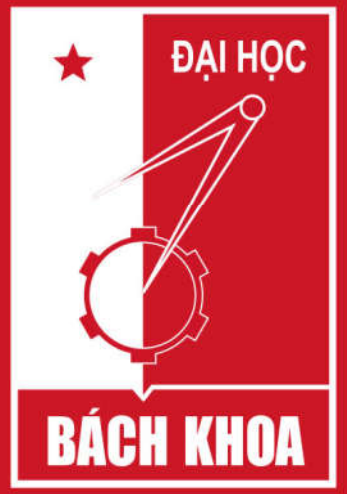


HA NOI UNIVERSITY OF SCIENCE AND TECHNOLOGY
SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGY

HA NOI UNIVERSITY OF SCIENCE AND TECHNOLOGY
SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGY

C Programming Basic

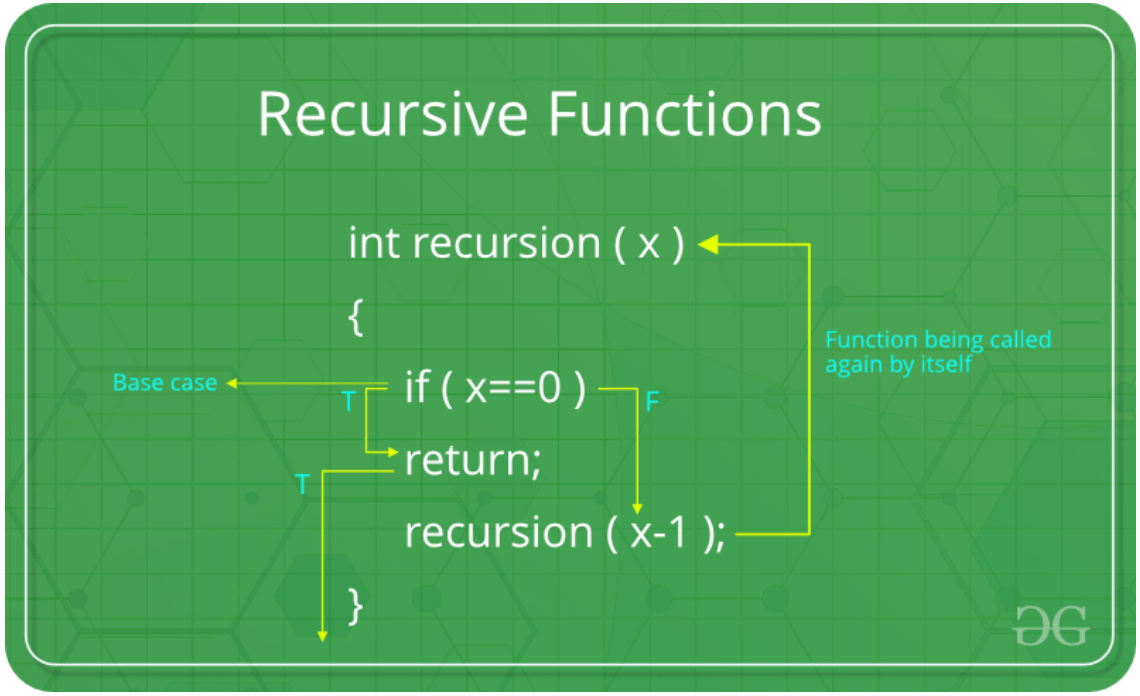
Recursion

Content

- Recursion
- Recursion with memorization
- Backtracking

Recursion

- Recursive function
  - Call itself (with smaller input parameters)
  - Base case
    - Parameters are small so that the results are obtain easily
    - The function does not call itself



Recursion

- Recursive function
  - Call itself (with smaller input parameters)
  - Base cases
    - Parameters are small so that the results are obtain easily
    - The function does not call itself

- $f(n) = 1 + 2 + \ldots + n$

Other form

- $f(n) = \begin{cases} 1, \text{ if } n = 1 \\ f(n-1) + n, \text{ if } n > 1 \end{cases}$

```
#include <stdio.h>
int f(int n){
    if(n == 1) return 1;
    return n + f(n-1);
}
int main(){
    printf("%d\n",f(4));
}
```

Recursion

- Fibonacci sequence: 1, 1, 2, 3, 5, 8, 13, 21, 34, …

$$f(n) = \begin{cases} 1, \text{ if } n = 0 \text{ or } n = 1 \\ f(n-1) + f(n-2), \text{ if } n > 1 \end{cases}$$

```
#include <stdio.h>
int f(int n){
    if(n <= 1) return 1;
    return f(n-1) + f(n-2);
}
int main(){
    for(int i = 0; i <= 10; i++)
        printf("%d ",f(i));
}
```

Recursion

- How many ways to select *k* objects from *n* given objects

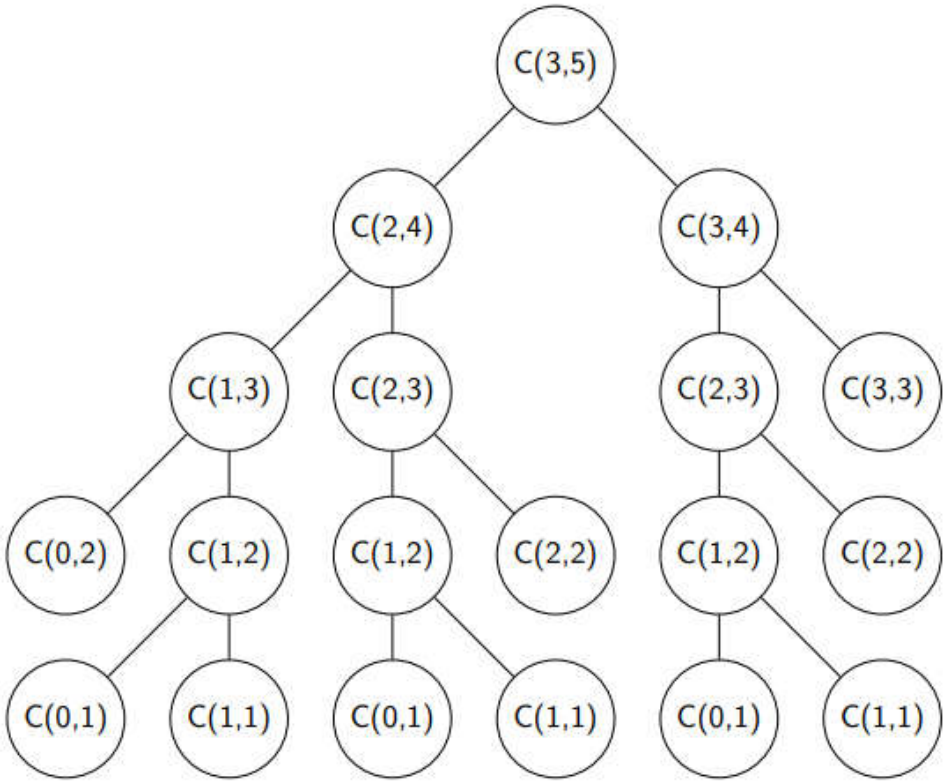
$$C(k,n) = \begin{cases} 1, \text{ if } k = 0 \text{ or } k = n \\ C(k,n-1) + f(k-1,n-1), \text{ otherwise} \end{cases}$$

```
#include <stdio.h>
int C(int k, int n){
    if(k == 0 || k == n) return 1;
    return C(k,n-1) + C(k-1,n-1);
}
int main(){
    printf("%d ",C(3,5));
}
```

Recursion with memorization

- How many ways to select *k* objects from *n* given objects

$$C(k,n) = \begin{cases} 1, \text{ if } k = 0 \text{ or } k = n \\ C(k,n-1) + f(k-1,n-1), \text{ otherwise} \end{cases}$$

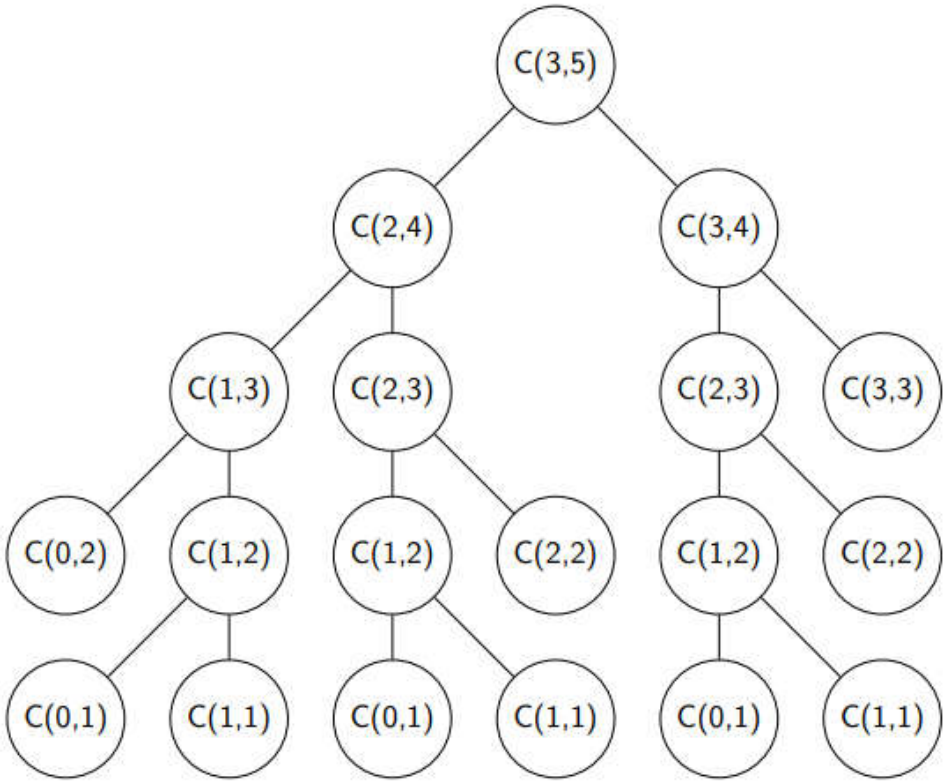


Recursion with memorization

- How many ways to select *k* objects from *n* given objects

$$C(k,n) = \begin{cases} 1, \text{ if } k = 0 \text{ or } k = n \\ C(k,n-1) + f(k-1,n-1), \text{ otherwise} \end{cases}$$

- Redundant computation
  - A function (with the same parameters) is called several times)



Recursion with memorization

- Solution
  - Use memory to record the results of functions
  - A function with given parameters is mapped to a memory location
  - The first time, let the function run and store the result into the corresponding memory
  - Later, only let the function run only if the result has not been available into the memory

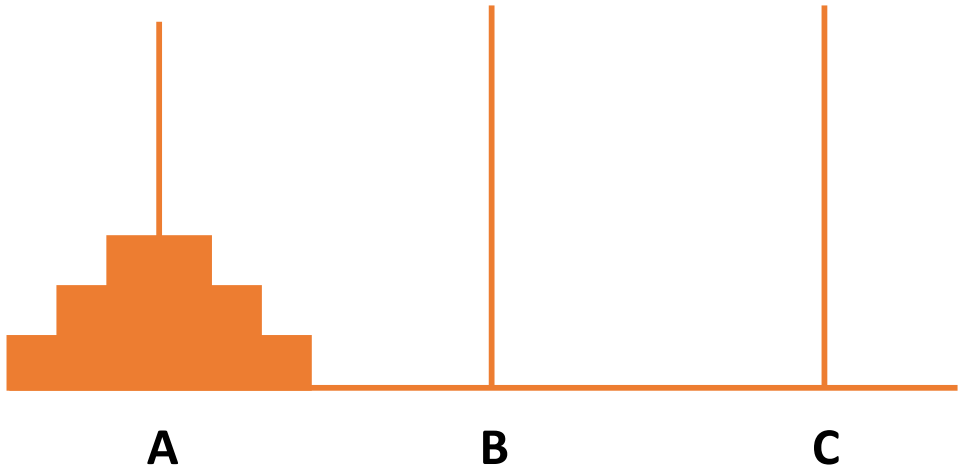
Recursion with memorization

- Solution
  - Use memory to record the results of functions
  - A function with given parameters is mapped to a memory location
  - The first time, let the function run and store the result into the corresponding memory
  - Later, only let the function run only if the result has not been available into the memory

```
#include <stdio.h>
#define MAX 100
int M[MAX][MAX];// M[k][n] store the value of
                // C(k,n)
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,n-1) + C(k-1,n-1);
    return M[k][n];
}
int main(){
    memset(M,0,sizeof(M));
    printf("%d ",C(3,5));
}
```

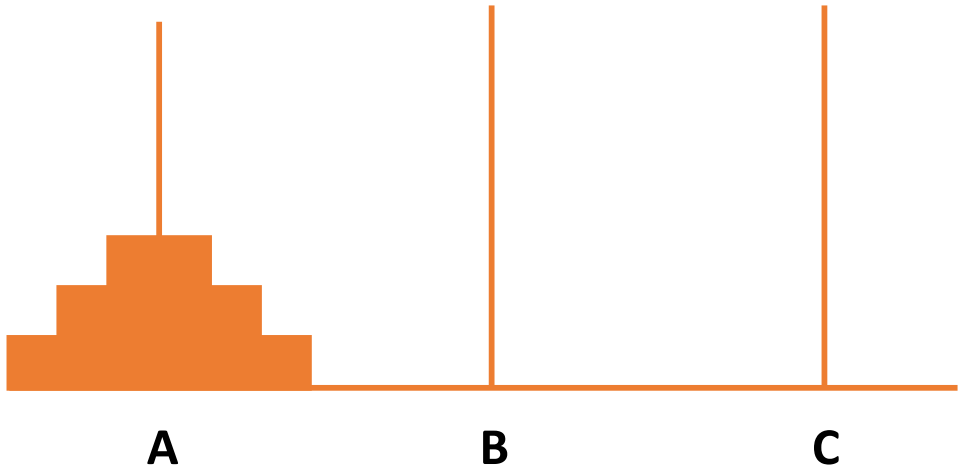
Hanoi tower

- There are *n* disks with different sizes and 3 rods A, B, C
- Initialization: *n* disks are in the rod A such that larger disks are below smaller ones



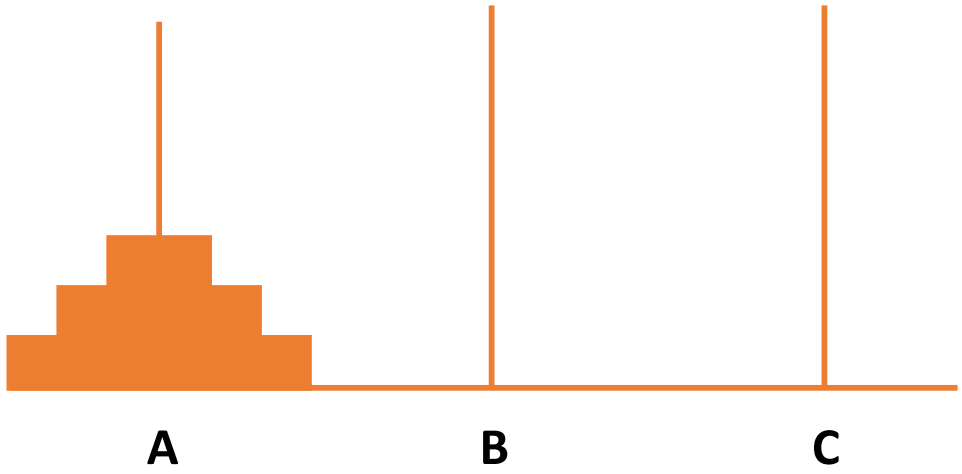
Hanoi tower

- There are *n* disks with different sizes and 3 rods A, B, C
- Initialization: *n* disks are in the rod A such that larger disks are below smaller ones
- Goal: Move *n* disks from A to B such that
  - Only one disk can be moved at a time
  - Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack or on an empty rod
  - No larger disk may be placed on top of a smaller disk



Hanoi tower

- There are *n* disks with different sizes and 3 rods A, B, C
- Initialization: *n* disks are in the rod A such that larger disks are below smaller ones
- Goal: Move *n* disks from A to B such that
  - Only one disk can be moved at a time
  - Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack or on an empty rod
  - No larger disk may be placed on top of a smaller disk



Lời giải

B1: A → B

B2: A → C

B3: B → C

B4: A → B

B5: C → A

B6: C → B

B7: A → B

Hanoi tower

```
#include <stdio.h>
int cnt = 0;
void move(int n, char A, char B, char C){
    if(n == 1){
        cnt++;
        printf("Step %d: Move a disk from %c to %c\n",cnt,A,B);
    }else{
        move(n-1,A,C,B);
        move(1,A,B,C);
        move(n-1,C,B,A);
    }
}
int main(){
    move(3,'A','B','C');
}
```

Backtracking

- Solve combinatorial configuration generation or optimization problems
- $A = \{(x_1, x_2, \ldots, x_n) \mid x_i \in A_i, \forall\, i = 1,\ldots, n\}$
- Generate all $x \in A$ satisfying a constraint P
- Procedure TRY(k):
  - Try all value v that can be assigned to x_k without violating the constraint P
  - For each feasible value v:
    - Assign v to x_k
    - If $k < n$: call recursively TRY(k+1) to try values for x_{k+1}
    - If $k = n$: get a solution

Backtracking

```
TRY(k)
  Begin
    Foreach v of A_k
     if check(v,k) /* check if v is feasible */
       Begin
         x_k = v;
         [may be: update incremental data structure D]
         if(k = n) solution();
         else TRY(k+1);
         [may be: recover D when backtracking]
       End
  End
Main()
Begin
  TRY(1);
End
```

Backtracking

- Generate all binary sequences of length n

Backtracking

- Generate all binary sequences of length n

```
#include <stdio.h>
#define N 100
int n;
int x[N];
void solution(){
    for(int i = 1; i <= n; i++)
       printf("%d ",x[i]);
    printf("\n");
}
int check(int v, int k){
    return 1;
}
```

```
void Try(int k){
    for(int v = 0; v <= 1; v++){
        if(check(v,k)){
            x[k] = v;
            if(k == n) solution();
            else Try(k+1);
        }
    }
}
int main(){
    n = 3;
    Try(1);
}
```

Backtracking

- Generate all binary sequences of length n containing no 2 consecutive bits 1

```
#include <stdio.h>
#define N 100
int n;
int x[N];
void solution(){
    for(int i = 1; i <= n; i++)
       printf("%d ",x[i]);
    printf("\n");
}
int check(int v, int k){
    if (k == 1) return 1;
    return x[k-1] + v <= 1;
}
```

```
void Try(int k){
    for(int v = 0; v <= 1; v++){
        if(check(v,k)){
            x[k] = v;
            if(k == n) solution();
            else Try(k+1);
        }
    }
}
int main(){
    n = 3;
    Try(1);
}
```

Backtracking

- Generate all permutations of length n

Backtracking

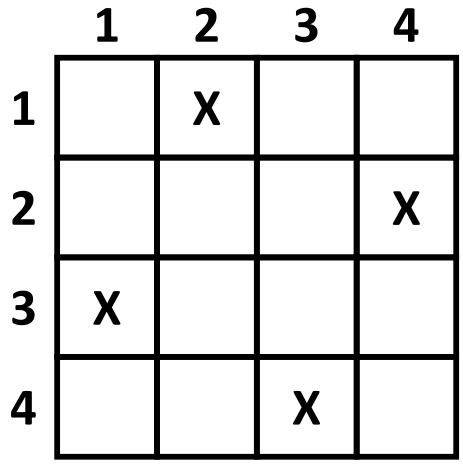
- Generate all permutations of length n

```
#include <stdio.h>
#define N 10
int n;
int x[N];
int mark[N];
void solution(){
    for(int i = 1; i <= n; i++)
        printf("%d ",x[i]);
    printf("\n");
}
```

```
void Try(int k){
    for(int v = 1; v <= n; v++){
        if(!mark[v]){// v has not been used
            x[k] = v;
            mark[v] = 1;// update mark
            if(k == n) solution();
            else Try(k+1);
            mark[v] = 0 ;// recover
        }
    }
}
int main(){
    n = 3; memset(mark,0,sizeof(mark));
    Try(1);
}
```

Backtracking: N-queen

- Place n queens on a chess board such that no two queens attack to each other
- Modelling
  - $x[1, \ldots, n]$ where $x[i]$ is the row of the queen on column i, for $i = 1, \ldots, n$
  - Constraint P
    - $x[i] \neq x[j],\ 1 \leq i < j \leq n$
    - $x[i] + i \neq x[j] + j,\ 1 \leq i < j \leq n$
    - $x[i] - i \neq x[j] - j,\ 1 \leq i < j \leq n$



Backtracking: N-queen

```
int check(int v, int k) {
  // kiểm tra xem v có thể gán được
  // cho x[k] không
  for(int i = 1; i <= k-1; i++) {
    if(x[i] == v) return 0;
    if(x[i] + i == v + k) return 0;
    if(x[i] - i == v - k) return 0;
  }
  return 1;
}
```

```
void TRY(int k) {
  for(int v = 1; v <= n; v++) {
    if(check(v,k)) {
      x[k] = v;
      if(k == n) printSolution();
      else TRY(k+1);
    }
  }
}
void main() {
    TRY(1);
}
```

Backtracking

- Generate all solutions to the positive integer linear equation: $x_1 + x_2 + \ldots + x_n = M$

Backtracking

- Generate all solutions to the positive integer linear equation:

$$x_1 + x_2 + \ldots + x_n = M$$

- Maintain T which is the sum of values of instantiated variables
- Function TRY(k)
  - Variables $x_1, x_2, \ldots, x_{k-1}$ are instantiated (has values)
  - $T = x_1 + x_2 + \ldots + x_{k-1}$
  - $x_{k+1} + x_{k+2} + \ldots + x_n \geq n\text{-}k$

  → $1 \leq x_k \leq M - T - (n - k)$

Backtracking

- Generate all solutions to the positive integer linear equation: $x_1 + x_2 + \ldots + x_n = M$

```
#include <stdio.h>
#define N 100
int n,M,T;
int x[N];
void solution(){
    for(int i = 1; i <= n; i++)
        printf("%d ",x[i]);
    printf("\n");
}
int check(int v, int k){
    if(k == n) return T + v == M;
    return 1;
}
```

```
void Try(int k){
    for(int v = 1; v <= M - T - (n-k); v++){
        if(check(v,k)){
            x[k] = v;
            T += v;
            if(k == n) solution();
            else Try(k+1);
            T -= v;
        }
    }
}
int main(){
    n = 3; M = 5; T = 0;
    Try(1);
}
```

ĐẠI HỌC BÁCH KHOA

25 YEARS ANNIVERSARY SOICT

VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGY

Thank you for your attentions!

soict.hust.edu.vn/ fb.com/groups/soict