CSCI 2110 Data Structures and Algorithms

# Module 2: Introduction to Algorithm Complexity

# Learning Objectives/Topics

- *What is algorithm time complexity? Why should we care?*
- *How do you express algorithm time complexity in terms of basic operations?*
- *Define the standard measure of algorithm complexity – the order of complexity or big O.*
- *Study practical examples of typical big O's and know simple rules for deriving big O.*
- *Know other types of complexity, namely, Big-Omega, Big-Theta and Little-O and their relation to Big-O.*
- *Distinguish between average case, worst-case and best-case running time complexities.*

# What is algorithm time complexity?

- *Algorithm time complexity is just a measure of how fast your algorithm/program runs.*

- *Thus it is a measure of the efficiency of the algorithm.*

- *This efficiency can be expressed by the speed or the running time of the algorithm (that is, of the program implementing the algorithm).*

# *Why should we care?*

- *Understanding algorithm time complexity can make a world of difference in software design.*
- *We can compare algorithms and choose the right algorithm for the right task.*
- *Some simple examples:*
  - *Bubble sort algorithm on a million records → 1 billion steps.*
  - *Quick sort algorithm on a million records → 20 million steps!*
  - *Linear search algorithm on a million records → 1 million steps.*
  - *Binary search algorithm on a million records → 20 steps!*

# Time Complexity vs. Space Complexity

- *In the previous examples, we are concerned about the runtime efficiency or the time complexity of the algorithm.*
- *Another factor that can also determine the efficiency of the algorithm is the space complexity.*
- *Space complexity is a measure of the memory required by the algorithm.*
- *Principles behind concepts of understanding time and space complexity are similar.*
- *We will focus on time complexity.*

# We need to measure the run time – why not use the "wall clock" approach?

- *Suppose that we run a competition in this class to test who has built the fastest spell checker algorithm.*
- *Riley says: "My spell checker took 2.5 minutes to complete its task".*
- *Sebastian announces: "My spell checker completed in 1.5 minutes".*
- *Mohamed shouts: "My spell checker took 50 seconds".*
- *Kulween pipes in: "My spell checker took 40 seconds!"*
- *Yunzhi quietly texts: "My spell checker just took 10 seconds."*
- *This is called the "wall clock" approach to measuring time complexity → looking at the absolute time the program took to run.*

# The “wall clock” approach is not reliable …

- *This is not a reliable measure because …*
- *…many factors cloud the actual efficiency of the algorithm, for example,*
  - *CPU Speed and OS*
  - *System environment (how many other processes were running simultaneously?)*
  - *Programming language and platform*
  - ***Differences in the document size (the size of the input)***

# What is a better approach?

- *We need to compare algorithms independent of the peripheral issues such as CPU speed, etc.*
- *Hence a better approach is to count the number of basic operations in the algorithm for a specific input size.*
- *The running time will be proportional to this count.*
- *What are the basic operations of an algorithm?*
  - *Additions/subtractions*
  - *Multiplications/divisions*
  - *Comparisons*
  - *Assignment (copy) operations, etc.*

# We will sneak in an approximation….

- *We will say that every basic operation such as*
  - *Addition/ Subtraction*
  - *Multiplication/Division/Modulus*
  - *Comparison*
  - *Assignment (copy)*
  - *etc.*

  *takes the same amount of time.*
- *We'll see later that in the long run, this approximation is valid.*
- *In some cases, we may not even count all the basic operations, but just some dominant operations.*
- *Again, in the long run this approximation will be valid.*

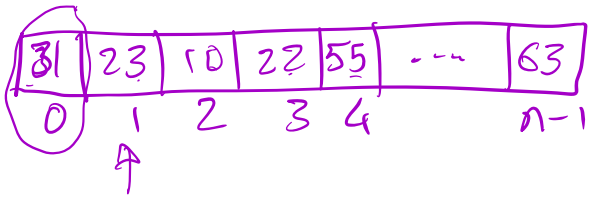
# What about the input data size?

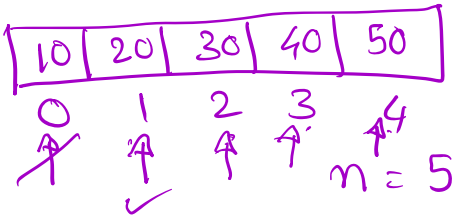
- *This is perhaps the most important parameter in comparing algorithms.*
- *If we say that an algorithm A runs faster than algorithm B, we need to make sure that they are running the same input data size.*
- *The input data could be, for example, the size of the array to be processed, number of characters in a file, number of records in a database, etc.*
- *If we increase the input size, will algorithm A still be faster than B?*
- *Therefore, we need to express the number of basic operations in terms of the input data size.*

Let's work through some examples to determine the running times in terms of basic operations.

**Example 1: Algorithm to find the largest integer in an array of n integers.**

```
public static int findLargest(int[] arr)
{
        int largest = arr[0];
        int index = 1;
        while (index < arr.length)
        {
                if (arr[index] > largest)
                        largest = arr[index];
                index++;
        }
        return largest;
}
```




Let t be the time for one basic operation.

Total run time of the algorithm (that is, the above code)

$= t + t + nt + (n-1)t + (n-1)t + (n-1)t + t$

(1) (2) (3) (4) (5) worst case (6) (7)

$= 3t + nt + 3(n-1)t$

$= 3t + nt + 3nt - 3t = 4nt$

**Run time of the algorithm = $4nt$**

or

**Run time is proportional to $4n$**

or simply, we say **Run time is $4n$**

Check it out $1+2+3+4 = 10$

$4(4+1)/2 = 10$

$4x^4 = 4*x*x*x*x$

**Example 2: Algorithm to evaluate a polynomial**

Write a program to evaluate

$P(x) = 4x^4 + 2x^3 - 5x^2 + 10x - 5$

given some value of $x$, say $x = 2$.

Number of multiplications $= 4 + 3 + 2 + 1 + 0$

Number of adds / subs $= 4$

Let's generalize this ( $n$ = degree of the polynomial)

$P(x) = ax^n + bx^{n-1} + cx^{n-2} + \ldots + mx + q$

Number of multiplications $= n + (n-1) + (n-2) + \ldots + 1 + 0$

$= \frac{n(n+1)}{2}$ (from Sum of Series formula)

Number of adds/subs $= n$

Total number of operations $= \frac{n(n+1)}{2} + n = \frac{n^2}{2} + \frac{n}{2} + n$

$= 0.5n^2 + 0.5n + n = 0.5n^2 + 1.5n$

Run time is $0.5n^2 + 1.5n$

**Example 3: Algorithm to find the weighted average of n items**

Determine the average response time for visiting n websites, given the response time for each website and the number of times that website is visited.

| Website | # times | Response Time |
|---|---|---|
| 1 | $m_1$ | $f_1$ |
| 2 | $m_2$ | $f_2$ |
| 3 | $m_3$ | $f_3$ |
| ⋮ | ⋮ | ⋮ |
| $n$ | $m_n$ | $f_n$ |

Average Response Time $= \frac{(m_1*f_1 + m_2*f_2 + \ldots + m_n*f_n)}{(m_1 + m_2 + m_3 + \ldots + m_n)}$

Run time of the program = $n$ multiplications + $(n-1)$ adds + $(n-1)$ adds + 1 division

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

$= 3n - 1$

Run time is $3n-1$

## *Ok, let's compare algorithms….*

- *Suppose that we have three algorithms with the following run times (or run times proportional to):*
  - *Algorithm A: 5000 n + 1000*
  - *Algorithm B: 200 $n^2$ + 500 n*
  - *Algorithm C: 1.1$^n$*
- *Which algorithm is the best?*
- *From the calculations for n=10, n=100, n=1000, and n=10,000 we conclude that Algorithm A performs the best in the "long run" or for "large enough values of n".*
- *This is refered to as the asymptotic complexity*
- *WE COMPARE ALGORITHMS FOR LARGE VALUES OF THE INPUT SIZE OR IN OTHER WORDS, BASED ON THEIR ASYMPTOTIC COMPLEXITIES*

Run times for different values of n
(Smaller the number, better the algorithm)

| | n = 10 | n = 100 | n = 1000 | n = 10,000 |
|---|---|---|---|---|
| A | 51000 | 501,000 | 5,001,000 | 50,001,000 |
| B | 25000 | 2,050,000 | 25,000,000 | 20,005,000,000 |
| C | 3 | 13780.19 | $2.5 \times 10^{41}$ | A VERY, VERY, VERY, VERY, VERY LARGE NUMBER !!! |

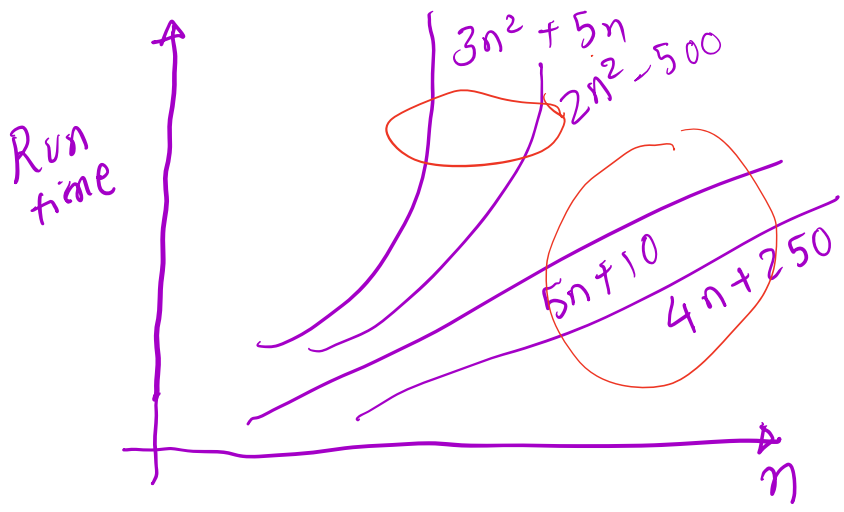
## *Now comes the Big O notation….*

- *Compare the following algorithms:*

- *Algorithm A: 5 n + 10*
- *Algorithm B: 4 n + 250*
- *Algorithm C: $3n^2 + 5n$*
- *Algorithm D: $2 n^2 - 500$*

- *We say that Algorithms A and B are in the "same league" for large values of n.*
- *Similarly Algorithms C and D are in the "same league" for large values of n.*
- *The Big O notation classifies algorithms into the "same league".*
- *In other words, we say that the complexity of algorithms A and B is O(n) and the complexity of algorithms C and D is $O(n^2)$.*

→ same breed of race horses



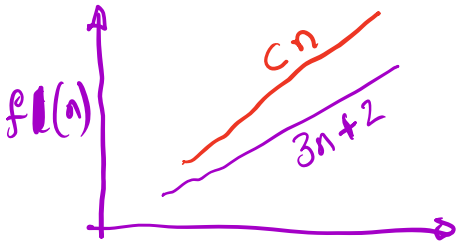

$O(n^2)$

$O(n)$

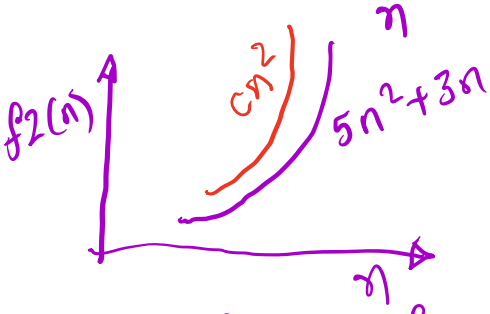
## NOW A LITTLE MATH....

## FORMAL DEFINITION OF BIG O OR THE O( ) NOTATION

Why is a function $f1(n) = 3n+2$ said to be $O(n)$?
Why is a function $f2(n) = 5n^2+3n$ said to be $O(n^2)$?
The basis is in the growth of these functions for large values of $n$.

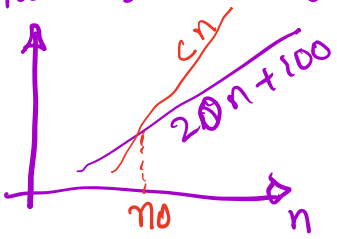



$3n+2$ is always bounded by $cn$, where $c$ is some constant. (that is, we can always find such a constant $c$)




$5n^2+3n$ is always bounded by $cn^2$, where $c$ is some constant.

What if the function looks as follows:




Even in this case, $20n+100$ is bounded by $cn$ for $n >= n_0$

Let's call $3n+2$ as $f(n)$ and $cn$ as $O(g(n))$

A function $f(n)$ is said to be of the order $g(n)$, written as $O(g(n))$, if there exist positive constants $c$ and $n_0$ such that $f(n) <= c\,g(n)$ for $n >= n_0$

## *Deriving the Big O is easy!*

- *Replace all additive constants in the run time with the constant 1.*
- *Retain only the highest order term in this modified run time.*
- *If the highest order term is 1, then the order is O(1).*
- *If the highest order term is not 1, remove the constant (if any) that multiplies the term.*
- *You are left with the order.*

Examples

Derive the order of complexities (Big O) for the following functions (that is, run times expressed in terms of the input size n)

1. $f(n) = 3n + 25 \rightarrow 3n + 1 \rightarrow 3n \rightarrow n$ $O(n)$

2. $25n^3 + 2n + 5 \rightarrow 25n^3 + 2n + 1 \rightarrow 25n^3 \rightarrow n^3$ $O(n^3)$

3. $\frac{n^2}{2} + 2000n + 1 \rightarrow \frac{n^2}{2} \rightarrow n^2$ $O(n^2)$

4. $1 + 2 + 3 + \cdots + n = \frac{n(n+1)}{2} = \frac{n^2}{2} + \frac{n}{2}$ $O(n^2)$

from Sum of series formula

must express the function as a polynomial.

5. $(3n+1)(n+2) = 3n^2 + 6n + n + 2 \rightarrow n^2$ $O(n^2)$

## EXAMPLES FOR DERIVING THE BIG O (cont'd.)

6. $\frac{10n+5n}{3n} + 4 = \frac{10n}{3n} + \frac{5n}{3n} + 4 = \frac{10}{3} + \frac{5}{3} + 4 \rightarrow$ $O(1)$

7. $3\log_2 n + 50n + 30 \rightarrow 50n \rightarrow n$ $O(n)$

8. $250n + 10n\log_2 n + 25$
$\rightarrow 250n + 10n\log_2 n + 1$ $O(n\log_2 n)$

9. $n\log n^3 + \log n^4 + n^2 = 3n\log n + 4\log n + n^2$ $O(n^2)$

$n\log n^3 = 3n\log n$

10. $12 * 2^n + 500 + n^{10} \Rightarrow 12 \times 2^n$ $O(2^n)$

11. $\frac{200}{n} + \frac{50}{\sqrt{n}} + 500 \Rightarrow$ $O(1)$

12. $\frac{200}{n} + \frac{50}{n^2} \rightarrow$ $O\left(\frac{1}{n}\right)$

---

ORDER TABLE

$$\frac{1}{n^2} < \frac{1}{n} < \frac{1}{\sqrt{n}} < 1 < \log_2 n < \sqrt{n} < n < n\log_2 n < n\sqrt{n} < n^2 < n^3 < 2^n$$

$2^1 = 2$ $2^4 = 16$ $2^3 = 8$

$2^5 = 32$ $2^6 = 64$ $\log_2 8 = 3$

**PRACTICAL EXAMPLES**

**O(1): Constant Time Complexity**

(Means that the algorithm requires the same fixed number of steps irrespective of the size of the task)

Accessing the $i^{th}$ element in an array of size n

num = a[i];


**O(n): LInear Time Complexity**

Find the largest integer in an array of size n.

Runtime = 4n O(n)

**O($n^2$): Quadratic Time Complexity**

Polynomial Evaluation (Degree of polynomial = n)

Runtime = $0.5n^2 + 1.5n \longrightarrow O(n^2)$

**O(log n): Logarithmic Time Complexity**

Binary Search of a sorted array of size n = $O(\log_2 n)$. E.g. Binary Search of 1024 items takes 10 steps ($2^{10} = 1024$)

**O(n log n): En Log En Time Complexity**

Quick Sort

Merge Sort

**O($n^3$): Cubic Time Complexity**

Matrix multiplication

Let n be the size of the matrices to be multiplied.

$$\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \times \begin{bmatrix} j & k & l \\ m & n & o \\ p & q & r \end{bmatrix} = \begin{bmatrix} x & x & x \\ x & x & x \\ x & x & x \end{bmatrix}$$

$x = aj + bm + cp$

# operations to get one number in the output = 3 mults & 2 adds

∴ Total number of operations = $3 \times 3 \times (3+2)$

In general, total number of operations = $n \times n \times (n + (n-1))$

$\Rightarrow O(n^3)$

**O($k^n$) Exponential Time Complexity**

"Brute Force" or exhaustive search through all possible combinations.

Example: Breaking a secret key stored as a n-bit binary number.

Encryption: Plaintext $\xrightarrow[\text{Secret key}]{}$ Ciphertext

Decryption: Ciphertext $\xrightarrow[\text{Secret key}]{}$ Plaintext

Caeser Cipher: HEY (Plaintext) $\xrightarrow[\text{Secret key } K=2]{}$ JGA

Many strong encryption algorithms exist. Examples: AES, 3DES, etc.

Problem: Knowing the ciphertext & the plaintext, can a hacker get the key?

Solution: Brute Force Search → check all possible combinations of an n-bit binary number (secret key).

$n=3$: 000, 001, 010, 011, 100, 101, 110, 111 (8)

$n=4$: 0000, 0001, ..... , 1111 (16) ← $2^4$

$n=5$: 00000, ..... 11111 (32) ← $2^5$ ← $2^3$

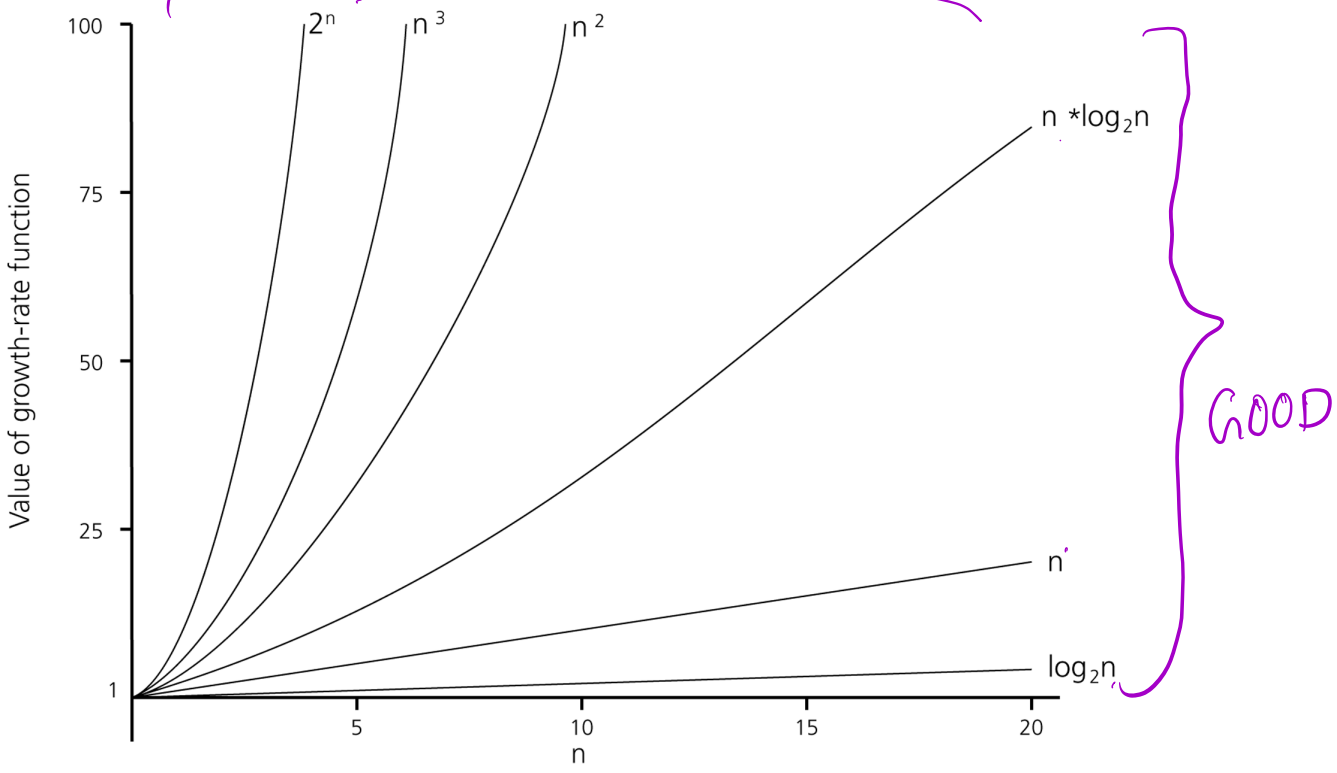
For an n-bit key, the hacker must check $2^n$ combinations.

This is an algorithm of complexity $O(2^n)$.

## COMPARISON OF GROWTH RATES

UGLY BAD GOOD

(b)




(a)

| Function | n = 10 | 100 | 1,000 | 10,000 | 100,000 | 1,000,000 |
|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| $\log_2 n$ | 3 | 6 | 9 | 13 | 16 | 19 |
| n | 10 | $10^2$ | $10^3$ | $10^4$ | $10^5$ | $10^6$ |
| $n * \log_2 n$ | 30 | 664 | 9,965 | $10^5$ | $10^6$ | $10^7$ |
| $n^2$ | $10^2$ | $10^4$ | $10^6$ | $10^8$ | $10^{10}$ | $10^{12}$ |
| $n^3$ | $10^3$ | $10^6$ | $10^9$ | $10^{12}$ | $10^{15}$ | $10^{18}$ |
| $2^n$ | $10^3$ | $10^{30}$ | $10^{301}$ | $10^{3,010}$ | $10^{30,103}$ | $10^{301,030}$ |

Example of an algorithm with $O(n!)$ complexity

Traveling Salesperson Problem: Given $n$ cities and inter-city distances, find the shortest path that starts at a city, visits each city exactly once, & returns to the starting city.

ESTIMATION PROBLEMS WITH BIG O

1. An algorithm takes 1 ms to run when the input size is 1000. Approximately, how long will the algorithm take to process an input size of 5000 if it has the following orders of complexities:

   a. Linear
   b. Quadratic
   c. nlogn

Given the data size, estimate the time

a) Linear $O(n)$

| # steps | time |
|---|---|
| 1000 | 1 ms |
| 5000 | x ? |

$x = \frac{1 \times 5000}{1000} = 5$

Answer: 5 ms

b) Quadratic $O(n^2)$

| #steps | time |
|---|---|
| $1000 \times 1000$ | 1 ms |
| $5000 \times 5000$ | x ? |

$x = \frac{1 \times 5000 \times 5000}{1000 \times 1000} = 25$

Answer = 25 ms

c) n log n

| #steps | Time |
|---|---|
| 1000 log 1000 | 1 ms |
| 5000 log 5000 | x ? |

$x = \frac{1 \times 5000 \log 5000}{1000 \log 1000} = 6.16$

Answer = 6.16 ms

2. Algorithm A takes 10 ms to solve a problem of size 1000. Algorithm B takes 100 ms to solve a problem of size 10,000. Algorithm A's complexity is quadratic while Algorithm B's complexity is cubic. How large a problem can each solve in 1 second?

1 sec = 1000 ms

Given the time, estimate the data size

Alg. A

$O(n^2)$

| Time | #steps |
|---|---|
| 10 ms | $1000 \times 1000$ |
| 1000 ms | $x * x$ ? |

$x^2 = \frac{1000 \times 1000 \times 1000}{10}$ (100)

$x = 10{,}000$

Answer: 10,000

Alg. B

$O(n^3)$

| Time | #steps |
|---|---|
| 100 | $10000 \times 10000 \times 10000$ |
| 1000 | $x * x * x$ ? |

$x^3 = \frac{10000 \times 10000 \times 10000 \times 1000}{100}$ (10)

$\therefore x = \sqrt[3]{10000 \times 10000 \times 10000 \times 100}$

$= 10000 \times \sqrt[3]{10} = 21544$

Answer: 21544

3. Software packages A and B spend exactly $T_A=c_An^2$ and $T_B = c_Bn^3+500$ milliseconds to process n data items, respectively, where $c_A$ and $c_B$ are some constants. During a test, A takes 1000 milliseconds and B takes 600 milliseconds to process n=100 data items. Which package is better for processing 10 data items? Which package is better for processing 1000 data items? (Show steps).

First find $C_A$

$$T_A = C_A n^2$$

$$1000 = C_A * 100 * 100 \quad \therefore C_A = \frac{1000}{100 \times 100} = 0.1$$

Next find $C_B$

$$T_B = C_B n^3 + 500$$

$$600 = C_B \times 100 \times 100 \times 100 + 500$$

$$100 = C_B * 100 * 100 \times 100 \quad \therefore C_B = \frac{100}{100 \times 100 \times 100} = 0.0001$$

---

$n = 10$

A is better

Alg. A

$$T_A = C_A n^2 = 0.1 * 10 \times 10 = 10 \text{ ms}$$

Alg. B

$$T_B = C_B n^3 + 500 = 0.0001 * 10^3 + 500 = 500.1 \text{ ms}$$

---

$n = 1000$

A is better

$$T_A = C_A n^2 = 0.1 * 1000^2 = 100{,}000 \text{ ms}$$

$$T_B = C_B n^3 + 500 = 0.0001 * 1000^3 + 500 = 100{,}500 \text{ ms}$$

These are mainly for your CSCI 3110

## OTHER ORDERS OR COMPLEXITY

Although we will use the Big O primarily as a measure of algorithm complexity in this course, there are three other types of complexity related to Big O.

We redefine Big-O to place it in context.

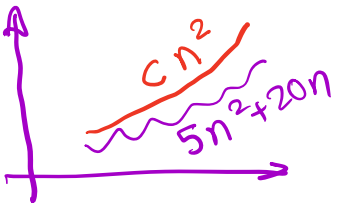
**Big O: A growth function T(N) is O(F(N)) if there are positive constants c and $N_0$ such that T(N) <= cF(N) when N>=$N_0$ .**

**Big Omega: A growth function T(N) is Ω(F(N)) if there are positive constants c and $N_0$ such that T(N) >= cF(N) when N>=$N_0$ .**

**Big Theta: A growth function T(N) is Θ(F(N)) if and only if T(N) is O(F(N) and T(N) is Ω(F(N)).**

**Little-O: A growth function T(N) is o(F(N)) if and only if T(N) is O(F(N) and T(N) is is not Θ(F(N)).**

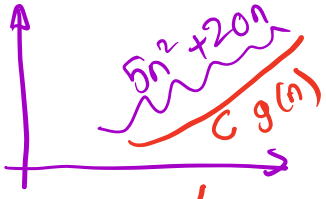
Big O




We say that $5n^2+20n$ is $O(n^2)$ because $5n^2+20n$ can never be worse than $cn^2$

Big O : Worst Case Complexity

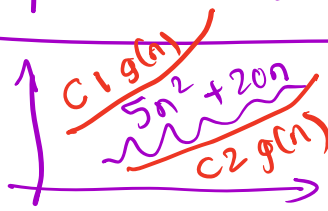
Ω




$5n^2+20n$ can never be better than $c\,g(n)$

Ω is best case complexity

Θ




Average case complexity

Θ

Little o


Big O → Little o

## ORDER ARITHMETIC

Let runtime (complexity) of an algorithm A be $f1(n) = O(g1(n))$

& ------ B be $f2(n) = O(g2(n))$

What is the order of complexity of algorithm whose run time is

a) $f1(n) + f2(n)$ ?

$MAX(O(g1(n)), O(g2(n)))$

b) $K * f1(n)$ where k is a constant ?

$O(g1(n))$

c) $f1(n) * f2(n)$ ?

$O(g1(n) * g2(n))$

---

| Alg. A | Alg B |
|---|---|
| $f1(n)$ | $f2(n)$ |
| $O(n)$ | $O(n \log n)$ |

Then algorithm with run time $f1(n) + f2(n)$ has complexity $O(n \log n)$

Another algorithm with run time $= f1(n) * f2(n)$ has complexity $O(n^2 \log n)$

## SOME SIMPLE RULES OF THUMB

Find the order of complexity of each of the following code segments

① 
```
for (int i = 1; i <= n ; i++)
{
    // some set of p statements
    // where p is a constant
}
```
# iterations = n

# operations = p * n

∴ O(n)

② 
```
for (int i = 1; i <= n; i++)
{ for (int j = 1; j <= n; j++)
  {
      // p statements
  }
}
```
# iterations = $n^2$

∴ $O(n^2)$

③ Code ① followed by Code ② # iterations = $n + n^2 \rightarrow O(n^2)$

④ 
```
for (int i=1; i<=100; i++)
{ for (int j=1; j<=n; j++)
  {
      // set of p statements
  }
}
```
# iterations = 100 × n

→ O(n)

⑤ 
```
for(int i=1; i<=n; i++)
{
    for (int j=1; j<=100; j++)
        // set of p statements
    for (int k=1; k<=n; k++)
        // set of q statements
}
```
# iterations = $n(100 + n) = 100n + n^2$

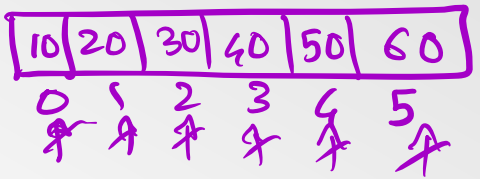
→ $O(n^2)$

# Best case, worst case and average case complexity

- The best-case running time of an algorithm is the running time under ideal or best conditions.
- The worst-case running time of an algorithm offers a guarantee that the running time will never be worse than it.
- The average running time of an algorithm is the expected running time on the average.
- If there is no reference to worst-case or average complexity order, it usually means worst-case.

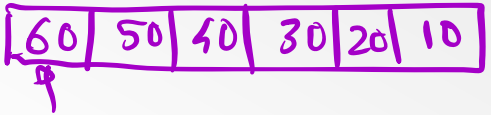
# Find the largest integer in an array of integers (size of the array is n) - Revisited

```
public static int findLargest(int [] a){
  int largest = a[0];                          t
  int i = 1;                                   t
  while (i<a.length)                           nt
  {
      if (a[i] > largest)                      (n-1)t
          largest = a[i];                      (n-1)t
      i++;                                     (n-1)t
  }
  return largest;                              t
}
```



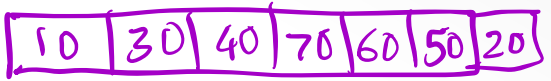

Worst Case: $3t + nt + 3(n-1)t$

$= 4nt \rightarrow O(n)$




Best Case $= 3t + nt + 2(n-1)t$

$\rightarrow 3nt + 1 \rightarrow O(n)$




Average Case $= 3t + nt + 2(n-1)t + (n-1)t \equiv 3.5nt + 1$

$\rightarrow O(n)$

# Summary

- The most reliable way to measure the run time of an algorithm is to count the number of basic operations it performs.
- Express the count of the basic operations of an algorithm as a function of the input size n.
- Then express the function in terms of the Big O.
- Big O refers to the asymptotic growth of the function for large values of n.
- This is the measure that we use to compare algorithms.

# Summary

- Algorithm run time complexity can be best case, worst case or average case.
- Default algorithm time complexity refers to worst case.
- Deriving Big O is easy!
- Some typical run time orders are: O(1) (constant), O(log n) (logarithmic), O(n) (linear),

O( n log n), $O(n^2)$ (quadratic), $O(n^3)$ (cubic), $O(k^n)$ (exponential).

⑥ for (int i=1; i<n; i=2*i)
sum++;

$\boxed{O(\log_2 n)}$

| n | #iterations |
|---|---|
| 1 ($2^0$) | 0 |
| 2 ($2^1$) | 1 |
| 4 ($2^2$) | 2 |
| 8 ($2^3$) | 3 |
| 16 ($2^4$) | 4 |
| $2^x$ | $x$ |
| $n$ | $\log_2 n$ |