香港浸會大學
HONG KONG BAPTIST UNIVERSITY

COMP2046

Problem Solving Using Object Oriented Approach

Additional - Recursion

not part of the exam

This course is colocated with *COMP2026* Problem Solving Using Object Oriented Programming

What is Recursion?

香港浸會大學
HONG KONG BAPTIST UNIVERSITY

- A **recursion** (aka recursive method, recursive function) is a method calling itself directly or indirectly.
- Prior to this chapter, we have shown some cases of accidential recursion which ends up as an infinite loop, e.g.

- Lecture 9 - Professor - if we remove `super` (directly call itself)

```
class Person {
  private final String name;
  Person(String name) {this.name = name;}
  protected String getName() {return name; }
}
class Professor extends Person {
  public Professor(String name) {
    super(name);
  }
  public String getName() {
                  //if super is missing
    return "Dr. " + getName();
  }
}
```

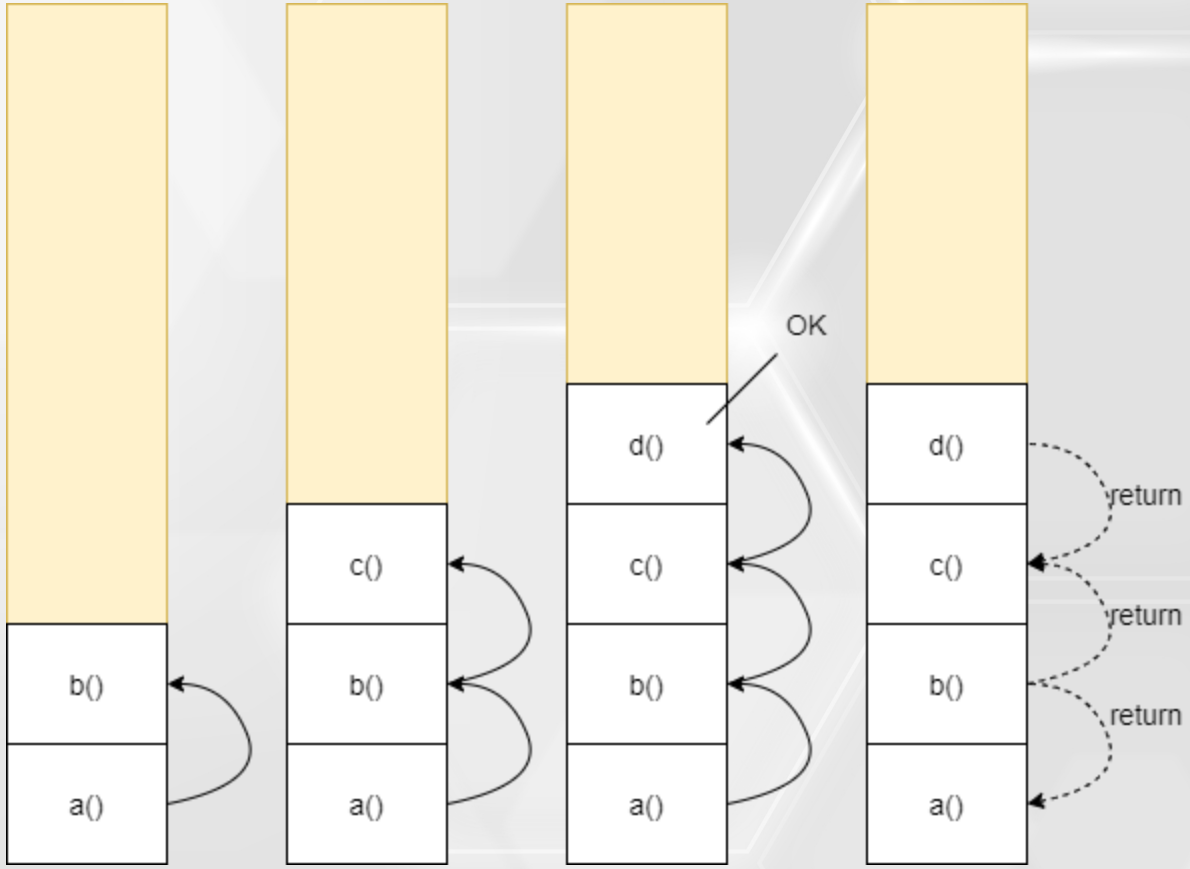
- Lecture 6 - Make Friends (indirectly call itself)

```
public void makeFriend(Person a) {
  if (numOfFriend < 5) {
    friends[numOfFriend++] = a;
    a.makeFriend(this);
  }
}
```

What was happening?

- Stack layout of methods calling

```
void a() { b(); }
void b() { c(); }
void c() { d(); }
void d() { System.out.println("OK"); }
```

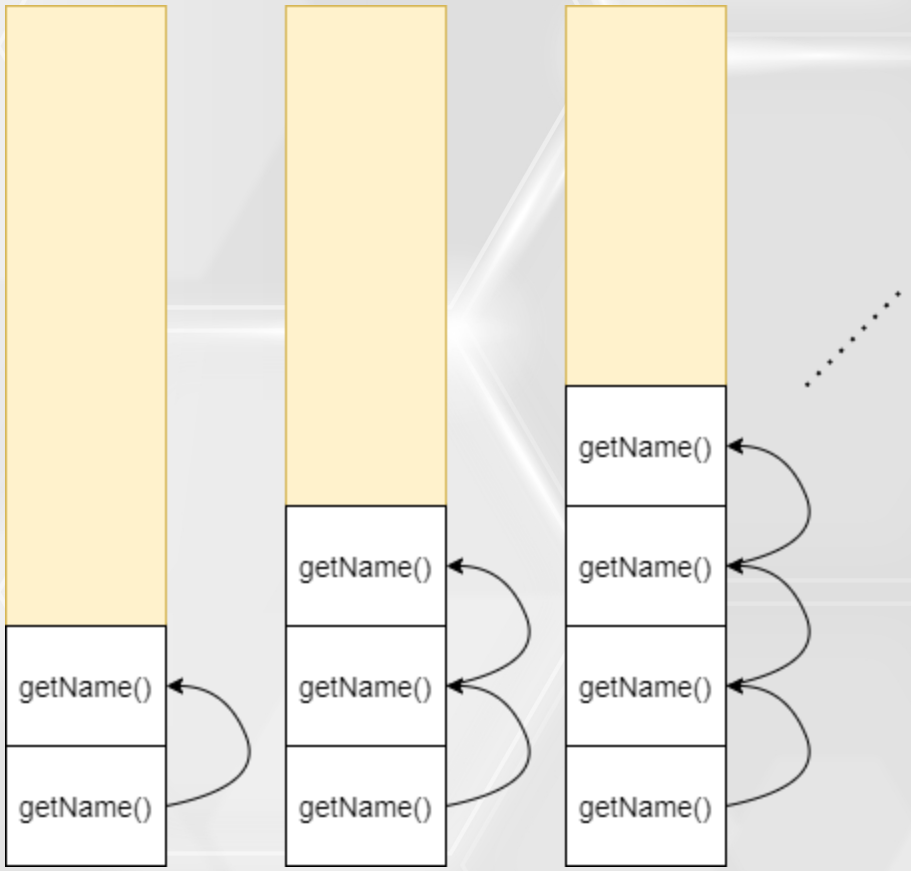


What was happening?

- Stack layout of methods calling

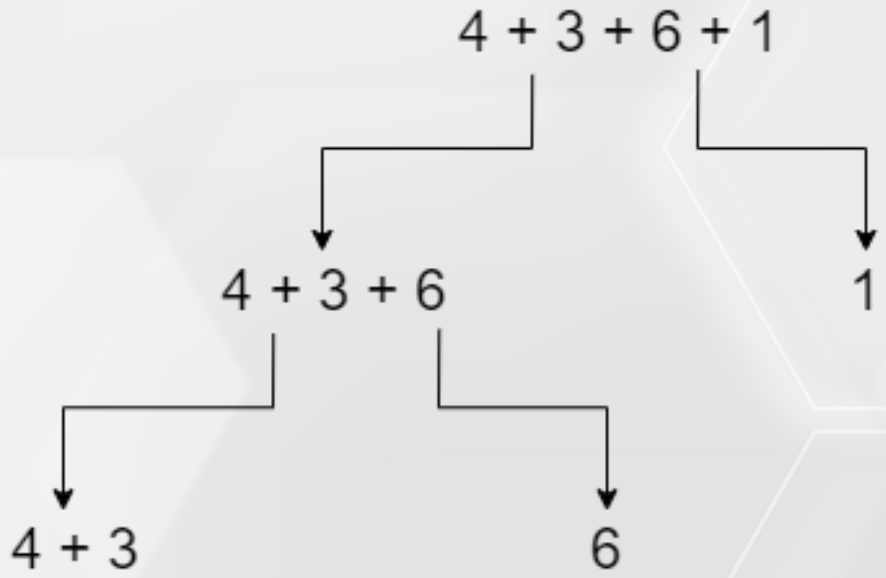
```
class Professor extends Person {
...
  public String getName() {
    return "Dr. " + getName();
  }
}
```

- This cause **Stack Overflow**.



Recursion

- A recursion algorithm can be useful to solve some special problems (e.g. searching problem).
- A recursion breaks down a bigger problem into a smaller problem instance.
  - Divide-and-conquer
- Like mathematical induction (if you know what is it).



A first glance of a working recursion


```
void printNumber(int n) {
  if (n == 0)
    return;
  System.out.print(n + " ");
  printNumber(n-1);
}
void run() {
  printNumber(5);
}
```

```
5 4 3 2 1
```

We loop without loop! 😱

Key-steps of Recursion

1. Always starts with a base case
2. Call yourself at certain point, with a smaller input
3. Return the value to your caller

- **Base case**: some very simple situation that you can actually hardcode it
- **Call yourself**: the recursion. Use yourself to solve the problem
- **Return value**: return value is not a must for recursion. Yet, many problems use return values in their recursion.

Recursion example - Factorial

香港浸會大學
HONG KONG BAPTIST UNIVERSITY

Factorial: $n!$ is defined as $n! = n \times (n-1) \times \cdots 2 \times 1$ with $0! = 1! = 1$

An ordinary factorial method (without recursion)

```
int factorial(int n) {
  int result = 1;
  if (n < 0)
    throw new ArithmeticException("n cannot be negative");
  for (int i = 1; i <= n; i++)
    result *= i;
  return result;
}
```

Recursion example - Factorial #1

香港浸會大學
HONG KONG BAPTIST UNIVERSITY

A recursion of factorial, started with the base case. The simplest case you know how to do it

```
int factorial(int n) {
  if (n < 0)
    throw new ArithmeticException("n cannot be negative");
  if (n == 1 || n == 0)
    return 1;
  // rest of the code...
}
```

Recursion example - Factorial #2

- Now imagine Kevin is a very kind person. He is kind enough to write a method `int X()` for you. This method can help you compute factorial of a smaller instance, say $(n-1)!$
  - You don't challenge Kevin how he implements this method. All you know is `int X()` can really compute $(n-1)!$.
- Now, can you use this `X()` to finish your program?

```
int factorial(int n) {
  if (n < 0)
    throw new ArithmeticException("n cannot be negative");
  if (n == 1 || n == 0)
    return 1;
  // rest of the code...
}
```

Recursion example - Factorial #2 (con't)


```
int factorial(int n) {
  if (n < 0)
    throw new ArithmeticException("n cannot be negative");
  if (n == 1 || n == 0)
    return 1;
  int result = X() * n;
  return result;
}
```

- Why `result = X() * n`?
- If `X()` really can compute $(n-1)!$ ➡ $(n-1)! * n = n!$
- The only problem is
  - Kevin is not that kind to write me `int X()`...

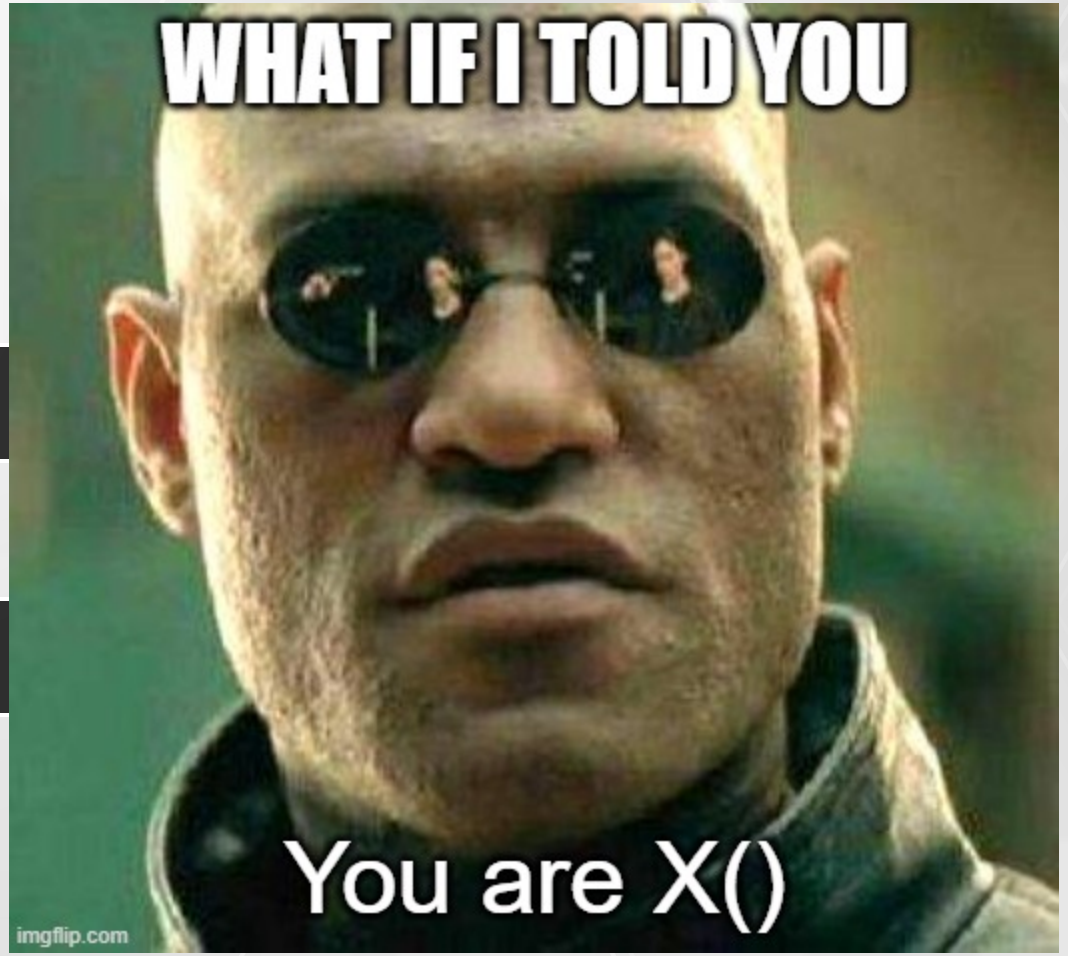
Recursion example

- Revisit what is the ***purpose*** of your method `int factorial(int n)`
  - To compute n factorial!
- So `X()` is indeed `factorial(n-1)`.

```
int result = X() * n;
```

is actually

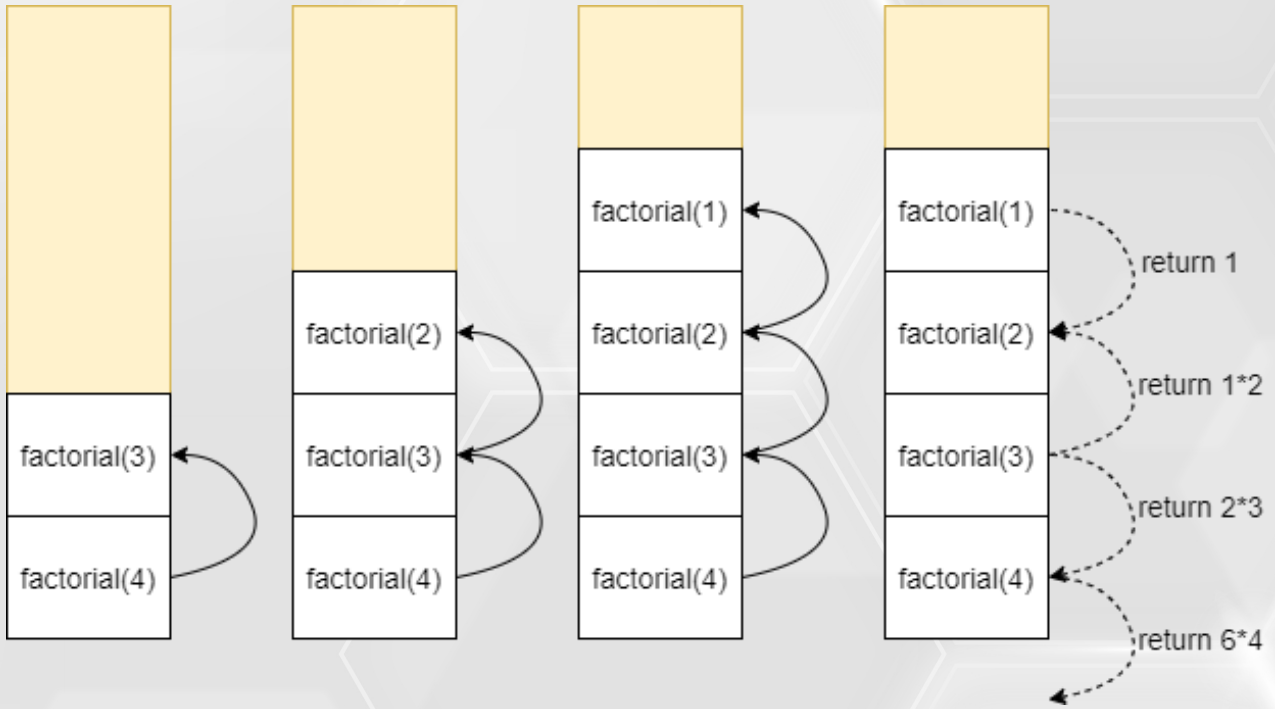
```
int result = factorial(n-1) * n;
```



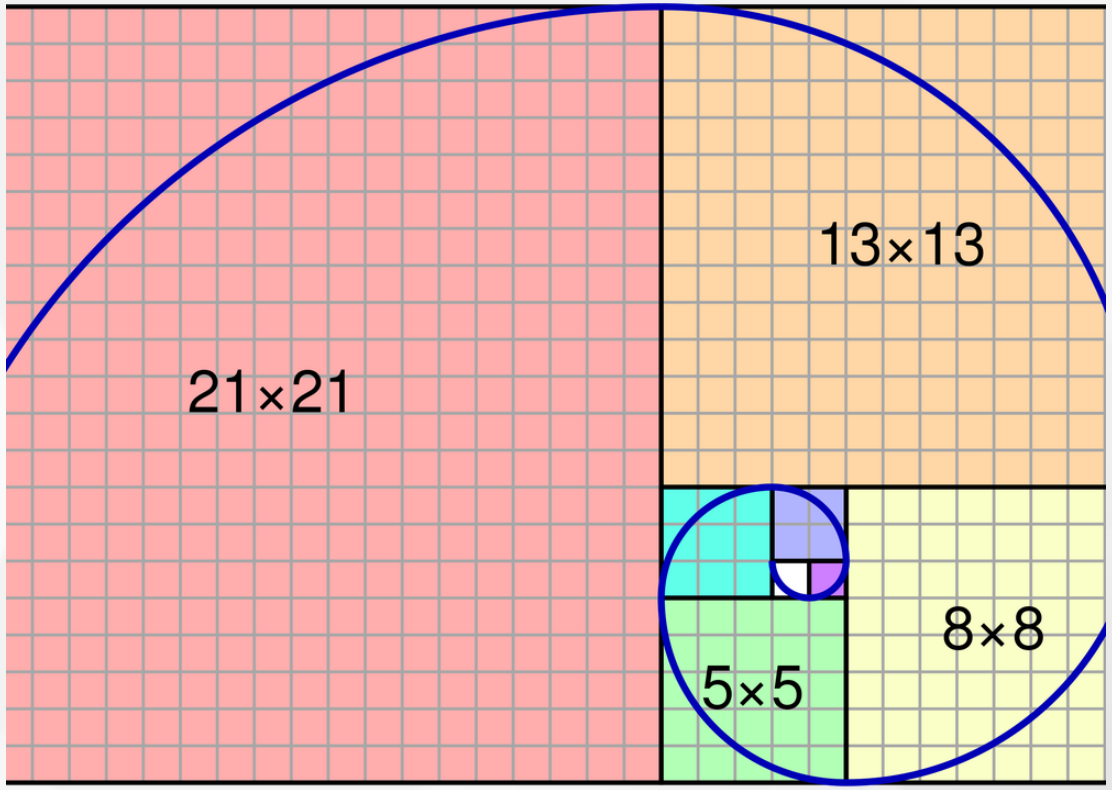
Factorial Recursion


```
int factorial(int n) {
  if (n < 0)
    throw new ArithmeticException("n cannot be negative");
  if (n == 1 || n == 0)
    return 1;
  return factorial(n-1) * n;
}
```

1. Base case: n == 1
2. Call yourself with input = n-1
3. Return value to caller



Fibonacci Sequence - the Golden Ratio



The Fibonacci Sequence

1,1,2,3,5,8,13,21,34,55,89,144,233,377…

1+1=2
1+2=3
2+3=5
3+5=8
5+8=13
8+13=21
13+21=34
21+34=55
34+55=89
55+89=144
89+144=233
144+233=377

$$f(n) = f(n-1) + f(n-2), \forall n \geq 3$$
$$f(1) = f(2) = 1$$

img credit: wiki common; glowscotland

Fibonacci Sequence Do together

1. What is your base case?
2. How would Kevin help you this time?
3. Return value?

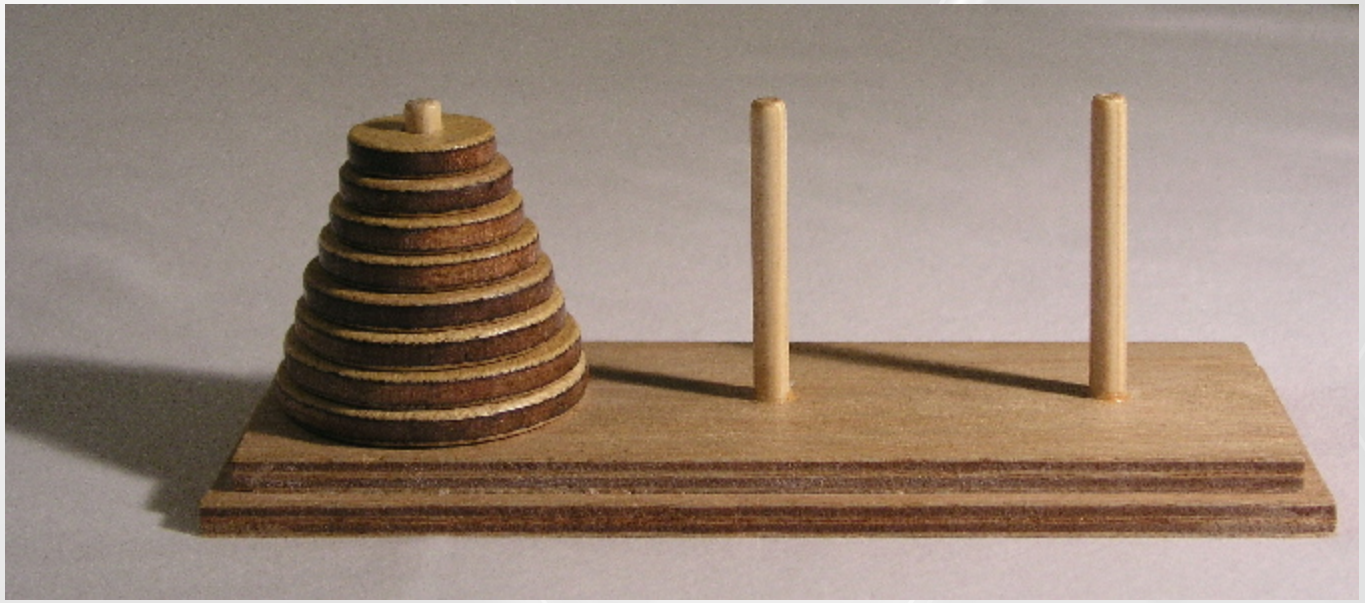
```
int fib(int n) {
  ...
}
```

Towers of Hanoi

- Towers of Hanoi is one of the classic problems every budding computer scientist must grapple with. Legend. Epic. Classic

Rules

- Move one disc at a time
- Big disc cannot on top of a small disc
- Goal: Move the disc from tower A to tower B, using tower C as a buffer



👨‍🏫💬 Let's work out how two-discs, three discs can be moved.

Towers of Hanoi

香港浸會大學
HONG KONG BAPTIST UNIVERSITY

```
public void solveTower(int disks, char src, char dest, char buffer) {
  if (disks == 1) {
    System.out.printf("\n Move %c --> %c", src, dest); //base case
    return;
  }

  //recursion step -- move discs - 1 from src to buffer
  solveTower(disks - 1, src, buffer, dest);
  //move 1 disc from src to dest
  solveTower(1, src, dest, buffer);
  //move discs - 1 from buffer back to dest
  solveTower(disks - 1, buffer, dest, src);
}
```

Kevin: read a few more times. Yeah, I did not get it for the first time too! ▶️

Searching problem with Recursion

- How to find the shortest path from a node to another node?

	a	b	c	d	e	f
a		4	2			
b	4		1	5		
c	2	1		8	10	
d		5	8		2	8
e			10	2		5
f				8	5	

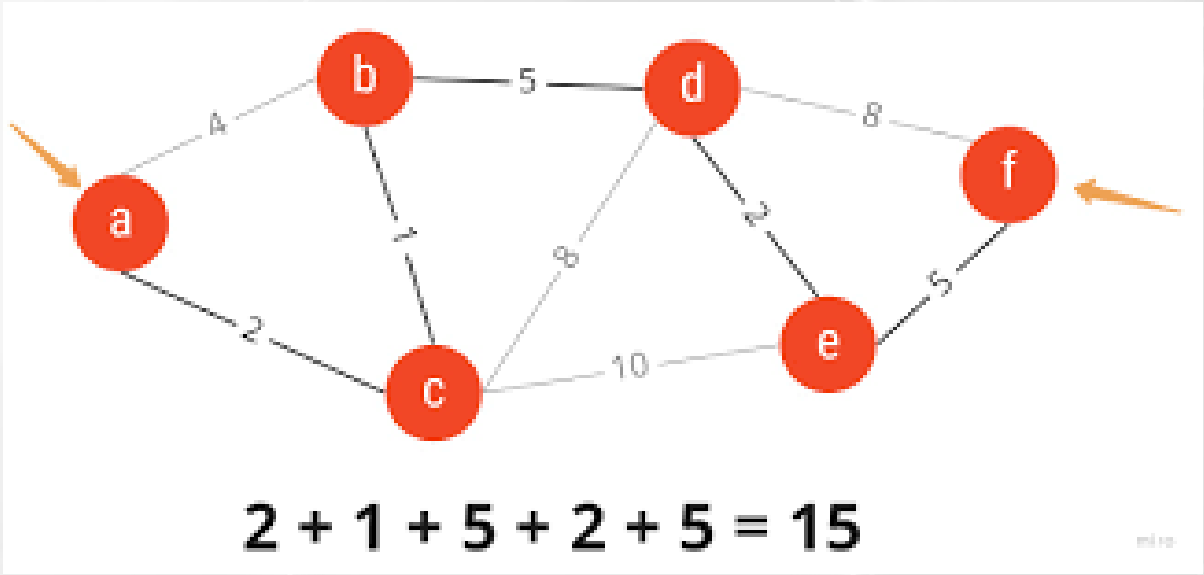


image credit: https://networkx.guide/algorithms/shortest-path/

Searching Problem with Recursion

香港浸會大學
HONG KONG BAPTIST UNIVERSITY

```
boolean search(int steps, int[][] matrix, int src, int dest) {
  if (steps < 0)
    return false;
  if (src == dest) {
    System.out.print((char)('a' + src));
    return true;
  }
  //find the neightbor that can reach the destination
  for (int i = 0; i < matrix.length; i++) {
    int nextStop = i;
    int cost = matrix[src][nextStop];
    if (search(steps - cost, matrix, nextStop, dest)) {
      System.out.printf(" < %c(%d) ", src + 'a', steps);
      return true;
    }
  }
  return false;
}
```

Searching Problem with Recursion

- The algorithm presented above only tell if a path can be searched with a given cost.
- How to find the best path (shortest)?

```
int i = 0;
for (; i < 1000000; i++)
  if (search(i, matrix, src, dest))
    break;
System.out.println("Minimum steps required: " + i);
```

- We can use a better algorithm to find the minimum distance.

Find the minimum distance


```
int minDistance(int[][] matrix, int src,
                int dest, boolean[] visited) {
  if (dest == src)
    return 0;
  if (visited[src])
    return INF;
  visited[src] = true;
  int distance = INF;
  for (int i = 0; i < matrix.length; i++) {
    int d = minDistance(matrix, i, dest, visited)
            + matrix[src][i];
    if (d < distance)
      distance = d; //minimum distance
  }
  visited[src] = false; //back-tracking
  return distance;
}
```

Back-tracking


```
visited[src] = false; //back-tracking
return distance;
```

- Why do we need to set `visited[src] = false;` ?
- Because you need to clean this flag for other searching path!
  - A > B > C > ....
  - A > C > D > ...
  - Both path use C, if you do not unflag C, it cannot be used in another path!
- This unset is called back-tracking.

Auto-Maze

- Refer to your code in Lab 2.
- Build an autopath that reach the destination.

```
*           X
X       X
X X   X     X
      X   X
    X       X X
  X         X X
X         X
              X D
```

```
*  .  .  .  .  .  X
X        X  .
X X   X  .  .  X
      X  .  X
    X  .  .     X X
  X  .  .     X X
X    .  .  X  .  .  .
         .  .  .  X D
```

Auto-maze


```java
/**
* I want to navigate myself from the position
* row col to the final destination 'D'.
* If it is reachable, a path of ...
* will be written from my current position to
* the destination & return true
* If it is not reachable, I will return false
* & clean up the dots that I have written.
*
* @param maze input 2D array
* @param row my row
* @param col my col
* @return true if reachable, false otherwise
*/
boolean autoMaze(char[][] maze, int row, int col) { }
```

Lab - Bridge Crossing

- This is an optional lab.
- To find the solution that cross the bridge within finite time.
- No marks will be given for this lab.

Hall of Fame

- Year 2021-22
  - Zhuang Ruiqi
- Year 2022-23
  - Cheung Yui Haang
  - Lee Ka Yan
  - UCHE Destiney Nnanna