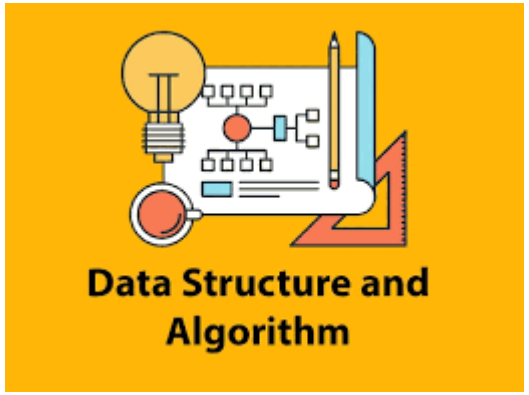
CẤU TRÚC DỮ LIỆU & GIẢI THUẬT

NGÀY 3: ĐỆ QUY & QUAY LUI



Giảng viên: Th.S Bùi Văn Kiên

Nội dung

- Khái niệm đệ quy
- Giới thiệu về stack
- Thuật toán quay lui
- Các bài toán ví dụ:
  - Sinh xâu nhị phân
  - Sinh hoán vị
  - Sinh tổ hợp
  - Phân tích số
  - Xếp quân hậu

1. Khái niệm đệ quy

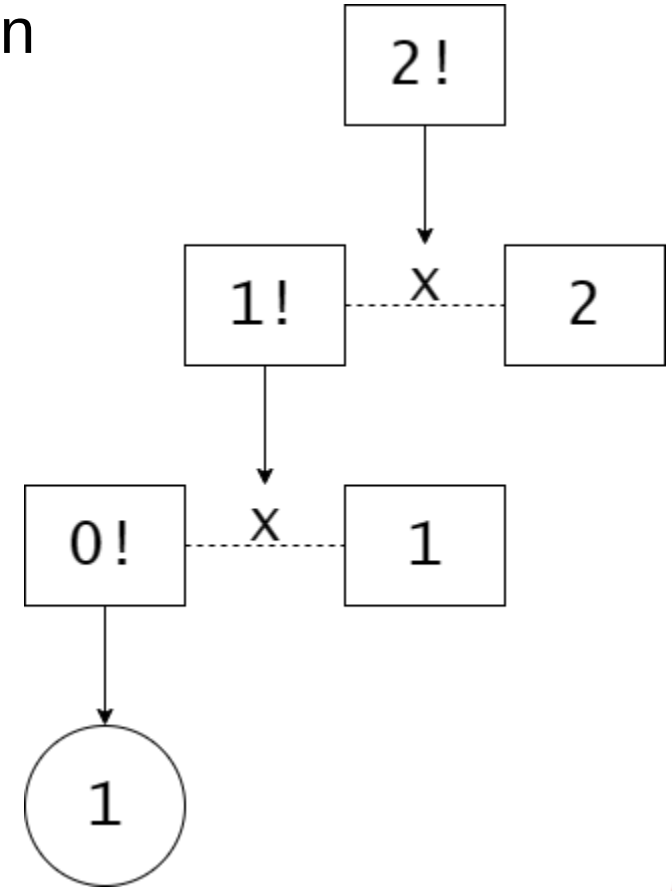
- Ta gọi một đối tượng là đệ quy (recursion) nếu nó được định nghĩa qua chính nó hoặc một đối tượng cùng dạng với chính nó bằng quy nạp
- Ví dụ:

```
// Đây là một hàm gọi đệ quy
void f() {
    f();
}
```

1. Khái niệm đệ quy

Ví dụ 1: hàm tính giai thừa n!

- Công thức đệ quy: n! = (n-1)! * n
- Hay là f(n) = f(n-1)*n



1. Khái niệm đệ quy

- Ví dụ 1: hàm tính giai thừa

```
void factorial(int n) {
    if (n == 0) return 1;              //trường hợp cơ sở
    return factorial(n - 1) * n;       //phần đệ quy
}
```

Tương đương với

```
//n = 0
void factorial_0() {
    return 1;
}

//n = 1
void factorial_1() {
    return factorial_0() * 1;
}

//n = 2
void factorial_2() {
    return factorial_1() * 2;
}
```

1. Khái niệm đệ quy

- Ví dụ 2: Dãy số Fibonacci

$$f_n = \begin{cases} 0 & \text{với } n = 0 \\ 1 & \text{với } n = 1 \\ f_{n-2} + f_{n-1} & \text{với } n > 1 \end{cases}$$

```
void fibo(int n) {
    if (n == 0) return 0;                  //trường hợp cơ sở
    if (n == 1) return 1;                  //trường hợp cơ sở
    return fibo(n - 2) + fibo(n - 1);      //phần đệ quy
}
```

1. Khái niệm đệ quy

- Cấu trúc tổng quát

```
If (suy biến) {
        <Giải quyết trường hợp suy biến>;
}
Else {
        <Tiền xử lý đệ qui>;
        <Gọi đệ qui>;
        <Xử lý hậu đệ qui>;
}
```

1. Khái niệm đệ quy

- Ưu điểm:
  - Sáng sủa, dễ hiểu, nêu rõ bản chất vấn đề
  - Tiết kiệm thời gian hiện thực mã nguồn
- Nhược điểm:
  - Tốn nhiều bộ nhớ, thời gian thực thi lâu
  - Một số bài toán không có lời giải đệ quy

1.2 Cấu trúc các vùng nhớ

Các vùng nhớ trong 1 chương trình C++

- Code segment: chứa mã máy của chương trình
- Data segment: chứa các biến global hoặc static
- Stack segment: Cơ chế First In – Last Out (FILO)

Stack là một phân khúc lưu trữ những biến tạm thời được CPU quản lý và tối ưu khá chặt chẽ. Khi một biến được khai báo trong hàm, nó sẽ được push vào Stack, khi hàm đó kết thúc, toàn bộ những biến đã được đẩy vào trong stack sẽ được pop và giải phóng. Lưu trên stack→ tính chất local.

Khi sử dụng, không cần quan tâm đến việc cấp phát và thu hồi biến đó.

1.2 Cấu trúc các vùng nhớ

Các vùng nhớ trong 1 chương trình C++

- Heap Segment:

Heap là phân khúc bộ nhớ cấp phát tự do. Tuy nhiên nó lại không được quản lý tự động, phải tự quản lý toàn bộ những vùng nhớ đã cấp phát trên Heap, nếu không còn sử dụng nữa mà không tự thu hồi sẽ gây ra hiện tượng rò rỉ bộ nhớ – memory leak.

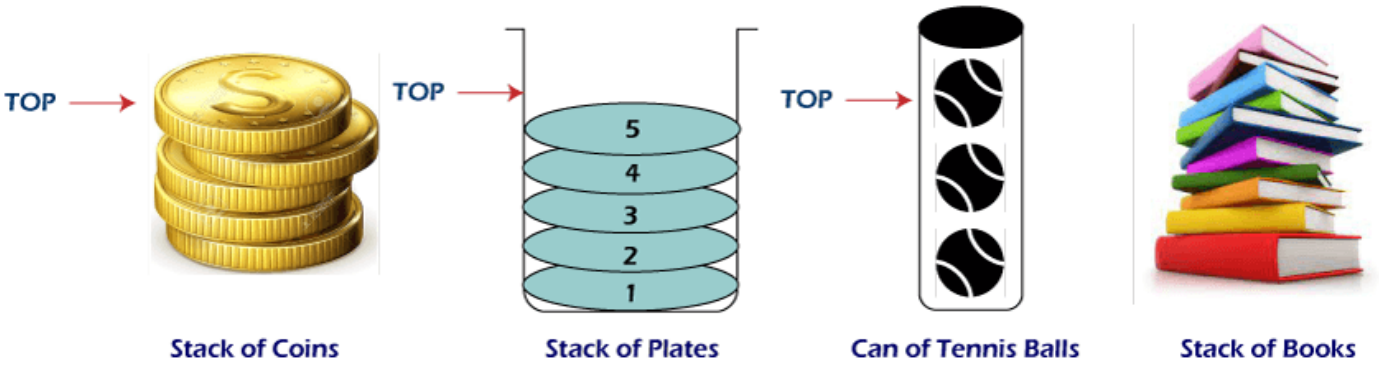
Để cấp phát vùng nhớ trên Heap, có thể dùng malloc() – tức là memory allocation hoặc new. Để thu hồi vùng nhớ bạn có thể dùng free() – tức là memory deallocation hoặc delete.

Truy cập ở phân khúc Stack sẽ nhanh hơn một chút so với Heap nhưng bù lại phân khúc Stack này lại có kích thước có giới hạn tùy vào cấu hình với trình biên dịch.

1.3 Giới thiệu về stack

- Ngăn xếp là một danh sách đặc biệt, trong đó các phép toán chỉ được thực hiện ở một đầu của danh sách (**phần tử top**).

- Tính chất: vào trước ra sau (FILO – First in Last Out)



1.3 Giới thiệu về stack

Trừu tượng hóa cấu trúc ngăn xếp

- Đặc tả dữ liệu

A = (a_0, a_1, …, a_{n-1}) trong đó a_{n-1} là đỉnh ngăn xếp

- Các thao tác:
  - Kiểm tra ngăn xếp có rỗng hay không: isEmpty()
  - Kiểm tra ngăn xếp có đầy hay không: isFull()
  - Trả về phần tử ở đỉnh ngăn xếp: top()
  - Thêm phần tử x vào đỉnh ngăn xếp: push(x)
  - Loại phần tử ở đỉnh ngăn xếp: pop()
  - Đếm số phần tử của ngăn xếp: size()

1.3 Giới thiệu về stack

- #include <stack>

```
#include <iostream>
#include <stack>
using namespace std;

int main() {
    stack<int> S;
    S.push(1);
    S.push(2);
    S.push(3);

    while(!S.empty()) {
        cout << S.top() << endl;
        S.pop();
    }
    return 0;
}
```

Stack
Head = Empty
1 (Head)
1 2 (Head)
1 2 3 (Head)

→ 1 2 (Head)
→ 1 (Head)

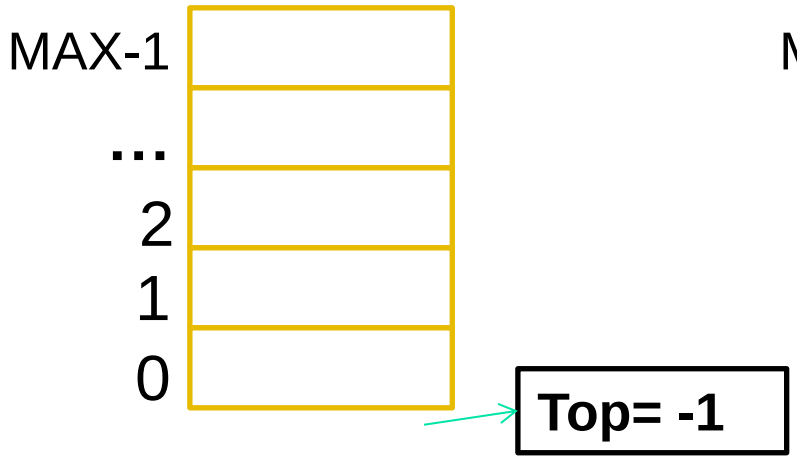
Output = 3 2 1

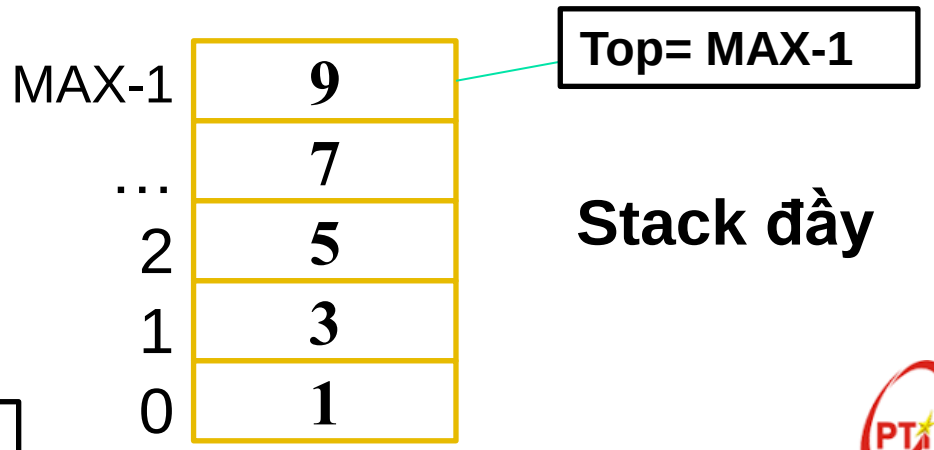
1.4 Biểu diễn stack bằng mảng

- Khai báo struct:

```
#define MAX 1000

struct Stack {
    int top;
    int a[MAX];

    Stack() { top = -1; }
    bool push(int x);
    int pop();
    int topElement();
    bool isEmpty();
};
```

Stack rỗng





Stack đầy

1.4 Biểu diễn stack bằng mảng

- Thao tác 1: Kiểm tra tính rỗng của stack

```
bool Stack::isEmpty() {
    if (top < 0) return 1;
    return 0;
}
```

- Thao tác 2: Kiểm tra tính đầy của stack

```
bool Stack::isFull() {
    if (top == MAXX-1) return 1;
    return 0;
}
```

1.4 Biểu diễn stack bằng mảng

- Thao tác 3: Trả về phần tử đầu tiên của danh sách

```
int Stack::topElement() {
    if (top < 0) {
        cout << "Stack is Empty\n";
        return 0;
    }
    else {
        int x = a[top];
        return x;
    }
}
```

1.4 Biểu diễn stack bằng mảng

- Thao tác 4: Đưa dữ liệu vào danh sách

Chỉ được thực hiện khi và chỉ khi ngăn xếp chưa tràn.

```
bool Stack::push(int x) {
    if (top >= (MAX - 1)) {
        cout << "Stack Overflow\n";
        return false;
    }
    else {
        a[++top] = x;
        cout << x << " pushed into stack\n";
        return true;
    }
}
```

a[++top] tương đương với
top = top+1
a[top] = x

1.4 Biểu diễn stack bằng mảng

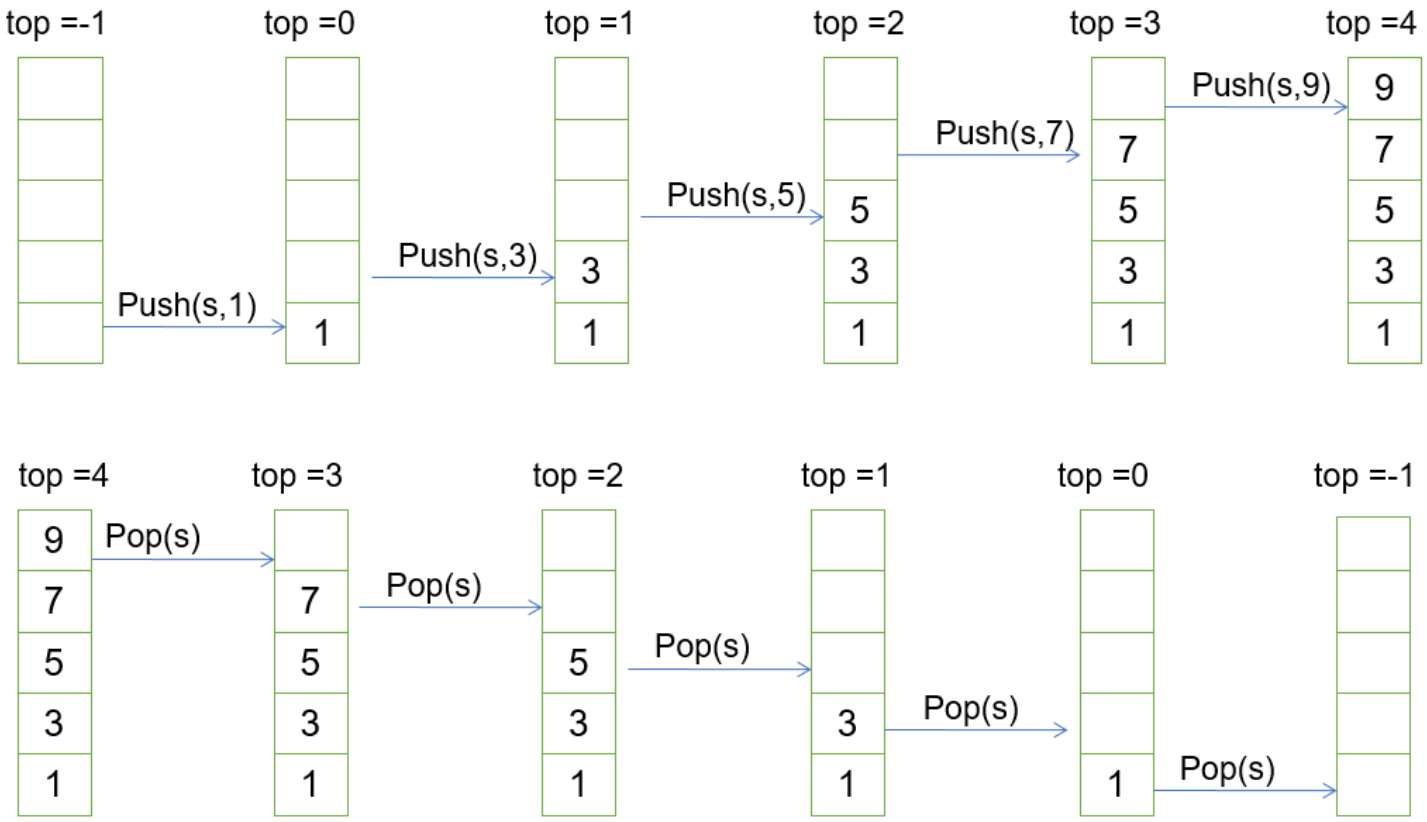
- Thao tác 5: Lấy dữ liệu ra khỏi danh sách

Chỉ được thực hiện khi và chỉ khi ngăn xếp chưa tràn.

```
int Stack::pop() {
    if (top < 0) {
        cout << "Stack Underflow\n";
        return 0;
    }
    else {
        int x = a[top--];
        return x;
    }
}
```

1.4 Biểu diễn stack bằng mảng

- Ví dụ: stack có giới hạn 5 phần tử



1.5 Ví dụ về Call stack

- Hàm tính giai thừa

```cpp
int factorial(int n) {
    if (n == 0) return 1;  // Điều kiện dừng
    return n * factorial(n - 1);
}

int main() {
    int result = factorial(5);
    cout << "Factorial: " << result << endl;
    return 0;
}
```

Stack Frame	Lệnh đang thực thi
`factorial(5)`	`return 5 * factorial(4)`
`factorial(4)`	`return 4 * factorial(3)`
`factorial(3)`	`return 3 * factorial(2)`
`factorial(2)`	`return 2 * factorial(1)`
`factorial(1)`	`return 1 * factorial(0)`
`factorial(0)`	**Base case: return 1**

2. Thuật toán quay lui

- Quay lui (backtracking) là một kĩ thuật thiết kế giải thuật dựa trên **đệ quy**, dùng để giải bài toán liệt kê các cấu hình. Ý tưởng của quay lui là tìm lời giải từng bước, mỗi bước chọn một trong số các lựa chọn có thể và gọi đệ quy cho bước tiếp theo.
- Nói cách khác, chúng ta đang xây dựng một danh sách gồm tất cả các tập hợp (hay dãy, ...), mà mỗi phần tử được xét tất cả các trường hợp có thể của nó. Phương pháp này cũng gọi là **duyệt vét cạn**.

2. Thuật toán quay lui

- Bài toán:
  - Cần xác định bộ X =(x_1, x_2,..,x_n) thỏa mãn ràng buộc.
  - Mỗi thành phần x_i ta có n_i khả năng cần lựa chọn.

```
void backtrack(int pos){
    // Trường hợp cơ sở
    if (<pos là vị trí cuối cùng>){
        <output/lưu lại tập hợp đã dựng nếu thoả mãn>
        return;
    }

    //Phần đệ quy
    for (<tất cả giá trị i có thể ở vị trí pos>){
        <thêm giá trị i vào tập đanh xét>
        backtrack(pos + 1);
        <xoá bỏ giá trị i khỏi tập đang xét>
    }
}
```

3. Bài toán ví dụ

(1) Sinh xâu nhị phân có N kí tự (N <= 20)

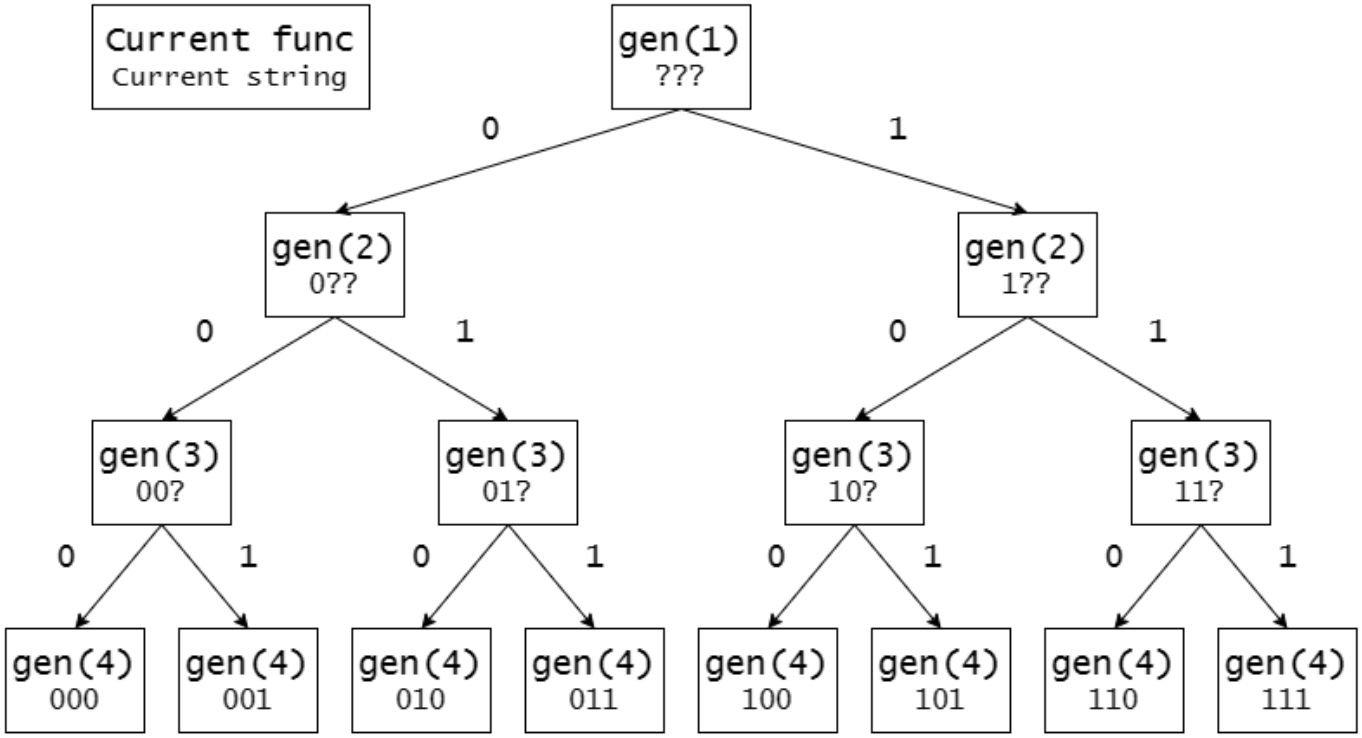
- Với N = 3, có tất cả 8 xâu:
  - 000
  - 001
  - 010
  - 011
  - 100
  - 101
  - 110
  - 111

```
#include <iostream>
using namespace std;

int n;
char ans[21];

void backtrack(int pos){
    if (pos > n){
        for(int i = 1; i <= n; i++)
            cout << ans[i];
        cout << endl;
        return;
    }
    for (char i = '0'; i <= '1'; i ++){
        ans[pos] = i;
        backtrack(pos + 1);
    }
}

int main(){
    cin >> n;
    backtrack(1);

    return 0;
}
```

3. Bài toán ví dụ

(1) Sinh xâu nhị phân có N kí tự (N <= 20)

- Với N = 3, có tất cả 8 xâu:



3. Bài toán ví dụ

(2) Sinh hoán vị

Cho S = {1, 2, …, N}.

Hãy sinh ra hoán vị của tập S.

Với N = 3, có 6 hoán vị:

- (1, 2, 3)
- (1, 3, 2)
- (2, 1, 3)
- (2, 3, 1)
- (3, 1, 2)
- (3, 2, 1)

```
#include <iostream>
using namespace std;

int n;
int a[21], visited[21];

void backtrack(int pos){
    if (pos > n){
        for(int i = 1; i <= n; i++)
            cout << a[i] << " ";
        cout << endl;
        return;
    }
    for (int i = 1; i <= n; i++){
        if(!visited[i]) {
            a[pos] = i;
            visited[i] = 1;
            backtrack(pos + 1);
            visited[i] = 0;
        }
    }
}

int main(){
    memset(visited, 0, sizeof(visited));
    cin >> n;
    backtrack(1);

    return 0;
}
```

3. Bài toán ví dụ

(3) Sinh tổ hợp

Cho S = {1, 2, …, N}. Hãy sinh ra các cấu hình có K phần tử riêng biệt.

Với N = 4, K = 2, có 6 tổ hợp:

- (1, 2)
- (1, 3)
- (1, 4)
- (2, 3)
- (2, 4)
- (3, 4)

```
#include <iostream>
using namespace std;

int n, k;
int a[21];

void backtrack(int pos){
    if (pos > k){
        for(int i = 1; i <= k; i++)
            cout << a[i] << " ";
        cout << endl;
        return;
    }
    for (int i = a[pos-1]+1; i <= n+pos-k; i++){
        a[pos] = i;
        backtrack(pos + 1);
    }
}

int main(){
    cin >> n >> k;
    a[0] = 0;
    backtrack(1);

    return 0;
}
```

QUESTIONS & ANSWERS