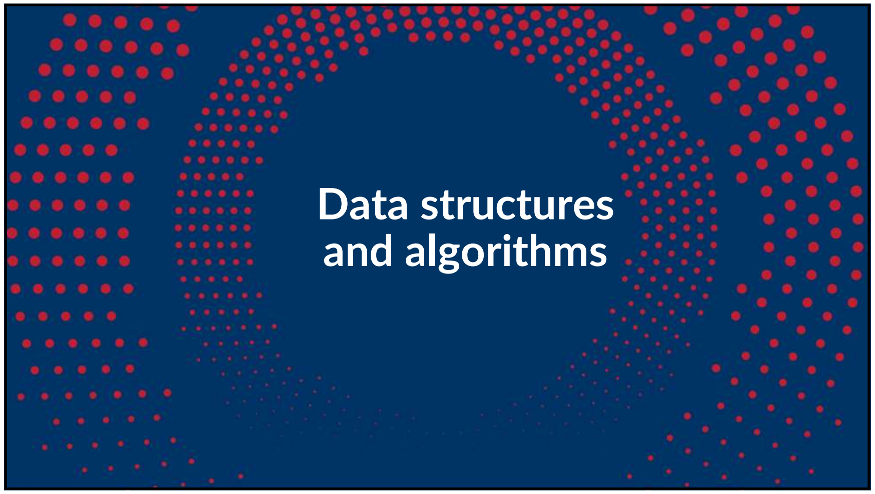
HUST

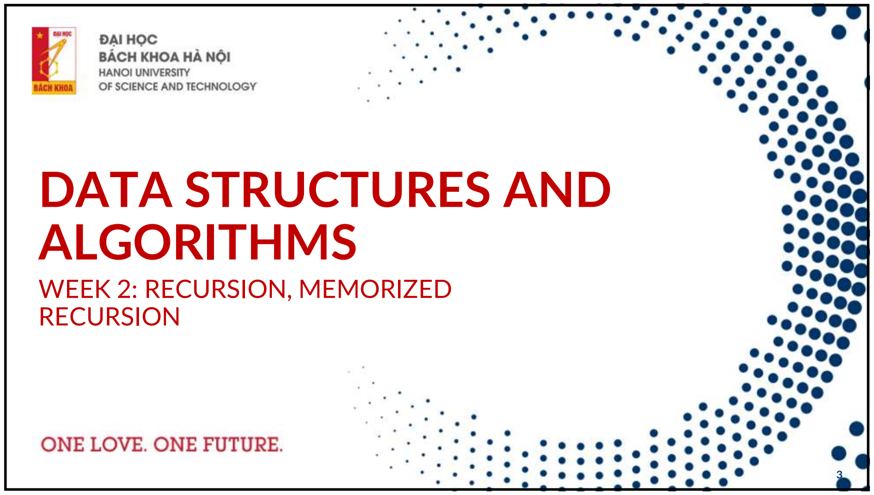
ĐẠI HỌC BÁCH KHOA HÀ NỘI

HANOI UNIVERSITY OF SCIENCE AND TECHNOLOGY

ONE LOVE. ONE FUTURE.



Data structures and algorithms

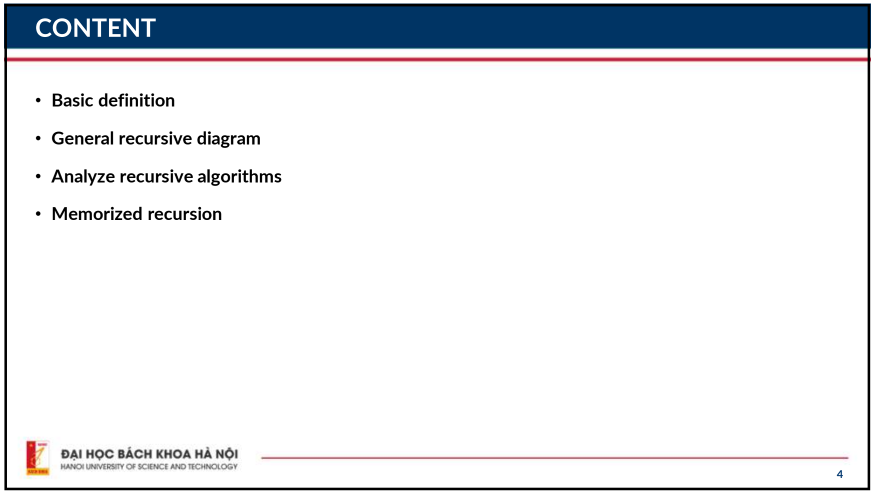


ĐẠI HỌC BÁCH KHOA HÀ NỘI
HANOI UNIVERSITY OF SCIENCE AND TECHNOLOGY

DATA STRUCTURES AND ALGORITHMS

WEEK 2: RECURSION, MEMORIZED RECURSION

ONE LOVE. ONE FUTURE.



CONTENT

- Basic definition
- General recursive diagram
- Analyze recursive algorithms
- Memorized recursion

BASIC DEFINITION

- Recursive object: defined through itself but on a smaller scale
- Recursive function
  - Basic step: Determine the value of a function with some initial parameter values
  - Recursive step: Determine the relationship between the function depending on the same function but with smaller parameters

- Basic: F(n) = 1, với n = 1
- Recursion: F(n) = F(n-1) + n, with n > 1

- Basic: C(k, n) = 1, với k = 0 hoặc k = n
- Recursion: C(k, n) = C(k-1,n-1) + C(k,n-1), for others

BASIC DEFINITION

- Recursive object: defined through itself but on a smaller scale
- The set is defined recursively
  - Basic step: determine the first elements of the set
  - Recursive step: determine the rule indicating that larger word parts belong to the set of original elements

- Basic: 3 belongs to S
- Recursion: If x and y belong to S then x + y belongs to S

GENERAL RECURSIVE DIAGRAM

- A recursive algorithm is an algorithm that calls itself with a smaller input size.
- Recursive algorithms are often used when needing to deal with recursively defined objects.
- Example: The recursive definition of a Fibonacci sequence:
  - $f(0) = 0$, $f(1) = 1$,
  - $f(n) = f(n-1) + f(n-2)$ with $n > 1$
- High-level programming languages often allow the construction of recursive functions, meaning that the body of the function contains calls to itself. Therefore, when implementing recursive algorithms, people often build recursive functions.

GENERAL RECURSIVE DIAGRAM

```
Recursive(input) {
   if (size of input is minimum) then
         Do basic steps;  /* Solve the problem with the smallest size*/
  else  {
         Recursive(input with smaller size);  /* Recursive step for
subproblems*/
         /* Note: There might exist other recursive calls*/
         Combine results of subproblems to get solution;
         return solution;
  }
}
```

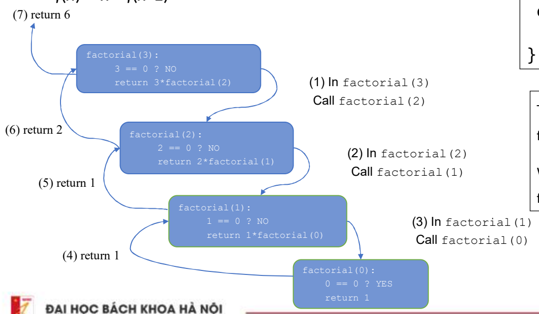
EXAMPLE

- **Example 1: Calculate *n*! with the recursive formulation:**

$f(0) = 1$

$f(n) = n * f(n-1)$

```
int factorial(int n){
 if (n==0)
     return  1;
 else
     return n*factorial(n-1);
}
```



(7) return 6

```
factorial(3):
     3 == 0 ? NO
     return 3*factorial(2)
```

(1) In `factorial(3)` Call `factorial(2)`

(6) return 2

```
factorial(2):
     2 == 0 ? NO
     return 2*factorial(1)
```

(2) In `factorial(2)` Call `factorial(1)`

(5) return 1

```
factorial(1):
     1 == 0 ? NO
     return 1*factorial(0)
```

(3) In `factorial(1)` Call `factorial(0)`

(4) return 1

```
factorial(0):
     0 == 0 ? YES
     return 1
```

The execution of factorial(3) will stop when factorial(2) returns a result

When factorial(2) return a result, factorial(3) continues executing.

EXAMPLE

- **Example 2: Calculate the Fibonacci sequence:**

$F(0) = 0; F(1) = 1$

$F(n) = F(n-1) + F(n-2)$ với $n \geq 2$

```
int F(int n){
 if (n < 2)
     return  n;
 else
     return F(n-1) + F(n-2);
}
```

EXAMPLE: HANOI TOWER

- Example 3: Hanoi Tower:

The Tower of Hanoi problem is presented as follows: “There are 3 piles a, b, c. On pile a there is a stack of n disks, the diameter decreasing from bottom to top. It is necessary to move the stack of disks from pile a to pile c following the rules:

- Only transfer 1 disk at a time
- Only discs with smaller diameters should be placed on top of disk with larger diameters. During the transfer process, it is allowed to use pile b as an intermediate pile.

The problem is: List the steps to move the disks that need to be performed to complete the task set out in the problem.

EXAMPLE: HANOI TOWER

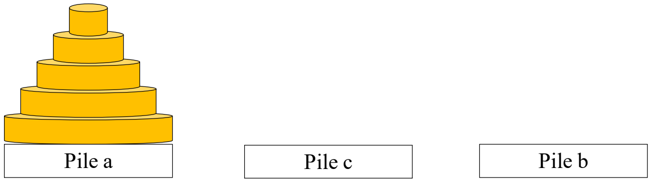
Moving n disks from pile a to pile c using pile b as an intermediary: **HanoiTower**(n, a, c, b);

Disc movement consists of 3 stages:

(1) Move n-1 disks from pile a to pile b, using pile c as an intermediary

(2) Move 1 disk (the disk with the largest diameter) from pile a to pile c

(3) Move n-1 disks from pile b to pile c, using pile a as an intermediary



EXAMPLE: HANOI TOWER

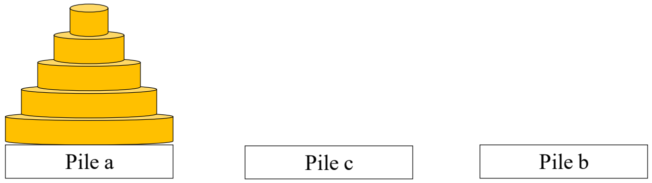
Moving n disks from pile a to pile c using pile b as an intermediary: **HanoiTower**(n, a, c, b);

Disc movement consists of 3 stages:

(1) Move n-1 disks from pile a to pile b, using pile c as an intermediary
Solve a problem of size n-1: HanoiTower(n-1,a,b,c)

(2) Move 1 disk (the disc with the largest diameter) from pile a to pile c
Solve a problem of size 1: HanoiTower(1,a,c,b)

(3) Move n-1 disks from pile b to pile c, using pile a as an intermediary
Solve a problem of size n-1: HanoiTower(n-1,b,c,a)



EXAMPLE: HANOI TOWER

The algorithm can be described in the following recursive procedure:

```
HanoiTower(n, a, c, b) {//Move n-1 disks from pile a to pile b, using pile c as an intermediary
    if (n==1) then <move a disk from pile a to pile c>
    else {
         HanoiTower(n-1,a,b,c);
         HanoiTower(1,a,c,b);
         HanoiTower(n-1,b,c,a);
    }
}
```

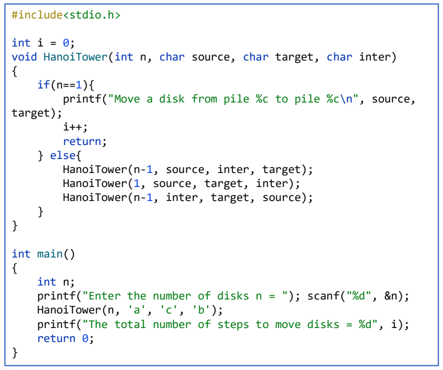
Disc movement consists of 3 stages:

(1) Move n-1 disks from pile a to pile b, using pile c as an intermediary

(2) Move 1 disk (the disk with the largest diameter) from pile a to pile c

(3) Move n-1 disks from pile b to pile c, using pile a as an intermediary

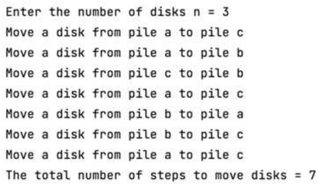
EXAMPLE: HANOI TOWER



```
#include<stdio.h>

int i = 0;
void HanoiTower(int n, char source, char target, char inter)
{
    if(n==1){
        printf("Move a disk from pile %c to pile %c\n", source,
target);
        i++;
        return;
    } else{
        HanoiTower(n-1, source, inter, target);
        HanoiTower(1, source, target, inter);
        HanoiTower(n-1, inter, target, source);
    }
}

int main()
{
    int n;
    printf("Enter the number of disks n = "); scanf("%d", &n);
    HanoiTower(n, 'a', 'c', 'b');
    printf("The total number of steps to move disks = %d", i);
    return 0;
}
```



```
Enter the number of disks n = 3
Move a disk from pile a to pile c
Move a disk from pile a to pile b
Move a disk from pile c to pile b
Move a disk from pile a to pile c
Move a disk from pile b to pile a
Move a disk from pile b to pile c
Move a disk from pile a to pile c
The total number of steps to move disks = 7
```

ANALYZE RECURSIVE ALGORITHMS

- To analyze recursive algorithms, we usually proceed as follows:
  - Let **T(n)** be the algorithm's calculation time
  - Build a recursive formula for **T(n)**
  - Solve the resulting recursive formula to give an estimate for **T(n)**

In general, we only need a close estimate of the growth rate of **T(n)**, so solving the recursive formula for T(n) is to give an estimate of the growth rate of **T(n)** in asymptotic notation.

ANALYZE RECURSIVE ALGORITHMS

Example: Calculate *n*! with the recursive formula:

$f(0) = 1$

$f(n) = n * f(n-1)$

```
int factorial(int n){
   if (n==0)
      return  1;
   else
      return n*factorial(n-1);
}
```

- Let $T(n)$ be the number of multiplication operations that must be performed in the call to factorial(n).
- We have:

$T(0) = 0,$

$T(n) = T(n-1) +1, n≥1$

ANALYZE RECURSIVE ALGORITHMS

Example: Calculate *n*! with the recursive formula:

$f(0) = 1$

$f(n) = n * f(n-1)$

```
int factorial(int n){
   if (n==0)
      return  1;
   else
      return n*factorial(n-1);
}
```

- Let $T(n)$ be the number of multiplication operations that must be performed in the call to factorial(n).
- We have:

$T(0) = 0,$

$T(n) = T(n-1) +1, n≥1$

ANALYZE RECURSIVE ALGORITHMS

- Let $T(n)$ be the number of multiplication operations that must be performed in the call to factorial(n). We have:

$T(0) = 0,$

$T(n) = T(n-1) +1, n≥1$

- Solving the recursive formula T(n), we have:

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

$= T(n-2) + 1 + 1$ replace $T(n-2)$

$= T(n-3) + 1 + 1 + 1$

$= T(n-3) + 3$

$= \ldots$

$= T(n-k) + k$

$T(n) = T(n-n) + n$ *Select* $k = n$

$= T(0) + n$

$= n$. So that: $T(n) = O(n)$

MEMORIZED RECURSION

```
int F(int n){
 if (n < 2)  return  n;
 else return F(n-1) + F(n-2);
}
```

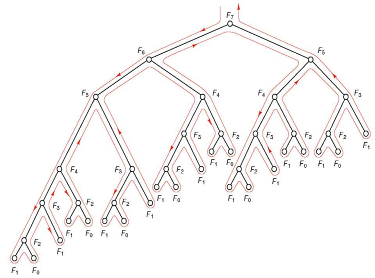
- Duplicate subproblems

Example 1: Calculate the Fibonacci sequence :

$F(0) =0; F(1) = 1$

$F(n) = F(n-1) + F(n-2)$ với $n ≥ 2$

- In a recursive algorithm, every time we need a solution to a subproblem, we have to solve it recursively. Therefore, there are subproblems that are solved over and over again. That leads to ineffectiveness of the algorithm. This phenomenon is called duplicate subproblem phenomenon.



MEMORIZED RECURSION

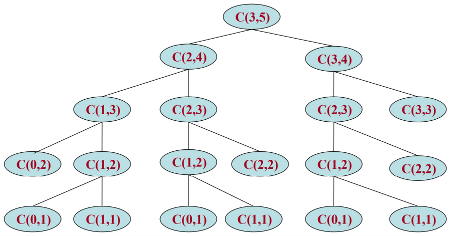
- Duplicate subproblems

Example 2: Calculate the binomial coefficient:

$C(0,n) = 1$, $C(n,n) = 1$; for all $n \geq 0$,

$C(k,n) = C(k-1,n-1)+C(k,n-1)$, $0 < k < n$

```
int C(int k, int n){
   if (k == 0|| k == n)return 1;
   else return C(k-1,n-1)+C(k,n-1);
}
```



MEMORIZED RECURSION

- In the two examples above, we have seen that the recursive algorithms for calculating Fibonacci numbers and calculating binomial coefficients are inefficient.
- To increase the efficiency of recursive algorithms, we can use memorized recursion techniques.
  - Using memorized recursion techniques, in many cases, we can preserve the recursive structure of the algorithm and at the same time ensure its efficiency. The biggest disadvantage of this approach is the memory requirement.

MEMORIZED RECURSION

- The idea of recursion:
  - Use a variable to remember information about the solution of subproblems right after the first time it is solved. That allows to shorten the algorithm's calculation time, because, whenever needed, it can be looked up without having to re-solve previously solved sub-problems.

MEMORIZED RECURSION

- Example 1: Calculate a Fibonacci sequence:

$F(0) = 0$; $F(1) = 1$

$F(n) = F(n-1) + F(n-2)$ with $n \geq 2$

Non-memorized recursion:

```
int F(int n){
 if (n < 2)
     return  n;
 else
     return F(n-1) + F(n-2);
}
```

Memorized recursion:

```
void init() {
   M[0] = 0; M[1] = 1;
   for (int i = 2; i <= n; i++) M[i] = 0;
}
int F(int n){
  if (n!= 0 && M[n] == 0)
      M[n] = F(n-1) + F(n-2);
  return M[n];
}
```

Before calling the function F(n), call the init() function to initialize the elements in the array M[ ] as follows:

M[0] = 0, M[1]=1,

M[*i*] = 0, với *i* ≥ 2.

MEMORIZED RECURSION

- Example 2: Calculate the binomial coefficient :

$C(0,n) = 1$, $C(n,n) = 1$;   for all $n \geq 0$,

$C(k,n) = C(k-1,n-1)+C(k,n-1)$, $0 < k < n$

Non-memorized recursion

```
int C(int k, int n){
   if (k == 0 || k == n)
     return 1;
   else
     return C(k-1,n-1) + C(k,n-1);
}
```

Memorized recursion:

```
void init() {
   for (int i = 0; i <= k; i++)
     for(int j = 0; j <= n; j++) M[i][j] = 0;
}
int C(int k, int n) {
   if (k == 0 || k == n) M[k][n] = 1;
   else {
     if(M[k][n] == 0) M[k][n] = C(k-1,n-1) + C(k,n-1);
   }
   return M[k][n];
}
```

Before calling the function C(k, n), call the init() function to initialize the elements in the array M[ ][ ] = 0

THANK YOU !