SC1007
Data Structures and Algorithms

Introduction to Algorithms and Analysis

Instructor: Luu Anh Tuan

Email: anhtuan.luu@ntu.edu.sg

Office: #N4-02c-86

College of Engineering

School of Computer Science and Engineering

Overview of SC1007

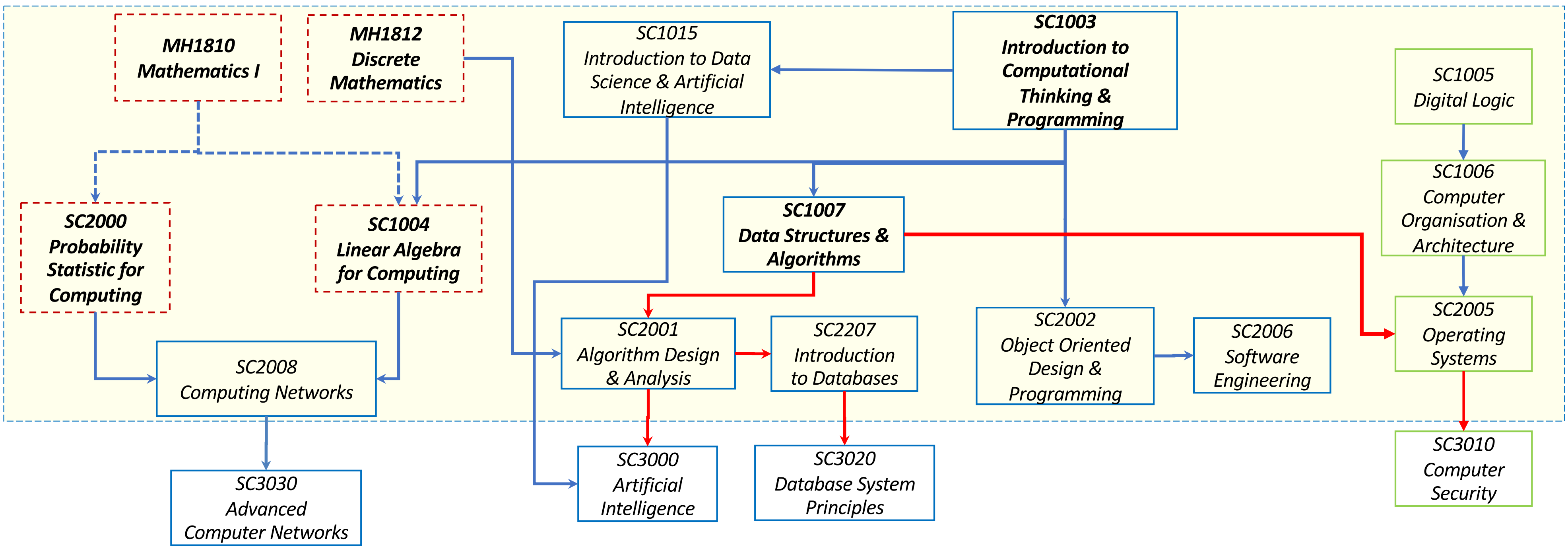
Data Structures:

- Introduce some classical data structures
  - Linear: Linked list, stack, queue
  - Non-linear: tree
- Implement these data structures

Algorithms:

- Analysis of Algorithm – time complexity and space complexity
- Introduce some typical algorithms and their applications

Why Learn Algorithms?



Why Learn Algorithms?

To Continuously Build a Way of Thinking.

What Is An Algorithm?

- An algorithm is a sequence of unambiguous instructions for solving a problem, i.e., for obtaining a required output for any legitimate input in a finite amount of time.

Introduction to The Design & Analysis of Algorithms

-Anany Levitin

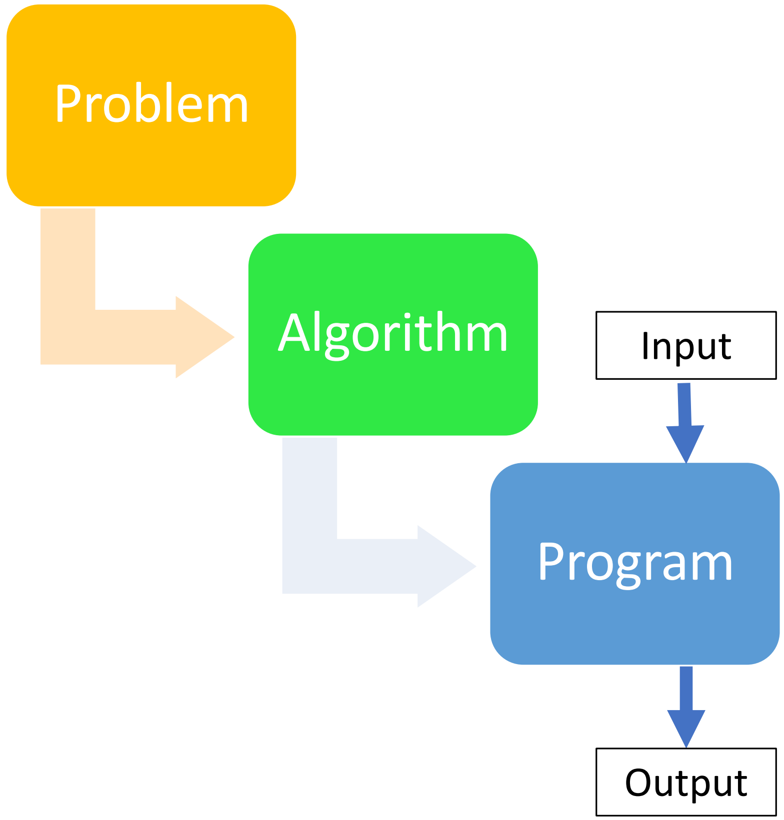
- An algorithm is any well-defined computational procedure that takes some value, or set of values, as input and produces some value, or set of values, as output.

Introduction to Algorithms

-T. H. Cormen et. al.

Algorithm VS Program

- A computer program is an instance, or concrete representation of an algorithm in some programming languages.
- Implementation is the task of turning an algorithm into a computer program.



Example 1: Arithmetic Series

- There are many ways (algorithms) to solve a problem
- Summing up 1 to n

Algorithm 1 Summing Arithmetic Sequence

function Method_One(n)
begin
$sum \leftarrow 0$
for $i = 1$ **to** n **do**
    $sum \leftarrow sum + i$
end

Algorithm 2 Summing Arithmetic Sequ

function Method_Two(n)
begin
$sum \leftarrow n * (1 + n)/2$
end

Algorithm 3 Summing Arithmetic Sequence

function Method_Three(n)
begin
if n=1 **then**
    return 1
else
    return n+Method_Three($n - 1$)
end

```java
import java.util.Scanner;

public class SumNumbers {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Enter a number (n): ");
        int n = scanner.nextInt();
        scanner.close();

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

```java
import java.util.Scanner;

public class SumNumbers {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Enter a number (n): ");
        int n = scann
        scanner.close
        
        int sum = 0;
        for (int i = 
            sum += i;
        }
    }
}
```

```c
#include <stdio.h>

int main() {
    int n, sum = 0;
    printf("Enter a number (n): ");
    scanf("%d", &n);
    for (int i = 1; i <= n; i++) {
        sum += i;
    }
    return 0;
}
```

```
import java.util.Scanner;

public class SumNumbers {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Enter a number (n): ");
        int n = scann
        scanner.close

        int sum = 0;
        for (int i = 
            sum += i;
        }
    }
}
```

```
#include <stdio.h>

int main() {
    int n, sum = 0;
    printf("Enter a number (n): ");
    scanf("%d", &n);
    for (int i = 1; i <= n; i++) {
        sum += i;
    }
    return 0;
}
```

```
n = int(input("Enter a number (n): "))
sum = 0
for i in range(1, n + 1):
    sum += i
```

Example 2: Fibonacci Sequence

- 1, 1, 2, 3, 5, 8, ...
- The n^{th} term is

$$F(n) = F(n-1) + F(n-2)$$

Which is a better algorithm?

Algorithm 4 Fibonacci Sequence: A Simple Recursive Function

```
function Fibonacci_Recursive(n)
begin
if n<1 then
    return 0
if n==1 OR n==2 then
    return 1
return Fibonacci_Recursive(n − 1)+Fibonacci_Recursive(n − 2)
end
```

Algorithm 5 Fibonacci Sequence: A Simple Iterative Function

```
function Fibonacci_Iterative(n)
begin
if n<1 then
    return 0
if n==1 OR n==2 then
    return 1
F_1 ← 1
F_2 ← 1
for i = 3 to n do
    begin
    F_i ← F_{i−2} + F_{i−1}
    F_{i−2} ← F_{i−1}
    F_{i−1} ← F_i
    end
return F_n
end
```

Is there any better algorithm?

Analysis of Algorithms

- The study of the efficiency and performance of algorithms
- Evaluate the speed and scalability of an algorithm
  - How its efficiency changes as input sizes grow
- Identify the most efficient algorithms for a given problem
- Understand the trade-offs between different approaches

Time and space complexities

- Analyze efficiency of an algorithm in two aspects
  - Time
  - Space

- Time complexity: the amount of time used by an algorithm
- Space complexity: the amount of memory units used by an algorithm

Time Complexity or Time Efficiency

1. Count the number of primitive operations in the algorithm

Time Complexity or Time Efficiency

1. Count the number of primitive operations in the algorithm

- Declaration: int x;
- Assignment: x =1;
- Arithmetic operations: +, -, *, /, % etc.
- Logic operations: ==, !=, >, <, &&, ||

These primitive operations take constant time to perform

Basically they are not related to the problem size

changing the input(s) does not affect its computational time

Time Complexity or Time Efficiency

1. Count the number of primitive operations in the algorithm

    i. Repetition Structure: for-loop, while-loop
    ii. Selection Structure: if/else statement, switch-case statement
    iii. Recursive functions

2. Express it in term of problem size

Time Complexity or Time Efficiency

i. Repetition Structure: for-loop, while-loop

```
j ← 1
factorial ← 1
while j ≤ n do
    factorial ← factorial ∗ j
    j ← j + 1
```

c_0

c_1

c_2

c_3

n iterations $\Rightarrow$ $\mathbf{n(c_2+c_3)}$

$f(n) = c_0+c_1+n(c_2+c_3)$

The function increases linearly with n (problem size)

Time Complexity or Time Efficiency

i. Repetition Structure: for-loop, while-loop

```
for j ← 1, m do
    for k ← 1, n do
        sum ← sum + M[ j ][ k ]
```

c_1

n iterations

$n(c_1)$

m iterations

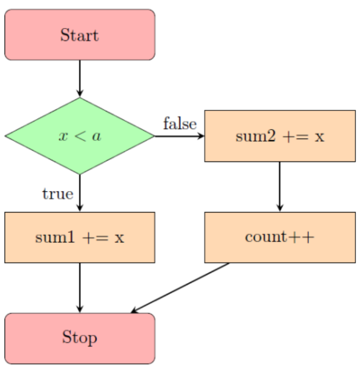
$m(n(c_1))$

The function increases quadratically with n if m==n

*Some constant time operations are ignored here.

Time Complexity or Time Efficiency

ii. Selection Structure: if/else statement, switch-case statement




```
if(x<a)
      sum1 += x;
else {
     sum2 += x;
     count ++;
     }
```

When x < a, only one primitive operation is executed
When x ≥ a, two primitive operations are executed

How do we analyze the time complexity?

1. Best-case analysis
2. Worst-case analysis
3. Average-case analysis

Time Complexity or Time Efficiency

ii. Selection Structure: if/else statement

```
if(x<a)
      sum1 += x;
else {
      sum2 += x;
      count ++;
      }
```

When x < a, only one primitive operation is executed
When x ≥ a, two primitive operations are executed

How do we analyze the time complexity?

1. **Best-case analysis** c_1
2. Worst-case analysis
3. Average-case analysis

Time Complexity or Time Efficiency

ii. Selection Structure: if/else statement

```
if(x<a)
     sum1 += x;
else {
     sum2 += x;
     count ++;
     }
```

When x < a, only one primitive operation is executed
When x ≥ a, two primitive operations are executed

How do we analyze the time complexity?

1. **Best-case analysis**
2. **Worst-case analysis** c_2
3. **Average-case analysis**

Time Complexity or Time Efficiency

ii. Selection Structure: if/else statement

```
if(x<a)
    sum1 += x;
else {
    sum2 += x;
    count ++;
    }
```

When x < a, only one primitive operation is executed
When x ≥ a, two primitive operations are executed

How do we analyze the time complexity?

1. Best-case analysis c_1
2. Worst-case analysis c_2
3. Average-case analysis

$$p(x < a)\ c_1 + p(x \geq a) c_2$$
$$= p(x < a)\ c_1 + (1 - p(x < a)) c_2$$

Time Complexity or Time Efficiency

ii. Selection Structure: switch-case statement

`switch(choice){`	
`case 1: compute the summation; break;`	$\dashrightarrow 5n$
`case 2: search BST; break;`	$\dashrightarrow 6\log_2 n$
`case 3: print BST; break;`	$\dashrightarrow 3n$
`case 4: search for the minimum; break;`	$\dashrightarrow 4\log_2 n$
`}`	

Time Complexity

1. Best-case analysis $\dashrightarrow C + 4\log_2 n$
2. Worst-case analysis $\dashrightarrow C + 5n$
3. Average-case analysis $\dashrightarrow C + \sum_{i=1}^{4} p(i)T_i$

Time Complexity or Time Efficiency

iii. Recursive functions

- Count the number of primitive operations in the algorithm
  - Primitive operations in each recursive call
  - Number of recursive calls

```
int factorial (int n)
{
    if(n==1) return 1;                c2
    else return n*factorial(n-1);     c1
}
```

- *n-1* recursive calls with the cost of c_1.
- The cost of the last call (n==1) is c_2.
- Thus, $c_1(n-1)+c_2$
- It is a linear function

Time Complexity or Time Efficiency

iii. Recursive functions

- Count the number of array[0]==a in the algorithm
  - array[0]==a in each recursive call
  - Number of recursive calls: n-1

```
int count (int array[],int n, int a)
{
    if(n==1)
          if(array[0]==a)
             return 1;
          else return 0;
    if(array[0]==a)
         return 1+ count(&array[1], n-1, a);
    else
         return count (&array[1], n-1, a);
}
```

$W_1 = 1$

$W_n = 1 + W_{n-1}$

$\quad = 1 + 1 + W_{n-2}$

Time Complexity or Time Efficiency

iii. Recursive functions

- Count the number of array[0]==a in the algorithm
  - array[0]==a in each recursive call
  - Number of recursive calls: n-1

```
int count (int array[],int n, int a)
{
    if(n==1)
            if(array[0]==a)
                return 1;
            else return 0;
    if(array[0]==a)
          return 1+ count(&array[1], n-1, a);
    else
          return count (&array[1], n-1, a);
}
```

$W_1 = 1$

$W_n = 1 + W_{n-1}$

$= 1 + 1 + W_{n-2}$

$= 1 + 1 + 1 + W_{n-3}$

…

$= 1 + 1 + … + 1 + W_1$

$= (n – 1) + W_1 = n$

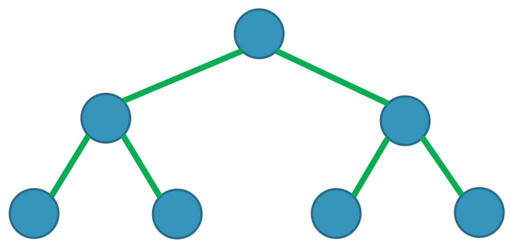
It is known as a **method of backward substitutions**

Time Complexity or Time Efficiency

iii. Recursive functions

- Count the number of multiplication operations in the algorithm

```
preorder (simple_t* tree)
{
    if(tree != NULL){
        tree->item *= 10;
        preorder (tree->left);
        preorder (tree->right);
    }
}
```



Geometric Series:

$$\begin{aligned} S_n &= a + ar + ar^2 + \ldots + ar^{n-1} \\ rS_n &= ar + ar^2 + \ldots + ar^{n-1} + ar^n \\ (1-r)S_n &= a - ar^n \\ S_n &= \frac{a(1-r^n)}{1-r} \end{aligned}$$

$W_0 = 0$

$W_1 = 1$

$W_2 = 1 + W_1 + W_1 = 3$

$W_3 = 1 + W_2 + W_2$

$= 1 + 2\,(1 + W_1 + W_1)$

$= 1 + 2\,(1 + 2)$

$= 1 + 2 + 4 = 7$

$W_{k-1} = 1 + 2 \cdot W_{k-2}$

$= 1 + 2 + 4 + 8 + \ldots + 2^{k-2}$

$W_k = 1 + 2 \cdot W_{k-1} = 1+2+4+8+\ldots+2^{k-1}$

$= \frac{2^k - 1}{2-1} = 2^k - 1$

Prove the hypothesis can be done by mathematical induction

It is known as a **method of forward substitutions**

Series

- Geometric Series

$$G_n = \frac{a(r^n - 1)}{r - 1}$$

- Arithmetic Series

$$A_n = \frac{n}{2}[2a + (n-1)d] = \frac{n}{2}[a_0 + a_{n-1}]$$

- Arithmetico-geometric Series

$$\sum_{t=1}^{k} t2^{t-1} = 2^k(k-1) + 1$$

- Faulhaber's Formula for the sum of the p-th powers of the first n positive integers

$$\sum_{k=1}^{n} k^2 = \frac{n(n+1)(2n+1)}{6}$$

$$\sum_{k=1}^{n} k^3 = \frac{n^2(n+1)^2}{4}$$

Cubic Time Complexity

$$\sum_{k=1}^{n} k^2 = \frac{n(n+1)(2n+1)}{6}$$

```
for (i=1; i<=n; i++)
    M[i] = 0;
    for (j=i; j>0; j--)
        for (k=i; k>0; k--)
            M[i] += A[j]*B[k];
```

- In each outer loop, both j and k are assigned by value of i.
- Inner loops takes i^2 iterations
- The overall number of iterations is

$$1^2 + 2^2 + 3^2 + \cdots + n^2 = \sum_{i=1}^{n} i^2$$

$$= \frac{n(n+1)(2n+1)}{6}$$

Order of Growth

Algorithm	1	2	3	4	5	6
Operation (µsec)	$13n$	$13n\log_2 n$	$13n^2$	$130n^2$	$13n^2+10^2$	2^n

Problem size (n)

10						
100						
10^4						
10^6						

Order of Growth

Algorithm	1	2	3	4	5	6
Operation (µsec)	$13n$	$13n\log_2 n$	$13n^2$	$130n^2$	$13n^2+10^2$	2^n

Problem size (n)

10	.00013	.00043	.0013	.013	.0014	.001024
100	.0013					
10^4	.13					
10^6	13					

Order of Growth

Algorithm	1	2	3	4	5	6
Operation (µsec)	13n	$13n\log_2 n$	$13n^2$	$130n^2$	$13n^2+10^2$	2^n

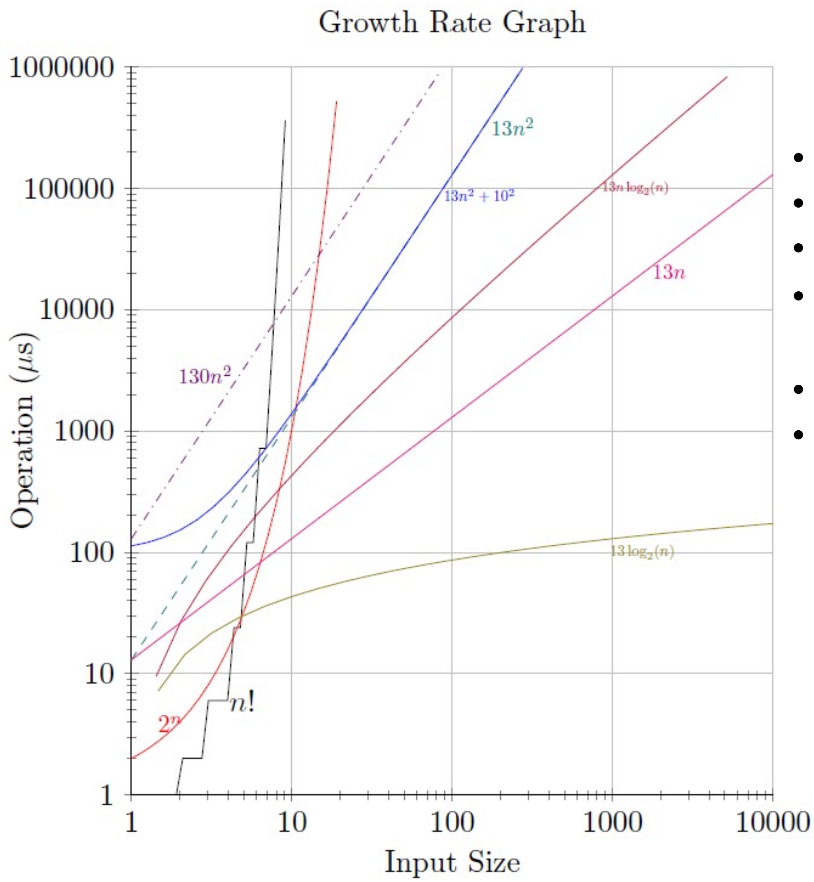
Problem size (n)

10	.00013	.00043	.0013	.013	.0014	.001024
100	.0013	.0086				
10^4	.13	.173				
10^6	13	259				

Order of Growth

Algorithm	1	2	3	4	5	6
Operation (µsec)	$13n$	$13n\log_2 n$	$13n^2$	$130n^2$	$13n^2+10^2$	2^n
Problem size (n)						
10	.00013	.00043	.0013	.013	.0014	.001024
100	.0013	.0086	.13	1.3	.1301	4×10^{16}years
10^4	.13	.173	22 mins	3.61hrs	22mins	
10^6	13	259	150 days	1505 days	150days	

Order of Growth



- $n!$ is the fastest growth
- 2^n is the second
- $13n$ is linear
- $13\log_2 n$ is the slowest

- 10^2 can be ignored when n is large
- $13n^2$ and $130n^2$ have similar growth.
  - $130n^2$ slightly faster

Common Complexity Classes

Order of Growth	Class	Example
1	Constant	Finding midpoint of an array
log_2n	Logarithmic	Binary Search
n	Linear	Linear Search
$nlog_2n$	Linearithmic	Merge Sort
n^2	Quadratic	Insertion Sort
n^3	Cubic	Matrix Inversion (Gauss-Jordan Elimination)
2^n	Exponential	The Tower of Hanoi Problem
n!	Factorial	Travelling Salesman Problem

When time complexity of algorithm A grows faster than algorithm B for the same problem, we say A is inferior to B.

Asymptotic Notations

- Worst-case complexity: Big-Oh (**O**)
- Best-case complexity: Big-Omega (**Ω**)
- Average-case complexity: Big-Theta (**Θ**)

Simplification Rules for Asymptotic Analysis

1. If $f(n) = O\big(cg(n)\big)$ for any constant c > 0, then $f(n) = O(g(n))$
2. If $f(n) = O\big(g(n)\big)$ and $\mathrm{g}(n) = O\big(h(n)\big)$, then $f(n) = O\big(h(n)\big)$

   e.g. $f(n) = 2n$, $\mathrm{g}(n) = n^2$, $\mathrm{h}(n) = n^3$

3. If $f_1(n) = O\big(g_1(n)\big)$ and $f_2(n) = O\big(g_2(n)\big)$,

   then $f_1(n) + f_2(n) = O(\max(g_1(n), g_2(n)))$

   e.g. $5\mathrm{n} + 3\log_2 n = O(n)$

4. If $f_1(n) = O\big(g_1(n)\big)$ and $f_2(n) = O\big(g_2(n)\big)$

   then $f_1(n)f_2(n) = O\big(g_1(n)g_2(n)\big)$

   e.g.$f_1(n) = 3n^2 = O(n^2)$, $f_2(n) = \log_2 n = O(\log_2 n)$

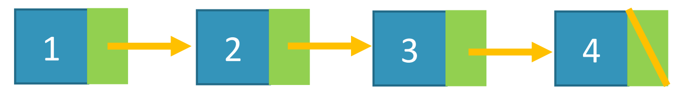
   Then $3n^2 \log_2 n = O(n^2 \log_2 n)$

Time Complexity of Sequential Search

```
pt=head;
while (pt->key != a){
    pt = pt->next;
    if(pt == NULL) break;
}
```

c_1

c_2 (n-1) iterations



Assume that the search key ***a*** is always in the list

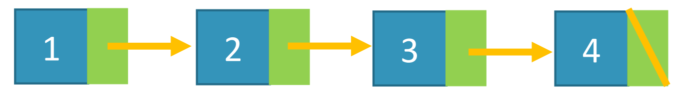
1. Best-case analysis: c_1 when **a** is the first item in the list => $\Theta(1)$
2. Worst-case analysis: $c_2 \cdot (n-1) + c_1$ => $\Theta(n)$
3. Average-case analysis
    - Assumed that every item in the list has an equal probability as a search key

$$\frac{1}{n}[c_1 + (c_1 + c_2) + (c_1 + 2c_2) + \cdots + (c_1 + (n-1)c_2)] = \frac{1}{n}\sum_{i=1}^{n}(c_1 + c_2(i-1))$$

$$= \frac{1}{n}[nc_1 + c_2 \sum_{i=1}^{n}(i-1)\ ]$$

$$= c_1 + \frac{c_2}{n} \cdot \frac{n}{2}(0 + (n-1)) = c_1 + \frac{c_2(n-1)}{2} = \Theta(n)$$

Time Complexity of Sequential Search




```
pt=head;
while (pt->key != a){
    pt = pt->next;
    if(pt == NULL) break;
}
```

c_1

c_2 n iterations

3. Average-case analysis
   - Assumed that every item in the list has an equal probability as a search key

$$\frac{1}{n}[c_1 + (c_1 + c_2) + (c_1 + 2c_2) + \cdots + (c_1 + (n-1)c_2)] = \frac{1}{n}\sum_{i=1}^{n}(c_1 + c_2(i-1))$$

$$= \frac{1}{n}[nc_1 + c_2 \sum_{i=1}^{n}(i-1)\ ]$$

$$= c_1 + \frac{c_2}{n} \cdot \frac{n}{2}(0 + (n-1)) = c_1 + \frac{c_2(n-1)}{2}$$

If the search key, a, is not in the list, then the time complexity is

$$c_1 + nc_2 = \Theta\ (n)$$

Since the probability of the search key is in the list is unknown, we only can have

$$T(n) = P(a\ in\ the\ list)(c_1 + \frac{c_2(n-1)}{2}) + (1 - P(a\ in\ the\ list))(c_1 + nc_2)$$

Hence, it is a linear function. **Θ (n)**

Space Complexity

- Determine number of entities in problem (also called problem size)
- Count number of basic units in algorithm

- Basic units
- Things that can be represented in a constant amount of storage space
- E.g. integer, float and character.

Space Complexity

- Space requirements for an array of n integers - $\Theta(n)$
- If a matrix is used to store edge information of a graph,

  i.e. $G[x][y] = 1$ if there exists an edge from x to y,

  space requirement for a graph with n vertices is $\Theta(n^2)$

<u>Space/time tradeoff principle</u>

- Reduction in time can be achieved by sacrificing space and vice-versa.

Course Schedule (Lectures, Labs, tutorials and assignments)

Week	Topic	Tutorials	Labs	Assignment Released Day
1	Introduction and Memory Management in Python	No Tutorial	No Labs	
2	Linked List (LL)	No Tutorial	No Labs	
3	Linked Lists : Doubly linked Lists and Circular lists.	No Tutorial	Lab 1 (LL)	
4	Stacks and Queues	T1 (LL)	Lab 2 (SQ)	
5	Priority Queues and Arithmetic Expressions	T2 (SQ)	Lab 3 (BT)	
6	Tree Structures: Binary Trees, Binary Search Trees, and AVL Trees	T3 (BT & BST)	Lab 4 (BST)	
7	No Lecture	No Tutorial	No Labs **Lab Test 1**	
		Recess Week		
8	Introduction to algorithms and analysis	No Tutorial	No Labs	
9	Searching	No Tutorial	Lab 5 (Complexity)	
10	Hash Table	T4 (AA + Searching)	Lab 6 (Searching)	AS3: AA + Searching
11	Trie	T5 (Hash Table)	Lab 7 (Hash Table)	AS4: Hash Table + Trie
12	Revision	T6 (Trie)	Lab 8 (Trie)	
13	No Lecture	No Tutorial	No Labs **Lab Test 2 + Final Quiz**	