

- If total execution is $7n^3 + 2n^2 + 6n + 4$
- = $O(n^3)$

- If total execution is $(n + 1)/2$
- = $O(n)$

- If total execution is $2n^2 + \log n$
- = $O(n^2)$
- The fastest growing one dominates the order of $f(n)$

# Example: Matrix Addition

- The problem size is given by N, the dimension of the matrix.

```
#define N 10
typedef int Matrix[N][N];

1       void MatrixAdd(Matrix A, Matrix B, Matrix C) {
2          for (int i = 0; i < N; i++)
3             for (int j = 0; j < N; j++)
4                C[i][j] = A[i][j] + B[i][j];
5          return;
6       }
```

| line | step count | frequency |
|---|---|---|
| 1 | 0 | 0 |
| 2 | 1 | N |
| 3 | 1 | $N^2$ |
| 4 | 1 | $N^2$ |
| 5 | 1 | 1 |
| 6 | 0 | 0 |

**Total = $2N^2 + N + 1$**
**= $O(N^2)$**

# Exercises

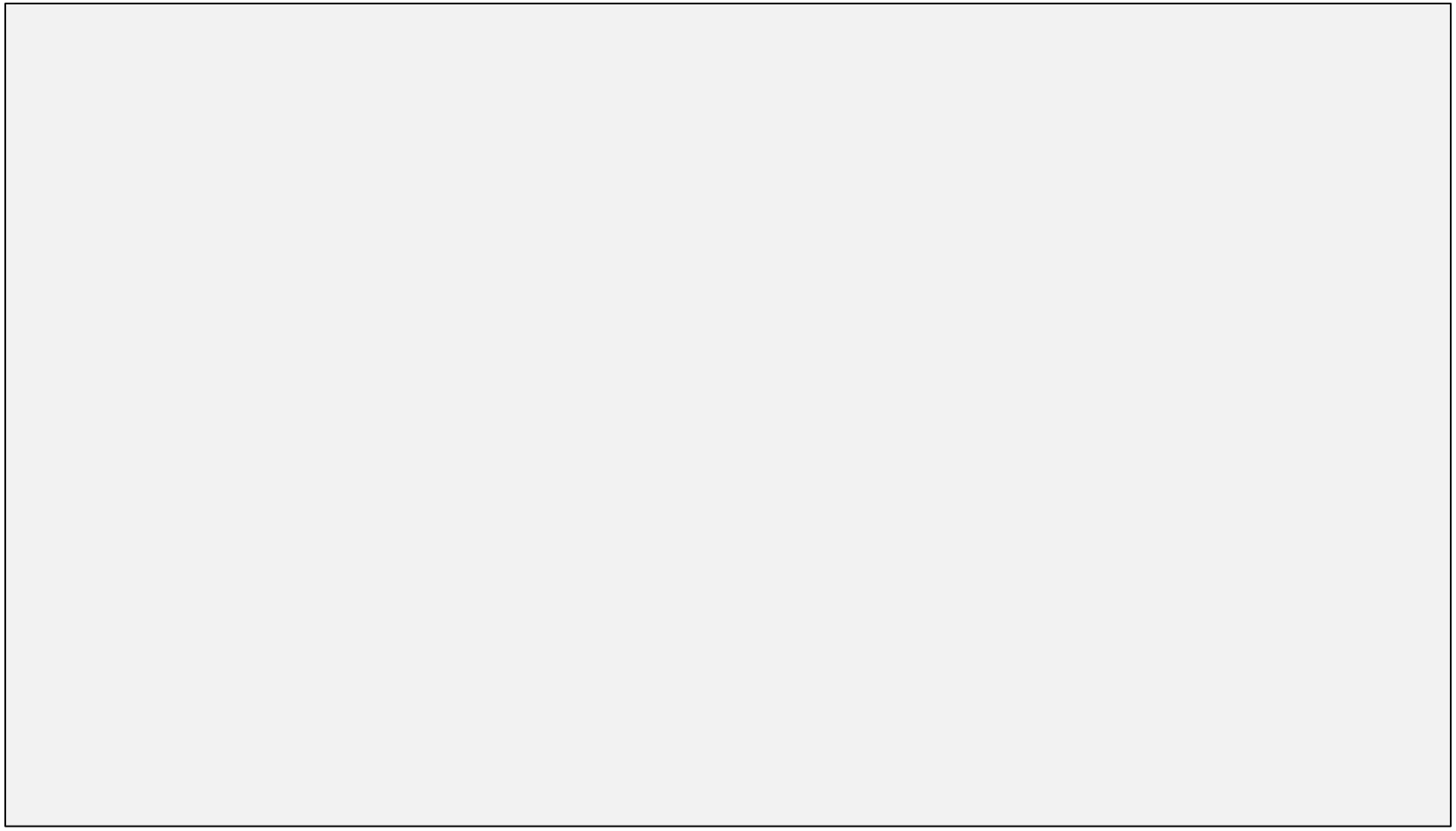
```
   //Code A
   int findMax(int a[], int n) {
#1    int i, max = a[0];
#2    for(i = 1; i < n; i++)
#3        if (a[i] > max)
#4            max = a[i];
#5    return max;
   }
```

```
   //Code C
#1 for(i = 0; i < n; i++)
#2   for(j = 0; j < i; j++)
#3      sum++;
```

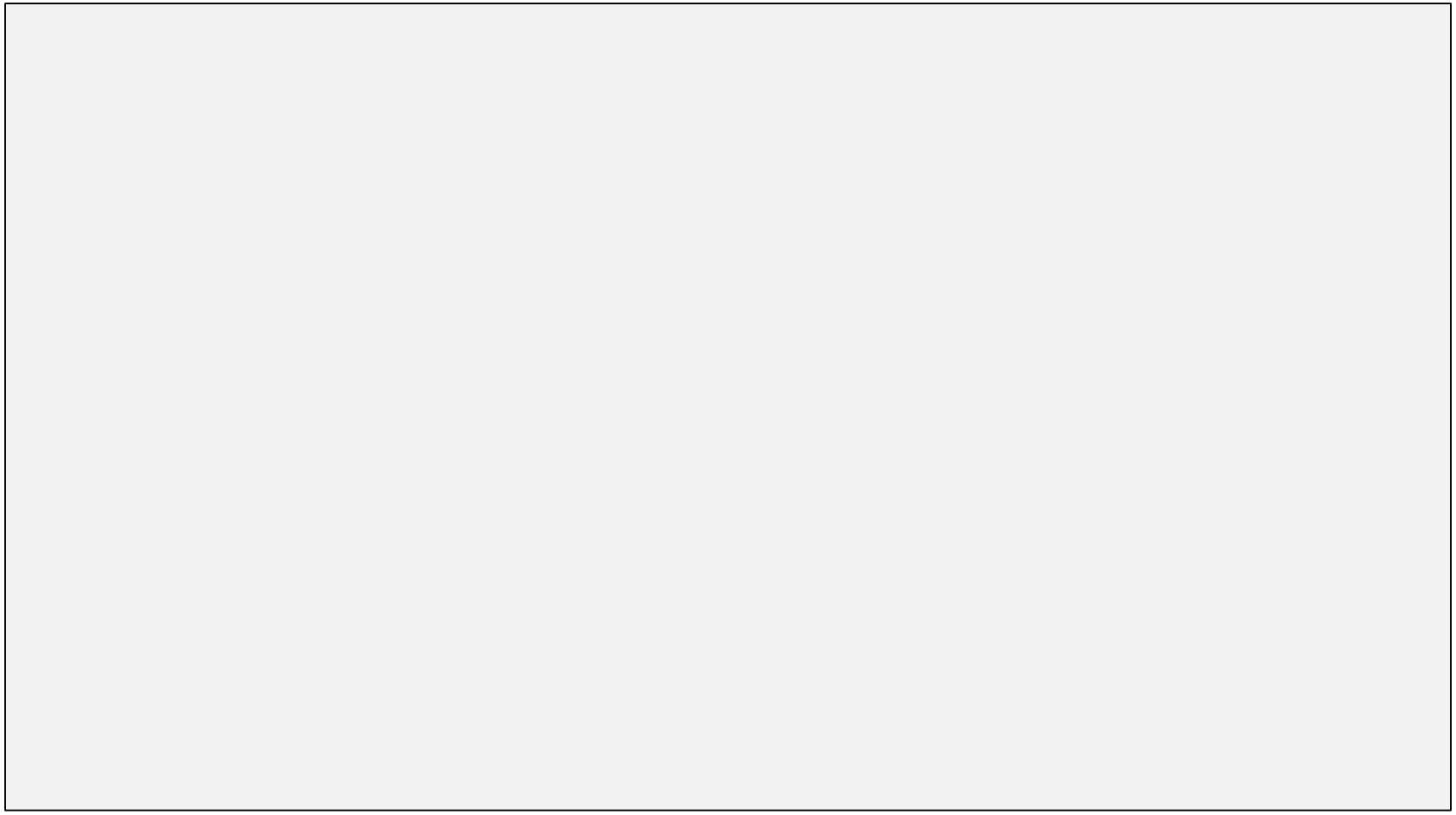
```
   //Code B
#1 for(i = 0; i < n; i++)
#2    for(j = 0; j < n * n; j++)
#3        sum++;
```

What are the Big-O of these 3 pieces of code?

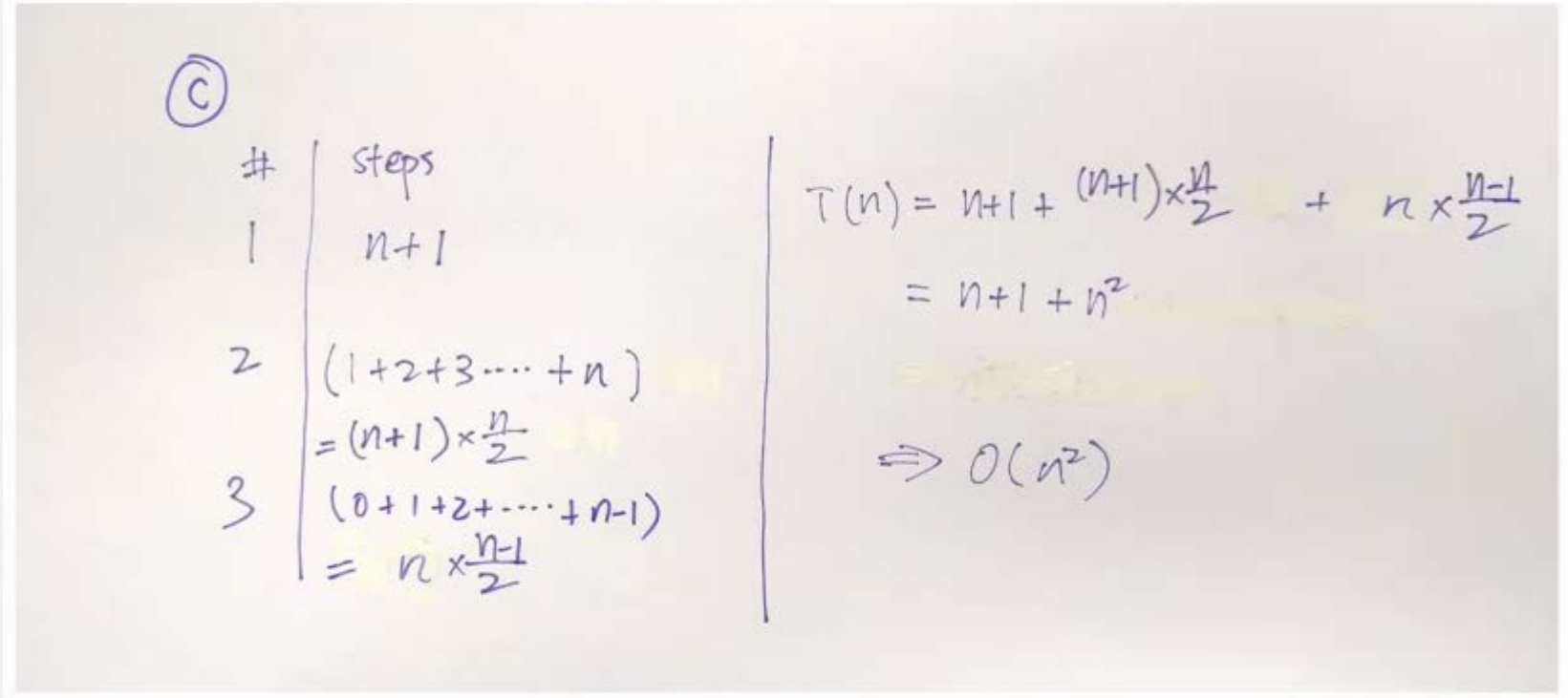
# Ex A



# Ex B



# Ex C



# Other Considerations

- Small no. of input
  - The growth rate of the running time may be less important than the constant factor when the input size is small
- Difficult to understand
  - An efficient but complicated algorithm may not be desirable because it is difficult to be maintained
  - e.g. recursive vs. non-recursive algorithm
- Space requirement
  - If an efficient algorithm uses too much space and is implemented with the slow secondary storage, it may negate the efficiency

# Algorithm Design Strategy

# Brute Force

- The most straightforward algorithm design technique covered on the course is brute force.

1. Initially the entire input is unprocessed
2. The algorithm processes a small piece of the input on each round
   - the amount of processed data gets larger and the
   - amount of unprocessed data gets smaller
3. Finally there is no unprocessed data and the algorithm halts
4. These types of algorithms are easy to implement and work efficiently on small inputs.

- Example:
  - Finding maximum/minimum in an array
  - Cracking a password by trying all combinations

# Divide and Conquer

- Now another technique called divide and conquer is introduced. It is often more efficient than the brute force approach.

1. The problem is divided into several subproblems that are like the original but smaller in size.
2. Small subproblems are solved straightforwardly
3. Larger subproblems are further divided into smaller units
4. Finally the solutions of the subproblems are combined to get the solution to the original problem

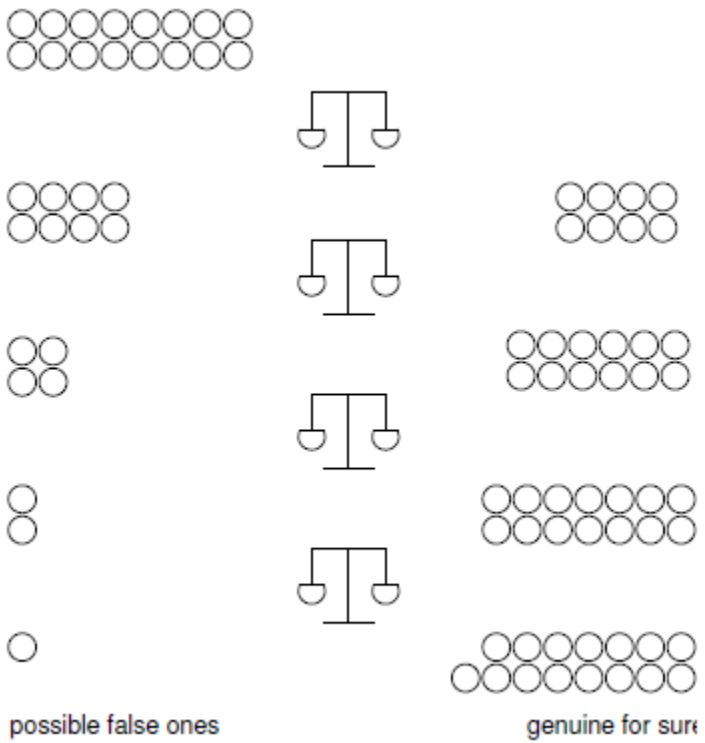
- Let's get back to the claim made earlier about the complexity notation not being fixed to programs and take an everyday, concrete example

# Finding The False Goldcoin

- The problem is well-known from logic problems.
- We have n gold coins, one of which is false. The false coin looks the same as the real ones but is lighter than the others. We have a scale we can use and our task is to find the false coin.
- We can solve the problem with Brute Force by choosing a random coin and by comparing it to the other coins one at a time.
- At least **1** and at most **n-1** weightings are needed. The best-case efficiency is O(1) and the worst and average case efficiencies are O(n).
- Alternatively we can always take two coins at random and weigh them. At most **n/2** weightings are needed and the efficiency of the solution is still the same.

# Finding The False Goldcoin

- The same problem can be solved more efficiently with divide and conquer:
- Divide the coins into the two pans on the scales. The coins on the heavier side are all authentic, so they don't need to be investigated further.
- Continue the search similarly with the lighter half, i.e. the half that contains the false coin, until there is only one coin in the pan, the coin that we know is false.
- The solution is recursive and the base case is the situation where there is only one possible coin that can be false.



# Finding The False Goldcoin

- The amount of coins on each weighting is 2 to the power of the amount of weightings still required: on the highest level there are $2^{weightings}$ coins, so based on the definition of the logarithm:

$$2^{weighings} = n \quad => \quad \log_2 n = weightings$$

- Only $\log_2 n$ weightings is needed, which is significantly fewer than $n/2$ when the amount of coins is large.
- The complexity of the solution is O(log n) both in the best and the worst-case.