

### NỘI DUNG:

- 2.1. Thuật toán duyệt
- 2.2. Thuật toán sinh
- 2.3. Thuật toán đệ qui
- 2.4. Thuật toán quay lui
- 2.5. Thuật toán tham lam
- 2.6. Thuật toán chi để trị
- 2.7. Thuật toán qui hoạch động
- 2.8. Thuật toán tìm kiếm mẫu
- 2.9. Thuật toán nhánh cận
- 2.9. CASE STUDY:

## 2.1. Giới thiệu thuật toán

### Phương pháp giải quyết bài toán (Problem):

#### Sử dụng các công cụ toán học:

- Sử dụng các định lý, mệnh đề, lập luận và suy logic của trong toán học để tìm ra nghiệm của bài toán.
- Ưu điểm: dễ dàng máy tính hóa các bài toán đã có thuật giải (MathLab).
- Nhược điểm: chỉ thực hiện được trên lớp các bài toán đã có thuật giải. Lớp bài toán này rất nhỏ so với lớp các bài toán thực tế.

#### Sử dụng máy tính và các công cụ tính toán:

- Giải được mọi bài toán đã có thuật giải bằng máy tính.
- Đối với một số bài toán chưa có thuật giải, ta có thể sử dụng máy tính để xem xét tất cả các khả năng có thể để từ đó đưa ra nghiệm của bài toán. Một thuật toán duyệt cần thỏa mãn hai điều kiện:
  - *Không được lặp lại bất kỳ khả năng nào.*
  - *Không được bỏ sót bất kỳ cấu hình nào.*

**Ví dụ 1.** Cho hình vuông gồm 25 hình vuông đơn vị. Hãy điền các số từ 0 đến 9 vào mỗi hình vuông đơn vị sao cho những điều kiện sau được thỏa mãn.

- Đọc từ trái sang phải theo hàng ta nhận được 5 số nguyên tố có 5 chữ số;
- Đọc từ trên xuống dưới theo cột ta nhận được 5 số nguyên tố có 5 chữ số;
- Đọc theo hai đường chéo chính ta nhận được 2 số nguyên tố có 5 chữ số;
- Tổng các chữ số trong mỗi số nguyên tố đều là S cho trước.

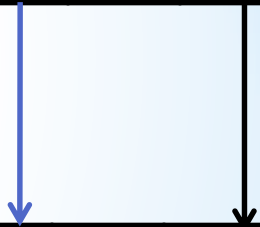
Ví dụ hình vuông dưới đây với  $S = 11$ .

3	5	1	1	1
5	0	0	3	3
1	0	3	4	3
1	3	4	2	1
1	3	3	1	3

**Thuật giải duyệt.** Chia bài toán thành hai bài toán con như sau:

- Tìm  $X = \{x \in [10001, \dots, 99999] \mid x \text{ là nguyên tố và tổng các chữ số là } S\}$ .
- Chiến lược vét cạn được thực hiện như sau:
  - Lấy  $x \in X$  đặt vào hàng 1 (H1): ta điền được ô vuông 1, 2, 3, 4, 5.
  - Lấy  $x \in X$  có số đầu tiên trùng với ô số 1 đặt vào cột 1 (C1): ta điền được ô vuông 6, 7, 8, 9.
  - Lấy  $x \in X$  có số đầu tiên trùng với ô số 9, số cuối cùng trùng với ô số 5 đặt vào đường chéo chính 2 (D2): ta điền được ô vuông 10, 11, 12.
  - Lấy  $x \in X$  có số thứ nhất và số thứ 4 trùng với ô số 6 và 12 đặt vào hàng 2 (H2): ta điền được ô vuông 13, 14, 15.
  - Lấy  $x \in X$  có số thứ nhất, thứ hai, thứ 4 trùng với ô số 2, 13, 10 đặt vào cột 2 (C2): ta điền được ô vuông 16, 17.
  - Làm tương tự như vậy ta điền vào hàng 5 ô số 25.
  - Cuối cùng ta chỉ cần kiểm tra  $D1 \in X$  và  $C5 \in X$ ?

3	5	1	1	1
5	0	0	3	3
1	0	3	4	3
1	3	4	2	1
1	3	3	1	3



1	2	3	4	5
6	13	14	12	15
7	16	11	18	19
8	10	20	22	23
9	17	21	24	25

## 2.2. Thuật toán sinh (Generation)

Thuật toán sinh được dùng để giải lớp các bài toán thỏa mãn hai điều kiện:

- *Xác định được một thứ tự trên tập các cấu hình cần liệt kê của bài toán. Biết được cấu hình đầu tiên, biết được cấu hình cuối cùng.*
- *Từ một cấu hình cuối cùng, ta xây dựng được thuật toán sinh ra cấu hình đứng ngay sau nó theo thứ tự.*

**Thuật toán:**

**Thuật toán Generation:**

**Bước1 (Khởi tạo):**

*<Thiết lập cấu hình đầu tiên>;*

**Bước 2 (Bước lặp):**

**while** (*<Lặp khi cấu hình chưa phải cuối cùng>*) **do**

*<Đưa ra cấu hình hiện tại>;*

*<Sinh ra cấu hình kế tiếp>;*

**endwhile;**

**End.**

**Ví dụ 1.** Duyệt các xâu nhị phân có độ dài  $n$ .

**Lời giải.** Xâu  $X = (x_1, x_2, \dots, x_n) : x_i = 0, 1; i=1, 2, \dots, n$  được gọi là xâu nhị phân có độ dài  $n$ . Ví dụ với  $n=4$ , ta có 16 xâu nhị phân dưới đây:

$STT$	$X = (x_1, \dots, x_n)$	$F(X)$	$STT$	$X = (x_1, \dots, x_n)$	$F(X)$
1	0000	0	9	1000	8
2	0001	1	10	1001	9
3	0010	2	11	1010	10
4	0011	3	12	1011	11
5	0100	4	13	1100	12
6	0101	5	14	1101	13
7	0110	6	15	1110	14
8	0111	7	16	1111	15

## Thuật toán sinh xâu nhị phân kế tiếp;

Input:

- +  $X = (x_1, x_2, \dots, x_n)$  là biến toàn cục.
- +  $OK = 1$  là biến toàn cục

Output:  $2^n$  xâu nhị phân có độ dài  $n$ .

```
Void Next_Bit_String(void) {  
    int i=n;  
    while (i>0 && X[i]!=0) { X[i] = 0; i--; }  
    if (i >0) X[i]=1;  
    else OK = 0;  
}
```

Actions:

```
X =(0,0,...,0); // Xâu nhị phân ban đầu.  
while (OK) { //Lặp khi xâu chưa phải cuối cùng  
    Result(); //Đưa ra xâu hiện tại>;  
    Next_Bit_String(); //Sinh ra xâu kế tiếp  
}
```

Endactions



## Bài tập. Hãy cho biết kết quả thực hiện chương trình dưới đây?

```
#include <iostream.h>
#include <stdlib.h>
#define MAX 100
#define TRUE 1
#define FALSE 0
int n, X[MAX], OK=TRUE, dem=0;
void Init (void){
    cout<<"\n Nhap n=";cin>>n;
    for (int i=1; i<=n; i++)
        X[i] =0;
}
void Result(void){
    cout<<"\n Ket qua "<<++dem<<":";
    for (int i=1; i<=n; i++)
        cout<<X[i]<<"  ";
}
```

```
void Next_Bit_String(void) {
    int i= n;
    while (i>0 && X[i]!=0){
        X[i] = 0; i --;
    }
    if (i > 0 ) X[i] = 1;
    else OK = FALSE;
}
int main() {
    Init(); //Nhap n = 4
    while (OK ){
        Result();
        Next_Bit_String();
    }
    system("PAUSE");
    return 0;
}
```



**Ví dụ 2.** Duyệt các tổ hợp chập  $K$  của  $1, 2, \dots, N$ .

**Lời giải.** Mỗi tổ hợp chập  $K$  của  $1, 2, \dots, N$  là một tập con  $K$  phần tử khác nhau của  $1, 2, \dots, N$ . Ví dụ với  $N=5, K=3$  ta sẽ có  $C(N,K)$  tập con dưới đây

STT	Tập con $X = (x_1, \dots, x_k)$
1	1 2 3
2	1 2 4
3	1 2 5
4	1 3 4
5	1 3 5
6	1 4 5
7	2 3 4
8	2 3 5
9	2 4 5
10	3 4 5

**Thứ tự tự nhiên.** Duyệt các tổ hợp chập  $K$  của  $1, 2, \dots, N$ .

Có thể xác định được nhiều trật tự khác nhau trên các tổ hợp. Tuy nhiên, thứ tự đơn giản nhất có thể được xác định như sau:

Ta gọi tập con  $X = (x_1, \dots, x_K)$  là đứng trước tập con  $Y = (y_1, y_2, \dots, y_K)$  nếu tìm được chỉ số  $t$  sao cho  $x_1 = y_1, x_2 = y_2, \dots, x_{t-1} = y_{t-1}, x_t < y_t$ . Ví dụ tập con  $X = (1, 2, 3)$  đứng trước tập con  $Y = (1, 2, 4)$  vì với  $t=3$  thì  $x_1 = y_1, x_2 = y_2, x_3 < y_3$ .

Tập con đầu tiên là  $X = (1, 2, \dots, K)$ , tập con cuối cùng là  $(N-K+1, \dots, N)$ . Như vậy điều kiện 1 của thuật toán sinh được thỏa mãn.

Thuật toán sinh tổ hợp:

```
Void Next_Combination(void) {  
    int i = k; // Xuất phát từ phần tử cuối cùng của tổ hợp  
    while ( i>0 && X[i] == N - K + i) i --; //Tìm phần tử  $X[i] \neq N-K+i$   
    if (i>0) { //Nếu i chưa vượt quá phần tử cuối cùng  
        X[i] = X[i] + 1; //Thay  $X[i] = X[i] + 1$   
        for (int j = i+1; j<=k; j++) //Từ phần tử thứ j +1 đến k  
            X[j] = X[i] + j - i; // Thay thế  $X[j] = X[i] + j - i$   
    }  
    else OK = 0; //OK =0 nếu đã đến tập con cuối cùng  
}
```

## Bài tập. Hãy cho biết kết quả thực hiện chương trình dưới đây?

```
#include <iostream.h>
#include <stdlib.h>
#define MAX 100
#define TRUE 1
#define FALSE 0
int n,k,X[MAX],OK=TRUE,dem=0;
void Init (void ){
    cout<<"\n Nhap n=";cin>>n;
    cout<<"\n Nhap k=";cin>>k;
    for (int i=1; i<=k; i++)
        X[i] =i;
}
void Result(void){
    cout<<"\n Ket qua buoc "<<++dem<<":";
    for (int i=1; i<=k; i++)
        cout<<X[i]<<"  ";
}
```

```
void Next_Combination(void) {
    int i= k;
    while (i>0 && X[i]==n-k+i)i--;
    if (i > 0 ) {
        X[i] = X[i] +1;
        for (int j = i+1; j<=k; j++)
            X[j] = X[i] + j - i;
    }
    else OK = FALSE;
}
int main()
{
    Init(); //Nhap n = 5, k = 3
    while (OK ){
        Result();
        Next_Combination();
    }
    system("PAUSE");
    return 0;
}
```

**Ví dụ 3.** Duyệt các hoán vị của 1, 2,..., N.

**Lời giải.** Mỗi hoán vị của 1, 2, ..., N là một cách xếp có tính đến thứ tự của 1, 2,...,N. Số các hoán vị là  $N!$ . Ví dụ với  $N = 3$  ta có 6 hoán vị dưới đây.

**Thứ tự tự nhiên.** Có thể xác định được nhiều trật tự khác nhau trên các hoán vị. Tuy nhiên, thứ tự đơn giản nhất có thể được xác định như sau. Hoán vị  $X = (x_1, x_2, \dots, x_n)$  được gọi là đứng sau hoán vị  $Y = (y_1, y_2, \dots, y_n)$  nếu tồn tại chỉ số  $k$  sao cho

$x_1 = y_1, x_2 = y_2, \dots, x_{k-1} = y_{k-1}, x_k < y_k$ . Ví dụ hoán vị  $X = (1, 2, 3)$  được gọi là đứng sau hoán vị  $Y = (1, 3, 2)$  vì tồn tại  $k = 2$  để  $x_1 = y_1$ , và  $x_2 < y_2$ .

STT	Hoán vị $X = (x_1, \dots, x_N)$
1	1 2 3
2	1 3 2
3	2 1 3
4	2 3 1
5	3 1 2
6	3 2 1

```

Void Next_Permutation(void) {
    int j = N-1; // Xuất phát từ phần tử N-1
    while ( j>0 && X[j]> X[j+1]) j --; //Tìm j sao cho X[j]>X[j+1]
    if (j>0) { //Nếu i chưa vượt quá phần tử cuối cùng
        int k =N; // Xuất phát từ k = N
        while ( X[j] > X[k] ) k --; // Tìm k sao cho X[j] <X[k]
        int t = X[j]; X[j] = X[k]; X[k] = t; //Đổi chỗ X[j] cho X[k]
        int r = j +1, s = N;
        while ( r <=s ) { //Lật ngược đoạn từ j +1 đến N
            t = r; r = s; s=t;
            r ++; s --;
        }
    }
    else OK =0; //Nếu đến hoán vị cuối cùng
}

```

## Bài tập. Hãy cho biết kết quả thực hiện chương trình dưới đây?

```
#include <iostream.h>
#include <stdlib.h>
#define MAX 100
#define TRUE 1
#define FALSE 0
int n,X[MAX],OK=TRUE,dem=0;
void Init (void ){
    cout<<"\n Nhap n=";cin>>n;
    for (int i=1; i<=n; i++)
        X[i] =i;
}
void Result(void){
    cout<<"\n Ket qua buoc " << ++dem << ":";
    for (int i=1; i<=n; i++)
        cout<<X[i]<<"  ";
}
```

```
void Next_Permutation(void) {
    int j= n-1;
    while (j>0 && X[j]>X[j+1]) j--;
    if (j > 0 ) {
        int k =n;
        while(X[j]>X[k]) k--;
        int t = X[j]; X[j]=X[k]; X[k]=t;
        int r = j +1, s =n;
        while (r<=s ) {
            t = X[r]; X[r]=X[s]; X[s] =t;
            r ++; s--;
        }
    }
    else OK = FALSE;
}

void main() {
    Init(); //Nhap n = 4
    while (OK ){ Result();
        Next_Permutation();
    }
}
```

**Ví dụ 4.** Cho số tự nhiên  $N$  ( $N \leq 100$ ). Hãy liệt kê tất cả các cách chia số tự nhiên  $N$  thành tổng của các số tự nhiên nhỏ hơn  $N$ . Các cách chia là hoán vị của nhau chỉ được tính là một cách. Ví dụ với  $N = 5$  ta có 7 cách chia như sau:

5				
4	1			
3	2			
3	1	1		
2	2	1		
2	1	1	1	
1	1	1	1	1



## //Hãy cho biết kết quả thực hiện ?

```
#include <iostream.h>
#include <stdlib.h>
#define MAX 100
#define TRUE 1
#define FALSE 0
int n, k, X[MAX], dem =0, OK =TRUE;
void Init(void ){
    cout<<"\n Nhap n=";  cin>>n;
    k = 1; X[k] = n;
}
void Result(void) {
    cout<<"\n Cach chia "<<++dem<<":";
    for (int i=1; i<=k; i++)
        cout<<X[i]<<"  ";
}
```

```
void Next_Division(void ){
    int i = k, j, R, S,D;
    while (i > 0 && X[i]==1 ) i--;
    if (i>0 ) {
        X[i] = X[i] - 1;  D = k - i + 1;
        R = D / X[i];    S = D % X[i];
        k= i;
        if (R>0) {
            for ( j = i +1; j<=i + R; j++)
                X[j] = X[i];
            k = k + R;
        }
        if (S>0 ){
            k = k +1; X[k] = S;
        }
    }
    else OK =0;
}

int main() {  Init(); //Nhap n = 5.
    while (OK ) {
        Result(); Next_Division();
    }
    return 0;
}
```

**Bài tập:** Sử dụng thuật toán sinh.

1. Cho dãy  $A[]$  gồm  $N$  số tự nhiên khác nhau và số tự nhiên  $K$ . Hãy sử dụng *thuật toán sinh* viết chương trình liệt kê tất cả các dãy con của dãy số  $A[]$  sao cho tổng các phần tử trong dãy con đó đúng bằng  $K$ .

Dayso.in

5 50  
5 10 15 20 25

Ketqua.out

3  
10 15 25  
5 20 25  
5 10 15 20

2. Cho dãy  $A_N = \{a_1, a_2, \dots, a_N\}$  gồm  $N$  số tự nhiên phân biệt. Hãy sử dụng thuật toán *sinh* (*quay lui, nhánh cận, qui hoạch động*) viết chương trình liệt kê tất cả các dãy con  $K$  phần tử của dãy số  $A_N$  ( $K \leq N$ ) sao cho tổng các phần tử của dãy con đó là một số đúng bằng  $B$ .

dayso.in

5 3 50  
5 10 15 20 25

ketgau.out

2  
5 20 25  
10 15 25

**Bài tập:** Sử dụng thuật toán sinh.

3. Hãy sử dụng thuật toán *sinh* (*quay lui*, *nhánh cận*, *qui hoạch động*) viết chương trình Viết chương trình tìm  $X = (x_1, x_2, \dots, x_n)$  và  $f(X)$  đạt giá trị lớn nhất. Trong đó:

$$f(x_1, x_2, \dots, x_n) = \sum_{i=1}^n c_i x_i$$

$$X = (x_1, x_2, \dots, x_n) \in D = \left\{ \sum_{i=1}^n a_i x_i \leq b; x_i \in \{0, 1\} \right\}$$

**Caitui.in**

4	8
5	10
3	5
2	3
4	6

**Ketqua.out**

15			
1	1	0	0

4. Một dãy số tự nhiên bất kỳ  $A_N = \{a_1, a_2, \dots, a_N\}$  được gọi là một *dãy số nguyên tố thuần nhất bậc K* nếu tổng K phần tử liên tiếp bất kỳ của dãy số  $A_N$  là một số nguyên tố ( $K \leq N$ ). Ví dụ dãy số  $A_N = \{3, 27, 7, 9, 15\}$  là một *dãy số nguyên tố thuần nhất bậc 3*. Cho dãy số  $A_N$ . Hãy liệt kê tất cả các *dãy số nguyên tố thuần nhất bậc K* có thể có được tạo ra bằng cách trao đổi các phần tử khác nhau của dãy số  $A_N$ .

Ví dụ.

Input:

- $n = 5, K = 3$
- $A = (3, 7, 9, 15, 27)$

Output:

4				
3	27	7	9	15
15	9	7	3	27
15	9	7	27	3
27	3	7	9	15

## 2.3. Thuật toán đệ qui (Recursion)

**Phương pháp định nghĩa bằng đệ qui:** Một đối tượng được định nghĩa thông qua chính nó được gọi là phép định nghĩa bằng đệ qui.

**Thuật toán đệ qui:** Thuật toán giải bài toán  $P$  thông qua bài toán  $P'$  giống như  $P$  được gọi là thuật toán đệ qui. Một hàm được gọi là đệ qui nếu nó được gọi đến chính nó. Một bài toán giải được bằng đệ qui nếu nó thỏa mãn hai điều kiện:

- **Phân tích được:** Có thể giải được bài toán  $P$  bằng bài toán  $P'$  giống như  $P$  và chỉ khác  $P$  ở dữ liệu đầu vào. Việc giải bài toán  $P'$  cũng được thực hiện theo cách phân tích giống như  $P$ .
- **Điều kiện dừng:** Dãy các bài toán  $P'$  giống như  $P$  là hữu hạn và sẽ dừng tại một bài toán xác định nào đó.

Thuật toán đệ qui tổng quát có thể được mô tả như sau:

*Thuật toán Recursion (  $P$  ) {*

*1. Nếu  $P$  thỏa mãn điều kiện dừng:*

*<Giải  $P$  với điều kiện dừng>;*

*2. Nếu  $P$  không thỏa mãn điều kiện dừng:*

*Recursion( $P'$ ).*

*}*

**Ví dụ:** Tìm tổng của n số tự nhiên bằng phương pháp đệ qui.

**Lời giải.** Gọi  $S_n$  là tổng của n số tự nhiên. Khi đó:

- **Bước phân tích:**  $S_n = n + S(n-1)$ ,  $n > 1$ .
- **Điều kiện dừng:**  $s_1 = 1$  nếu  $n=1$ ;

Từ đó ta có lời giải của bài toán như sau:

```
int      Tong (int i ) {  
    if (i ==1 ) return(1); //Điều kiện dừng  
    else return(i + Tong(i-1)); //Điều kiện phân tích được  
}
```

**Ví dụ.** Tìm  $n!$ .

**Lời giải.** Gọi  $S_n$  là  $n!$ . Khi đó:

- **Bước phân tích:**  $S_n = n*(n-1)!$  nếu  $n > 0$ ;
- **Điều kiện dừng:**  $s_0=1$  nếu  $n=0$ .

Từ đó ta có lời giải của bài toán như sau:

```
long     Giaithua (int i ) {  
    if (i ==0 ) return(1); //Điều kiện dừng  
    else return(i *Giaithua(i-1)); //Điều kiện phân tích được  
}
```

**Ví dụ:** Tìm ước số chung lớn nhất của a và b bằng phương pháp đệ qui.

**Lời giải.** Gọi  $d = \text{USCLN}(a, b)$ . Khi đó:

- **Bước phân tích:**

- $d = \text{USCLN}(a-b, b)$  nếu  $a > b$ .

- $d = \text{USCLN}(a, b-a)$  nếu  $a < b$ .

- **Điều kiện dừng:**  $d = a$  hoặc  $d = b$  nếu  $a = b$ ;

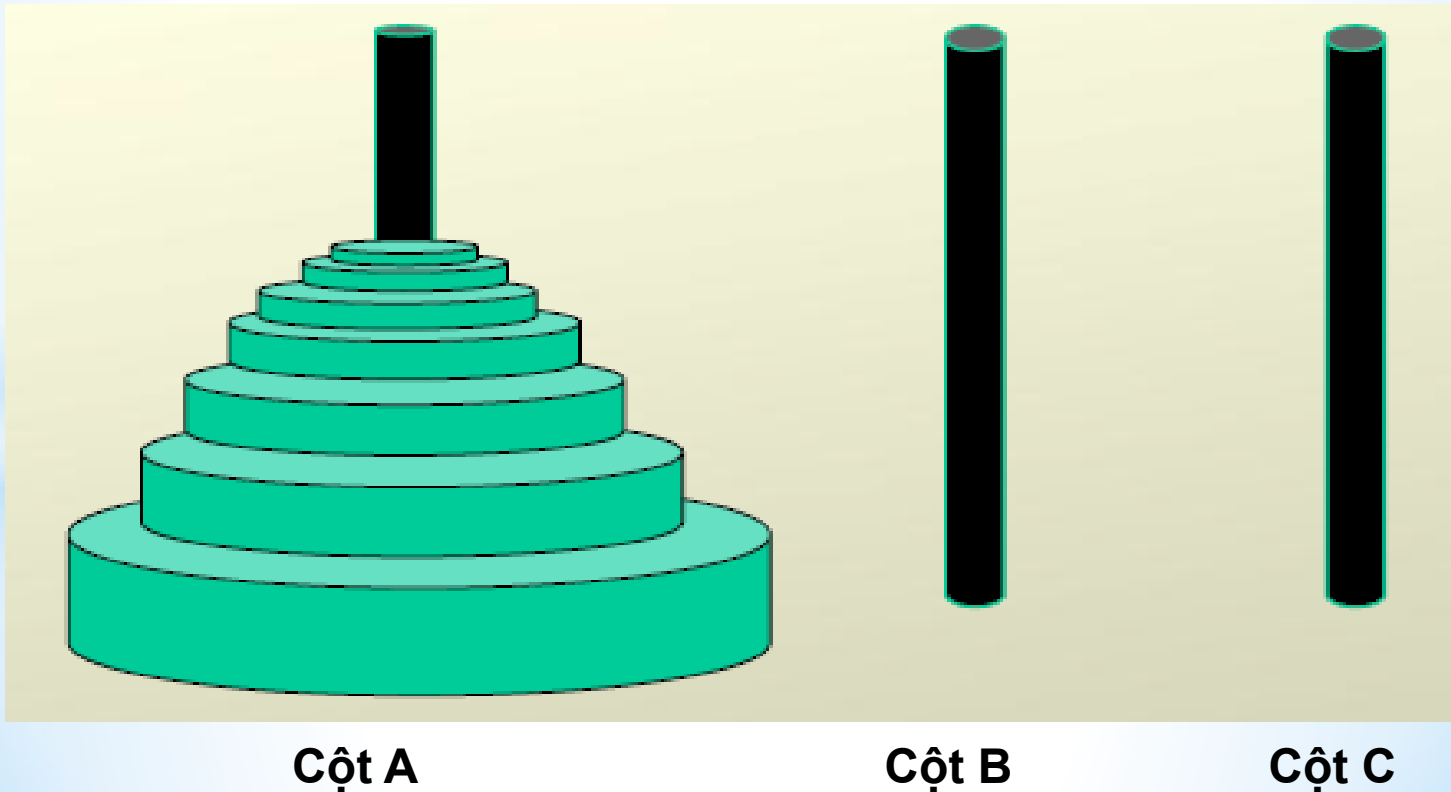
Từ đó ta có lời giải của bài toán như sau:

```
int    USCLN (int a, int b ) {  
    if (a == b ) return(a); //Điều kiện dừng  
    else { //Điều kiện phân tích được  
        if (a > b) return(USCLN(a-b, b));  
        else return(USCLN(a, b-a));  
    }  
}
```



**Ví dụ.** Bài toán tháp Hà Nội. Có  $n$  đĩa bạc có đường kính khác nhau được đặt chồng lên nhau. Các đĩa được đặt chồng lên nhau theo thứ tự giảm dần của đường kính. Có ba vị trí A, B, C có thể đặt đĩa. Chồng đĩa ban đầu đặt ở vị trí A. Một nhà sư chuyển các đĩa từ vị trí A sang vị trí B thỏa mãn điều kiện:

- Khi di chuyển một đĩa phải đặt vào ba vị trí đã cho.
- Mỗi lần chuyển một đĩa và phải là đĩa trên cùng.
- Tại một vị trí đĩa được đặt cũng phải lên trên cùng.
- Đĩa lớn hơn không được phép đặt trên đĩa nhỏ hơn.



## Lời giải.

- Nếu  $n=1$  thì ta chỉ cần dịch chuyển một lần là xong.
- Với  $n>1$ : ta cần dịch chuyển  $n-1$  đĩa còn lại từ cột A sang cột C, sau đó dịch chuyển một đĩa cuối cùng từ cột A sang cột B. Sau đó ta lại phải chuyển  $n-1$  đĩa từ cột C sang cột A lấy cột B làm cột trung gian.

Thuật toán Hanoi-Tower ( $n, A, B, C$ ) {

    If (  $n==1$  )

        <Dịch chuyển đĩa cuối cùng từ A sang C>;

    Else {

        Hanoi-Tower( $n-1, A, C, B$ );

        Hanoi-Tower( $n-1, C, B, A$ );

    }

}

## 2.4. Thuật toán quay lui (Back track)

Giả sử ta cần xác định bộ  $X = (x_1, x_2, \dots, x_n)$  thỏa mãn một số ràng buộc nào đó. Ứng với mỗi thành phần  $x_i$  ta có  $n_i$  khả năng cần lựa chọn. Ứng với mỗi khả năng  $j \in n_i$  dành cho thành phần  $x_i$  ta cần thực hiện:

- Kiểm tra xem khả năng  $j$  có được chấp thuận cho thành phần  $x_i$  hay không? Nếu khả năng  $j$  được chấp thuận thì nếu  $i$  là thành phần cuối cùng ( $i=n$ ) ta ghi nhận nghiệm của bài toán. Nếu  $i$  chưa phải cuối cùng ta xác định thành phần thứ  $i+1$ .
- Nếu không có khả năng  $j$  nào được chấp thuận cho thành phần  $x_i$  thì ta quay lại bước trước đó ( $i-1$ ) để thử lại các khả năng khác.

Thuật toán Back-Track ( int  $i$  ) {

    For (  $j = \text{<Khả năng 1>}; j \leq n_i; j++$  ) {

        if ( <chấp thuận khả năng  $j$ > ) {

$X[i] = \text{<khả năng } j\text{>};$

            if (  $i == n$  ) Result();

            else Back-Track( $i+1$ );

        }

    }

**Ví dụ 1.** Duyệt các xâu nhị phân có độ dài n.

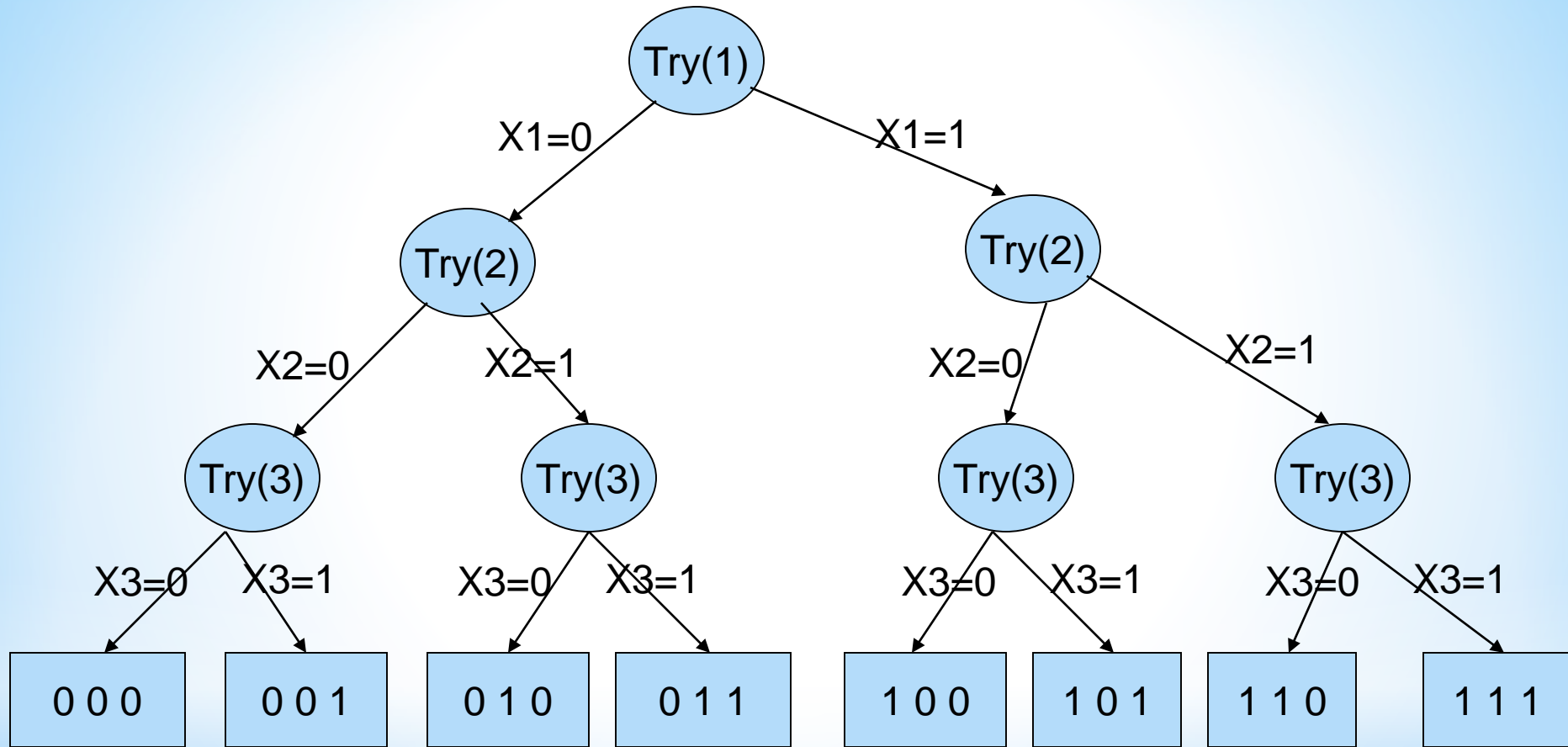
Lời giải. Xâu nhị phân  $X = (x_1, x_2, \dots, x_n) \mid x_i = 0, 1$ . Mỗi  $x_i \in X$  có hai lựa chọn  $x_i = 0, 1$ . Cả hai giá trị này đều được chấp thuận mà không cần có thêm bất kỳ điều kiện gì.

Thuật toán được mô tả như sau:

```
Void Try ( int i ) {  
    for (int j =0; j<=1; j++){  
        X[i] = j;  
        if ( i ==n) Result();  
        else Try (i+1);  
    }  
}
```

Khi đó, việc duyệt các xâu nhị phân có độ dài n ta chỉ cần gọi đến thủ tục Try(1).  
Cây quay lui được mô tả như hình dưới đây.

Cây đệ qui duyệt các xâu nhị phân độ dài  $n = 3$ .



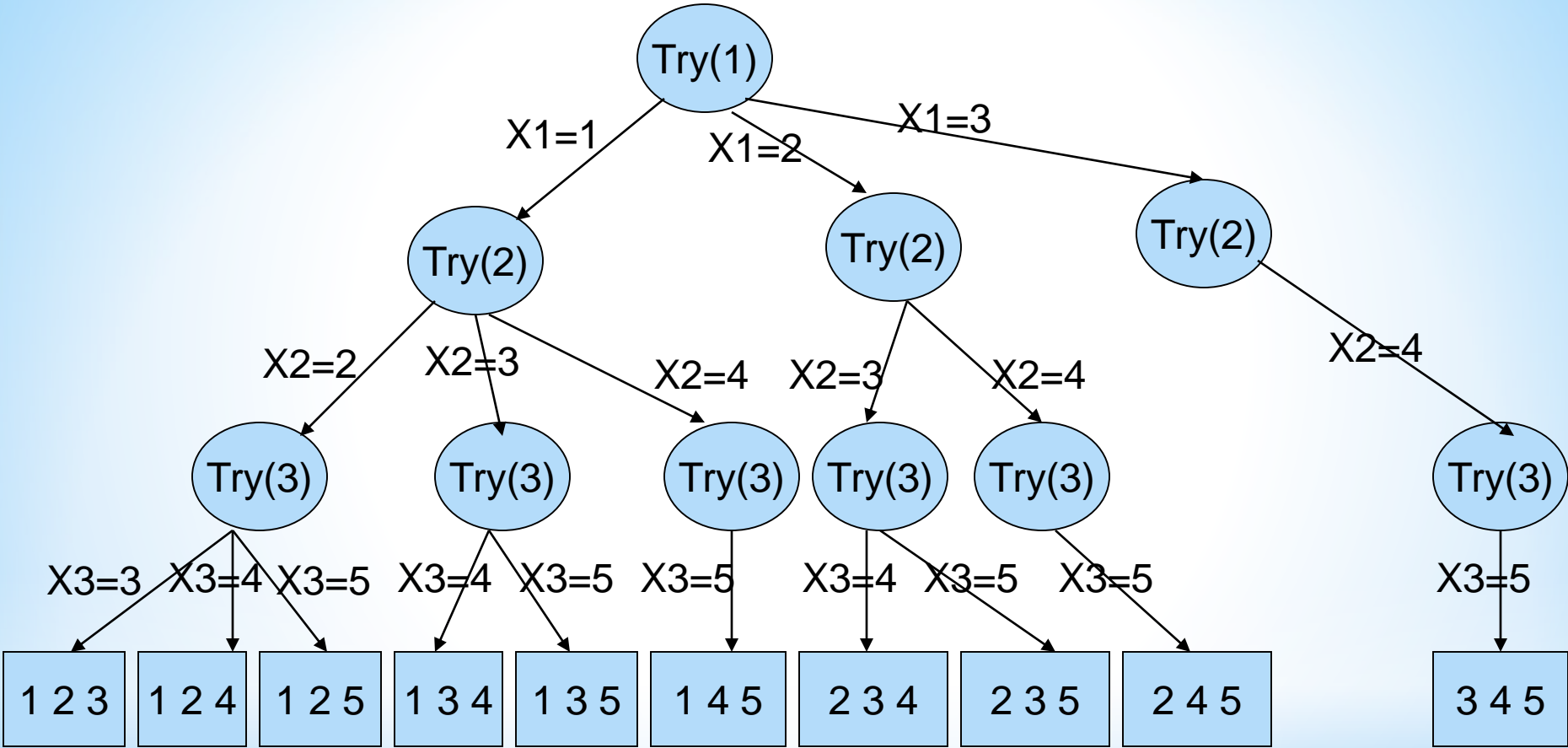
**Ví dụ 2.** Duyệt các tập con K phần tử của 1, 2, ..., N.

**Lời giải.** Mỗi tập con K phần tử  $X = (x_1, x_2, \dots, x_K)$  là bộ không tính đến thứ tự K phần tử của 1, 2, ..., N. Mỗi  $x_i \in X$  có  $N-K+i$  lựa chọn. Các giá trị này đều được chấp thuận mà không cần có thêm bất kỳ điều kiện gì. Thuật toán được mô tả như sau:

```
Void Try ( int i ) {  
    for (int j =X[i-1]+1; j<=N-K+ i; j++){  
        X[i] = j;  
        if ( i ==K) Result();  
        else Try (i+1);  
    }  
}
```

Khi đó, việc duyệt các tập con K phần tử của 1, 2, ..., N ta chỉ cần gọi đến thủ tục Try(1). Cây quay lui được mô tả như hình dưới đây.

Cây đệ qui duyệt các tập con 3 phần tử của 1, 2, ..., 5.





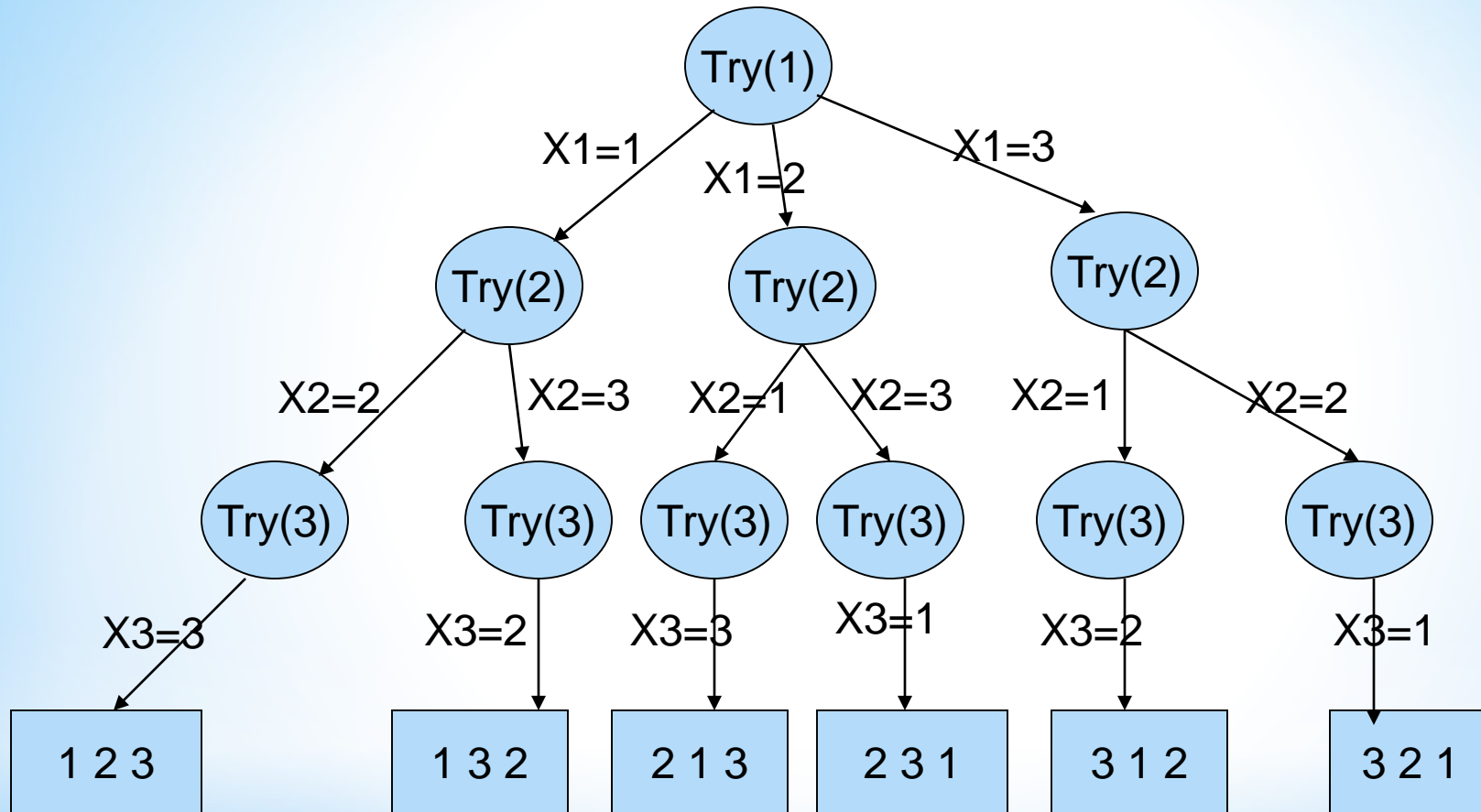
**Ví dụ 3.** Duyệt các hoán vị của 1, 2, ..., N.

**Lời giải.** Mỗi hoán vị  $X = (x_1, x_2, \dots, x_N)$  là bộ có tính đến thứ tự của 1, 2, ..., N. Mỗi  $x_i \in X$  có N lựa chọn. Khi  $x_i = j$  được lựa chọn thì giá trị này sẽ không được chấp thuận cho các thành phần còn lại. Để ghi nhận điều này, ta sử dụng mảng chuaxet[] gồm N phần tử. Nếu chuaxet[i] = True điều đó có nghĩa giá trị i được chấp thuận và chuaxet[i] = False tương ứng với giá trị i không được phép sử dụng. Thuật toán được mô tả như sau:

```
Void Try ( int i ) {  
    for (int j =1; j<=N; j++){  
        if (chuaxet[j] ) {  
            X[i] = j; chuaxet[j] = False;  
            if ( i ==N) Result();  
            else Try (i+1);  
            Chuaxet[j] = True;  
        }  
    }  
}
```

Khi đó, việc duyệt các hoán vị của 1, 2, ..., N ta chỉ cần gọi đến thủ tục Try(1). Cây quay lui được mô tả như hình dưới đây.

Cây đệ qui duyệt các hoán vị của 1, 2, 3.

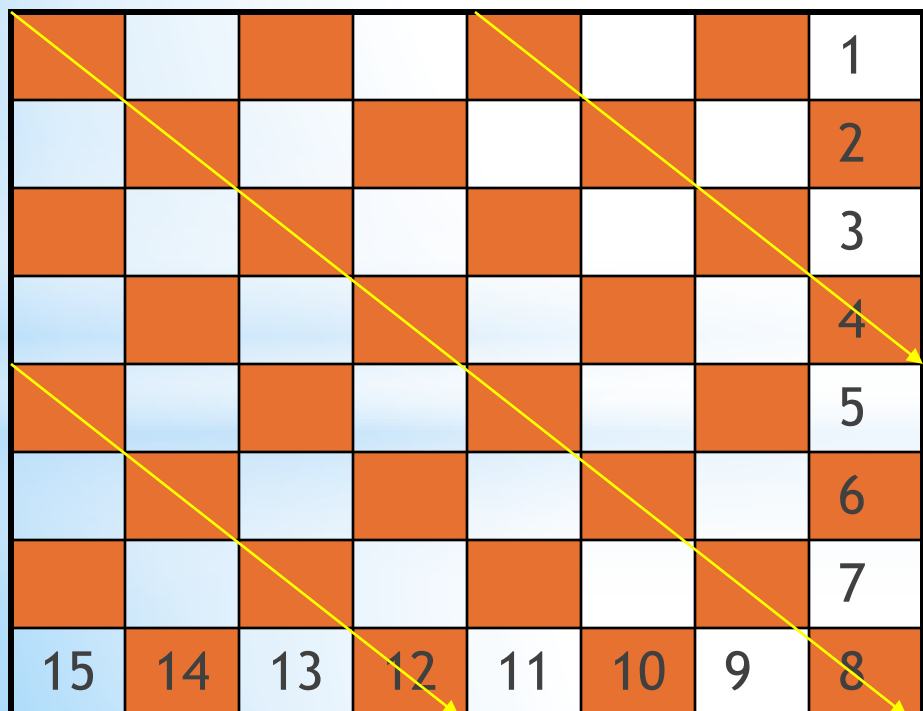


**Ví dụ 4.** Bài toán N quân hậu. Trên bàn cờ kích cỡ  $N \times N$ , hãy đặt N quân hậu mỗi quân trên 1 hàng sao cho tất cả các quân hậu đều không ăn được lẫn nhau.

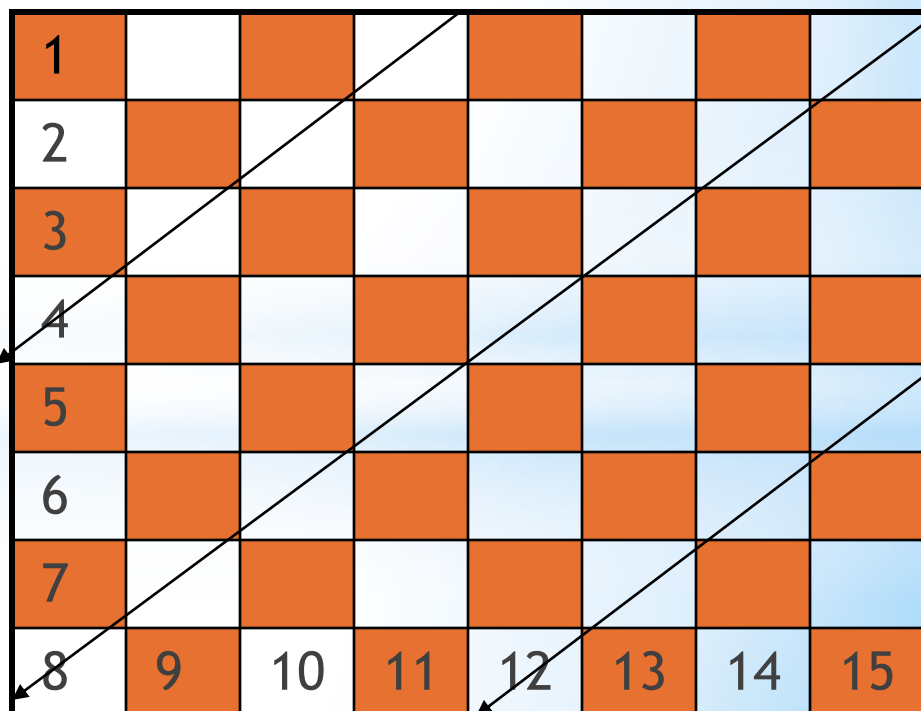
**Lời giải.** Gọi  $X = (x_1, x_2, \dots, x_n)$  là một nghiệm của bài toán. Khi đó,  $x_i = j$  được hiểu là quân hậu hàng thứ  $i$  đặt ở cột  $j$ . Để các quân hậu khác không thể ăn được, quân hậu thứ  $i$  cần không được lấy trùng với bất kỳ cột nào, không được cùng đường chéo xuôi, không được cùng trên đường chéo ngược. Ta có  $n$  cột  $A = (a_1, \dots, a_n)$ , có  $Xuoi[2*n-1]$  đường chéo xuôi,  $Nguoc[2*n-1]$  đường chéo ngược.

Nếu  $x_i = j$  thì  $A[j] = \text{True}$ ,  $Xuoi[i-j+n] = \text{True}$ ,  $Nguoc[i+j-1] = \text{True}$ .

**Đường chéo xuôi: Xuoi  $[i - j + n]$**



**Đường chéo ngược: Nguoc  $[i + j - 1]$**



## Thuật toán:

```
void Try (int i){  
    for(int j=1; j<=n; j++){  
        if( A[j] && Xuoi[ i - j + n ] && Nguoc[i + j -1]){  
            X[i] =j; A[j]=FALSE;  
            Xuoi[ i - j + n]=FALSE;  
            Nguoc[ i + j - 1]=FALSE;  
            if(i==n) Result();  
            else Try(i+1);  
            A[j] = TRUE;  
            Xuoi[ i - j + n] = TRUE;  
            Nguoc[ i + j - 1] = TRUE;  
        }  
    }  
}
```

## 2.5. Thuật toán tham lam (Greedy Algorithm)

### 2.5.1. Giới thiệu thuật toán

**Thuật toán tham lam (Greedy Algorithm):** là thuật toán xem xét quá trình giải quyết bài toán bằng việc tạo nên lựa chọn tối ưu cục bộ tại mỗi bước thực hiện với mong muốn tìm ra lựa chọn tối ưu toàn cục. Giải thuật tham lam thường không mang tính tổng quát. Tuy vậy, bằng việc xem xét các lựa chọn tối ưu cục bộ cho ta lời giải gần đúng với phương án tối ưu toàn cục cũng là giải pháp tốt trong thời gian chấp nhận được.

**Thuật toán Greedy bao gồm 5 thành phần chính:**

1. Một tập các ứng viên mà giải pháp thực hiện tạo ra.
2. Một hàm lựa chọn (selection function) để chọn ứng viên tốt nhất cho giải pháp.
3. Một hàm thực thi (feasibility function) được sử dụng để xác định các ứng viên có thể có đóng góp cho giải pháp thực thi.
4. Một hàm mục tiêu (objective function) dùng để xác định giá trị của phương pháp hoặc một phần của giải pháp.
5. Một hàm giải pháp (solution function) dùng để xác định khi nào giải pháp kết thúc.

### 2.5.2. Activities Slection Problem.

Lựa chọn hành động là bài toán tối ưu tổ hợp điển hình quan tâm đến việc lựa chọn các hành động không mâu thuẫn nhau trong các khung thời gian cho trước. Bài toán được phát biểu như sau:

Cho tập gồm  $n$  hành động, mỗi hành động được biểu diễn như bộ đôi thời gian bắt đầu  $s_i$  và thời gian kết thúc  $f_i$  ( $i=1, 2, \dots, n$ ). Bài toán đặt ra là hãy lựa chọn nhiều nhất các hành động có thể thực hiện bởi một máy hoặc một cá nhân mà không xảy ra tranh chấp. Giả sử mỗi hành động chỉ thực hiện đơn lẻ tại một thời điểm.

#### **Input:**

- Số lượng hành động: 6
- Thời gian bắt đầu  $Start[] = \{ 1, 3, 0, 5, 8, 5 \}$
- Thời gian kết thúc  $Finish[] = \{ 2, 4, 6, 7, 9, 9 \}$

**Output:** Số lượng lớn nhất các hành động có thể thực hiện bởi một người.

$OPT[] = \{0, 1, 3, 4 \}$

**Lời giải.** Thuật toán tham lam giải quyết bài toán được thực hiện như sau:

**Algorithm Greedy-Activities- Selection**(  $N$ ,  $S[]$ ,  $F[]$  ):

**Input:**

- $N$  là số lượng hành động (công việc).
- $S[]$  thời gian bắt đầu mỗi hành động.
- $F[]$  thời gian kết thúc mỗi hành động.

**Output:**

- Danh sách thực thi nhiều nhất các hành động.

**Actions:**

**Bước 1** (sắp xếp). Sắp xếp các hành động theo thứ tự tăng dần của thời gian kết thúc.

**Bước 2** (Khởi tạo) Lựa chọn hành động đầu tiên làm phương án tối ưu ( $OPT=1$ ).  $N = N \setminus \{i\}$ ;

**Bước 3** (Lặp).

```
for each activities  $j \in N$  do {  
    if (  $S[j] \geq F[i]$  ) {  
         $OPT = OPT \cup j$ ;  $i = j$ ;  $N = N \setminus \{i\}$   
    }  
}
```

**Bước 4** (Trả lại kết quả).

Return ( $OPT$ )

**EndActions.**



**Ví dụ. Thuật toán Greedy-ActivitiesN-Selection(int N, int S[], int F[]):**

**Input:**

- Số lượng hành động  $n = 8$ .
- Thời gian bắt đầu  $S[] = \{1, 3, 0, 5, 8, 5, 9, 14\}$ .
- Thời gian kết thúc  $F[] = \{2, 4, 6, 7, 9, 9, 12, 18\}$ .

**Output:**

- $OPT = \{ \text{Tập nhiều nhất các hành động có thể thực hiện bởi một máy} \}$ .

**Phương pháp tiến hành::**

Bước	$i = ? \ j = ?$	$(S[j] \geq F[i]) \ ? \ i = ?$	$OPT = ?$
			$OPT = OPT \cup \{1\}$ .
1	$i=1; \ j=2$	$(3 \geq 2)$ : Yes; $i=2$ .	$OPT = OPT \cup \{2\}$ .
2	$i=2; \ j=3$ ;	$(0 \geq 4)$ : No; $i=2$ .	$OPT = OPT \cup \phi$ .
3	$i=2; \ j=4$ ;	$(5 \geq 4)$ : Yes; $i=4$ .	$OPT = OPT \cup \{4\}$ .
4	$i=4; \ j=5$ ;	$(8 \geq 7)$ : Yes; $i=5$ .	$OPT = OPT \cup \{5\}$ .
5	$i=5; \ j=6$ ;	$(5 \geq 9)$ : No; $i=5$ .	$OPT = OPT \cup \phi$ .
6	$i=5; \ j=7$ ;	$(9 \geq 9)$ : Yes; $i=7$ .	$OPT = OPT \cup \{7\}$ .
7	$i=7; \ j=8$ ;	$(14 \geq 12)$ : Yes; $i=8$ .	$OPT = OPT \cup \{8\}$ .
$OPT = \{ 1, 2, 4, 5, 7, 8 \}$			

**2.5.3. Bài toán n-ropes.** Cho  $n$  dây với chiều dài khác nhau. Ta cần phải nối các dây lại với nhau thành một dây. Chi phí nối hai dây lại với nhau được tính bằng tổng độ dài hai dây. Nhiệm vụ của bài toán là tìm cách nối các dây lại với nhau thành một dây sao cho chi phí nối các dây lại với nhau là ít nhất.

**Input:**

- Số lượng dây: 4
- Độ dài dây  $L[] = \{4, 3, 2, 6\}$

**Output:** Chi phí nối dây nhỏ nhất.

$OPT = 39$

Chi phí nhỏ nhất được thực hiện như sau: lấy dây số 3 nối với dây số 2 để được tập 3 dây với độ dài 4, 5, 6. Lấy dây độ dài 4 nối với dây độ dài 5 ta nhận được tập 2 dây với độ dài 6, 9. Cuối cùng nối hai dây còn lại ta nhận được tập một dây với chi phí là  $6+9=15$ . Như vậy, tổng chi phí nhỏ nhất của ba lần nối dây là  $5 + 9 + 15 = 29$ .

Ta không thể có cách nối dây khác với chi phí nhỏ hơn 39. Ví dụ lấy dây 1 nối dây 2 ta nhận được 3 dây với độ dài  $\{7, 2, 6\}$ . Lấy dây 3 nối dây 4 ta nhận được tập hai dây với độ dài  $\{7, 8\}$ , nối hai dây cuối cùng ta nhận được 1 dây với độ dài 15. Tuy vậy, tổng chi phí là  $7 + 8 + 15 = 30$ .

**Thuật toán.** *Sử dụng phương pháp tham lam dựa vào hàng đợi ưu tiên.*

**Thuật toán Greedy-N-Ropes(int L[], int n):**

**Input:**

- n : số lượng dây.
- L[] : chi phí nối dây.

**Output:**

- Chi phí nối dây nhỏ nhất.

**Actions:**

**Bước 1** . Tạo pq là hàng đợi ưu tiên lưu trữ độ dài n dây.

**Bước 2** (Lặp).

OPT = 0; // Chi phí nhỏ nhất.

While (pq.size>1) {

    First = pq.top; pq.pop(); //Lấy và loại phần tử đầu tiên trong pq.

    Second = pq.top; pq.pop(); //Lấy và loại phần tử kế tiếp trong pq.

    OPT = First + Second; //Giá trị nhỏ nhất để nối hai dây

    Pq.push(First + Second); //Đưa lại giá trị First + Second vào pq.

}

**Bước 3**( Trả lại kết quả).

Return(OPT);

**EndActions.**

**Ví dụ. Thuật toán Greedy-N-Ropes(int L[], int n):**

**Input:**

- Số lượng dây n = 8.
- Chi phí nối dây L[] = { 9, 7, 12, 8, 6, 5, 14, 4}.

**Output:**

- Chi phí nối dây nhỏ nhất.

**Phương pháp tiến hành::**

Bước	Giá trị First, Second	OPT=?	Trạng thái hàng đợi ưu tiên.
		0	4, 5, 6, 7, 8, 9, 12, 14
1	First=4; Second=5	9	6, 7, 8, 9, 9, 12, 14
2	First=6; Second=7	22	8, 9, 9, 12, 13, 14
3	First=8; Second=9	39	9, 12, 13, 14, 17
4	First=9; Second=12	60	13, 14, 17, 21
5	First=13; Second=14	87	17, 21, 27
6	First=17; Second=21	125	27, 38
7	First=27; Second=38	190	65
OPT = 190			

### 2.5.5. Bài toán sắp đặt lại chuỗi ký tự

**Bài toán.** Cho chuỗi ký tự  $s[]$  độ dài  $n$  và số tự nhiên  $d$ . Hãy sắp đặt lại các ký tự trong chuỗi  $s[]$  sao cho hai ký tự giống nhau đều cách nhau một khoảng là  $d$ . Nếu bài toán có nhiều nghiệm, hãy đưa ra một cách sắp đặt đầu tiên tìm được. Nếu bài toán không có lời giải hãy đưa ra thông báo “Vô nghiệm”.

Ví dụ.

**Input:**

- Chuỗi ký tự  $S[] = \text{"ABB"};$
- Khoảng cách  $d = 2$ .

**Output:** BAB

**Input:**

- Chuỗi ký tự  $S[] = \text{"AAA"};$
- Khoảng cách  $d = 2$ .

**Output:** Vô nghiệm.

**Input:**

- Chuỗi ký tự  $S[] = \text{"GEEKSFORGEEKS"};$
- Khoảng cách  $d = 3$ .

**Output:** EGKEGKESFESFO

## 2.5.5. Bài toán sắp đặt lại xâu ký tự

Thuật toán Greedy-Arrang-String (S[], d):

Input:

- Xâu ký tự S[].
- Khoảng cách giữa các ký tự d.

Output: Xâu ký tự được sắp đặt lại thỏa mãn yêu cầu bài toán.

Formats : Greedy-Arrang-String(S, d, KQ);

Actions:

**Bước 1.** Tìm Freq[] là số lần xuất hiện mỗi ký tự trong xâu.

**Bước 2.** Sắp xếp theo thứ tự giảm dần theo số xuất hiện ký tự.

**Bước 3.** (Lắp).

i = 0; k = <Số lượng ký tự trong Freq[]>;

While ( i < k ) {

    p = Max(Freq); //Chọn ký tự xuất hiện nhiều lần nhất.

    For ( t = 0; t < p; t++ ) // điền các ký tự i, i+d, i+2d, ..., i + pd

        if (i+(t\*d) > n ) { < Không có lời giải>; return;>

        KQ[i + (t\*d)] = Freq[i].kytu;

    }

    i++;

}

**Bước 4**( Trả lại kết quả): Return(KQ);

EndActions.

**Greedy-Arrang-String (S[], d):**Input : S[] = “GEEKSFORGEEKS”; d= 3.

**Bước 1.** Tìm tập ký tự và số lần xuất hiện mỗi ký tự.

Freq[]	
G	2
E	4
K	2
S	2
F	1
O	1
R	1

Sắp xếp theo thứ tự giảm dần của số lần xuất hiện.

**Bước 2.** Sắp xếp theo thứ tự giảm dần số lần xuất hiện.

Freq[]	
E	4
G	2
K	2
S	2
F	1
O	1
R	1

**Bước 3.** Lặp.

	KQ[I + p*d]												
I =0	E			E			E			E			
i=1	E	G		E	G		E			E			
i=2	E	G	K	E	G	K	E			E			
i=3	E	G	K	E	G	K	E	S		E	S		
i=4	E	G	K	E	G	K	E	S	F	E	S		
i=5	E	G	K	E	G	K	E	S	F	E	S	O	
i=6	E	G	K	E	G	K	E	S	F	E	S	O	R



## 2.6. Thuật toán chia để trị (Devide and Conquer)

### 2.6.1. Giới thiệu thuật toán

**Thuật toán chia để trị (Devide and Conquer):** dùng để giải lớp các bài toán có thể thực hiện được ba bước:

1. Devide (Chia). Chia bài toán lớn thành những bài toán con có cùng kiểu với bài toán lớn.
2. Conquer (Trị). Giải các bài toán con. Thông thường các bài toán con chỉ khác nhau về dữ liệu vào nên ta có thể thực hiện bằng một thủ tục đệ qui.
3. Combine (Tổng hợp). Tổng hợp lại kết quả của các bài toán con để nhận được kết quả của bài toán lớn.

**Một số thuật toán chi để trị điển hình:**

- Thuật toán tìm kiếm nhị phân (Binary-Search).
- Thuật toán Quick-Sort.
- Thuật toán Merge-Sort.
- Thuật toán Strassen nhân hai ma trận.
- Thuật toán Kuley-Tukey nhân tính toán nhanh dịch chuyển Fourier.
- Thuật toán Karatsuba tính toán phép nhân...

## 2.6.2. Thuật toán nhân nhanh Karatsuba

**Bài toán :** Giả sử ta có hai số nguyên  $a = (a_{n-1}a_{n-2}..a_1a_0)_2$ ,  $b = (b_{n-1}b_{n-2}..b_1b_0)_2$ . Khi đó phép nhân hai số nguyên thực hiện theo cách thông thường ta cần thực hiện  $n^2$  lần tổng của tất cả các tích riêng theo thuật toán sau:

**Thuật toán Multiple( a, b: positive integer):**

Input :

- $a = (a_{n-1}a_{n-2}..a_1a_0)_2$ : số a được biểu diễn bằng n bit ở hệ cơ số 2.
- $b = (b_{n-1}b_{n-2}..b_1b_0)_2$  : số b được biểu diễn bằng n bit ở hệ cơ số 2.

**Output:**

- $c = (c_{2n-1}c_{2n-2}..c_1c_0)_2$ : số c là tích hai số a và b.

**Formats:**  $c = \text{Multiple}(a, b);$

**Actions:**

**Bước 1.** Tính tổng các tích riêng.

```
for ( j = 0; j < n; j++ ) {  
    if ( b_j == 1 ) c_j = a << j;  
    else c_j = 0;  
}
```

**Bước 2.** Tính tổng các tích riêng.

```
p = 0;  
for ( j = 0; j < n; j++ ) p = p + c_j;
```

**Bước 3.** Trả lại kết quả.

```
Return p;
```

**EndActions.**

**Thuật toán nhân nhanh Karatsuba** cho phép ta nhân nhanh hai số ở hệ cơ số bất kỳ với độ phức tạp là  $O(n^{1.59})$ . Thuật toán được thực hiện theo giải thuật chia để trị như sau:

Với mọi số tự nhiên  $x, y$  bất kỳ gồm  $n$  chữ số ở hệ cơ số  $B$  đều tồn tại số tự nhiên  $m \leq n$  sao cho:

$$x = x_1 B^m + x_0 \quad y = y_1 B^m + y_0$$

Với  $x_0, y_0$  nhỏ hơn  $B^m$ . Khi đó:

$$\begin{aligned} xy &= (x_1 B^m + x_0)(y_1 B^m + y_0) \\ xy &= z_2 B^{2m} + z_1 B^m + z_0 \end{aligned}$$

Trong đó:

$$\begin{aligned} z_2 &= x_1 y_1 \\ z_1 &= x_1 y_0 + x_0 y_1 \\ z_0 &= x_0 y_0 \end{aligned}$$

Như vậy ta chỉ cần thực hiện 4 phép nhân và một số phép cộng và một số phép trừ khi phân tích các toán hạng.

**Ví dụ.** Ta cần nhân hai số  $x = 12345$ ,  $y = 6789$  ở hệ cơ số  $B = 10$ . Chọn  $m=3$ , khi đó:

$$12345 = \mathbf{12} \cdot 1000 + \mathbf{345}$$

$$6789 = \mathbf{6} \cdot 1000 + \mathbf{789}$$

Từ đó ta tính được:

$$z_2 = \mathbf{12} \times \mathbf{6} = 72$$

$$z_0 = \mathbf{345} \times \mathbf{789} = 272205$$

$$\begin{aligned} z_1 &= (\mathbf{12} + \mathbf{345}) \times (\mathbf{6} + \mathbf{789}) - z_2 - z_0 = 357 \times 795 - 72 - 272205 \\ &= 283815 - 72 - 272205 = 11538. \end{aligned}$$

$$\text{Kết quả} = 72 \cdot 1000^2 + 11538 \cdot 1000 + 272205 = \mathbf{83810205}.$$

Chú ý. Thời gian thuật toán sẽ thực hiện nhanh hơn khi ta lấy cơ số  $B$  cao (ví dụ  $B = 1000$ ).

### 2.5.3. Tìm tổng dãy con liên tục lớn nhất

**Bài toán:** Cho dãy số nguyên bao gồm cả số âm lẫn số dương. Nhiệm vụ của ta là tìm dãy con liên tục có tổng lớn nhất.

**Ví dụ.** Với dãy số  $A = \{-2, -5, 6, -2, -3, 1, 5, -6\}$  thì tổng lớn nhất của dãy con liên tục ta nhận được là 7.

**Thuật toán 1. Max-SubArray( Arr[], n):** //Độ phức tạp  $O(n^2)$ .

**Bước 1** (Khởi tạo):

Max = Arr[0]; // Chọn Max là phần tử đầu tiên.

**Bước 2** (lặp):

for (i=1; i<n; i++) { //Duyệt từ vị trí 1 đến n-1

S = 0; //Gọi S là tổng liên tục của i số

for ( j =0; j<=n; j++) { //tính tổng của Arr[0],...,Arr[i].

S = S + Arr[j];

if (Max < S ) // Nếu Max nhỏ hơn

Max = S;

}

}

**Bước 3** (Trả lại kết quả):

Return(Max).

**Thuật toán 2. Devide-Conquer ( Arr[], n):** //Độ phức tạp  $O(n\log(n))$ .

```
int maxCrossingSum(int arr[], int l, int m, int h) {  
    int sum = 0, left_sum = INT_MIN, right_sum = INT_MIN;  
    for (int i = m; i >= l; i--) { //Tìm tổng dãy con từ l đến m  
        sum = sum + arr[i];  
        if (sum > left_sum)    left_sum = sum;  
    }  
    sum = 0;  
    for (int i = m+1; i <= h; i++) { //Tìm tổng dãy con từ m+1 đến h  
        sum = sum + arr[i];  
        if (sum > right_sum)    right_sum = sum;  
    }  
    return left_sum + right_sum; //Trả lại kết quả  
}
```

```
int maxSubArraySum(int arr[], int l, int h) {  
    if (l == h) return arr[l];  
    int m = (l + h)/2; // Tìm điểm ở giữa  
    return max(maxSubArraySum(arr, l, m),  
               maxSubArraySum(arr, m+1, h),  
               maxCrossingSum(arr, l, m, h));  
}
```

**Thuật toán 3.** Hãy cho biết kết quả thực hiện chương trình //Độ phức tạp  $O(n)$ .

```
#include<stdio.h>
```

```
int maxSubArraySum(int a[], int n){
    int max_so_far = 0, max_ending_here = 0;
    for(int i = 0; i < n; i++) {
        max_ending_here = max_ending_here + a[i];
        if(max_ending_here < 0)
            max_ending_here = 0;
        if(max_so_far < max_ending_here)
            max_so_far = max_ending_here;
    }
    return max_so_far;
}

int main(){
    int a[] = {-2, -3, 4, -1, -2, 1, 5, -3};
    int n = sizeof(a)/sizeof(a[0]);
    int max_sum = maxSubArraySum(a, n);
    printf("Tong day con lien tuc lon nhat: %d\n", max_sum);
    getchar();return 0;
}
```



## 2.6.4. Thuật toán tìm kiếm nhị phân.

**Bài toán.** Cho một dãy  $A = (a_1, a_2, \dots, a_n)$  đã được sắp xếp theo thứ tự tăng dần. Nhiệm vụ của ta là tìm vị trí của  $x$  xuất hiện trong  $A[]$ .

**Thuật toán Binary-Search( int Arr[], int l, int r, int x ):**

**Input :**

- Arr[] mảng số đã được sắp xếp.
- Giá trị l : vị trí bên trái nhất bắt đầu tìm kiếm.
- Giá trị r : vị trí bên phải nhất bắt đầu tìm kiếm.
- Giá trị x: số cần tìm trong mảng Arr[]

**Output:**

- Return(mid) là vị trí của x trong khoảng l, r.
- Return(-1) nếu x không có mặt trong khoảng l, r.

**Formats:**

Binary-Search(Arr, l, r, x ).

**Độ phức tạp thuật toán:  $O(\log(n))$ :  $n = l-r$ .**

#### 2.6.4. Thuật toán tìm kiếm nhị phân.

```
Int Binary-Search( int Arr[], int l, int r, int x ) {  
    if (r >= l) {  
        int mid = l + (r - l)/2; //Bước chia:Chia bài toán làm hai phần  
        if (arr[mid] == x) //Trị trường hợp thứ nhất  
            return mid;  
        if (arr[mid] > x) //Trị trường hợp thứ hai  
            return binarySearch(arr, l, mid-1, x);  
        return binarySearch(arr, mid+1, r, x); //Trị trường hợp còn lại  
    }  
    return -1; //Kết luận tìm không thấy  
}
```

**Ta có thể khử đệ qui theo thủ tục dưới đây:**

```
int binarySearch(int arr[], int l, int r, int x) {  
    while (l <= r) {  
        int m = l + (r-l)/2; //Bước chia  
        if (arr[m] == x) return m; // Trị trường hợp 1.  
        if (arr[m] < x) l = m + 1; // Trị trường hợp 2.  
        else r = m - 1; // Trị trường hợp 3.  
    }  
    return -1; // Kết luận không tìm thấy.  
}
```

## 2.7. Thuật toán nhánh cận (Branch-And-Bound)

Bài toán tối ưu tổ hợp được phát biểu dưới dạng sau:

$$\text{Tìm min } \{ f(X) : X \in D \}.$$

$$\text{Hoặc tìm max } \{ f(X) : X \in D \}.$$

Trong đó,  $D$  là tập hữu hạn các phần tử. Không hạn chế tính tổng quát của bài toán, ta xem tập  $D$  là tích đề các của các tập hợp khác nhau thỏa mãn tính chất  $P$  nào đó.

$$D = \{ X = (x_1, x_2, \dots, x_n) \in A_1 \times A_2 \times \dots \times A_n : X \text{ thỏa mãn tính chất } P \}$$

- $X \in D$  được gọi là một phương án cần duyệt.
- Hàm  $f(X)$  được gọi là hàm mục tiêu của bài toán.
- Miền  $D$  được gọi là tập phương án của bài toán.
- $X \in D$  thỏa mãn tính chất  $P$  được gọi là tập các ràng buộc của bài toán.

**Xây dựng hàm đánh giá cận trên:** Thuật toán nhánh cận có thể giải được bài toán đặt ra nếu ta tìm được một hàm  $g$  xác định trên tất cả phương án bộ phận của bài toán thỏa mãn bất đẳng thức (\*).

$$g(a_1, a_2, \dots, a_k) \leq \min \{ f(X) : X \in D, x_i = a_i, i = 1, 2, \dots, k \} \quad (*)$$

Nói cách khác, giá trị của hàm  $g$  tại phương án bộ phận cấp  $k$  ( $a_1, a_2, \dots, a_k$ ) không vượt quá giá trị nhỏ nhất của hàm mục tiêu trên tập con các phương án.

$$D(a_1, a_2, \dots, a_k) = \{ X \in D : x_i = a_i, i = 1, 2, \dots, k \}$$

Giá trị của hàm  $g(a_1, a_2, \dots, a_k)$  là cận dưới của hàm mục tiêu trên tập  $D(a_1, a_2, \dots, a_k)$ . Hàm  $g$  được gọi là hàm cận dưới,  $g(a_1, a_2, \dots, a_k)$  gọi là cận dưới của tập  $D(a_1, a_2, \dots, a_k)$ .

## Hạn chế các phương án duyệt:

Giả sử ta đã có hàm  $g$ . Để giảm bớt khối lượng duyệt trên tập phương án trong quá trình liệt kê bằng thuật toán quay lui ta xác định được  $X^*$  là phương án làm cho hàm mục tiêu có giá trị nhỏ nhất trong số các phương án tìm được  $f^* = f(X^*)$ . Ta gọi  $X^*$  là phương án tốt nhất hiện có,  $f^*$  là kỷ lục hiện tại.

Nếu

$$f^* < g(a_1, a_2, \dots, a_k)$$

thì

$$f^* < g(a_1, a_2, \dots, a_k) \leq \min \{ f(X) : X \in D, x_i = a_i, i=1, 2, \dots, k \}.$$

Điều này có nghĩa tập  $D(a_1, a_2, \dots, a_k)$  chắc chắn không chứa phương án tối ưu. Trong trường hợp này ta không cần phải triển khai phương án bộ phận  $(a_1, a_2, \dots, a_k)$ . Tập  $D(a_1, a_2, \dots, a_k)$  cũng bị loại bỏ khỏi quá trình duyệt. Nhờ đó, số các tập cần duyệt nhỏ đi trong quá trình tìm kiếm.

## Thuật toán nhánh cận:

Thuật toán Branch\_And\_Bound (k) {

  for  $a_k \in A_k$  do {

    if (<chấp nhận  $a_k$ >){

$x_k = a_k$ ;

      if (  $k == n$  )

        <Cập nhật kỷ lục>;

      else if (  $g(a_1, a_2, \dots, a_k) \leq f^*$  )

        Branch\_And\_Bound (k+1) ;

    }

  }

}

## Giải bài toán cái túi bằng thuật toán nhánh cận:

Bài toán cái túi có thể được phát biểu tổng quát dưới dạng sau:

Tìm giá trị lớn nhất của hàm mục tiêu  $f(X)$  với  $X \in D$ . Trong đó,  $f(X)$  được xác định như dưới đây:

$$f^* = \max \left\{ f(X) = \sum_{i=1}^n c_i x_i : \sum_{i=1}^n a_i x_i \leq b, x_i \in Z_+, i = 1, 2, \dots, n \right\} \quad (1)$$

$$D = \left\{ X = (x_1, x_2, \dots, x_n) : \sum_{i=1}^n a_i x_i \leq b, x_i \in Z_+, i = 1, 2, \dots, n \right\}$$

**Ví dụ về một bài toán cái túi:**

$$f(X) = 10x_1 + 5x_2 + 3x_3 + 6x_4 \rightarrow \max,$$

$$5x_1 + 3x_2 + 2x_3 + 4x_4 \leq 8,$$

$$x_j \in Z_+, j = 1, 2, 3, 4.$$

**Lời giải:**

**Bước 1.** Sắp xếp các đồ vật thỏa mãn công thức (2):

$$\frac{c_1}{a_1} \geq \frac{c_2}{a_2} \geq \dots \geq \frac{c_n}{a_n} \quad (2)$$

**Bước 2 (Lập):** Lập trên các bài toán bộ phận cấp  $k=1, 2, \dots, n$ :

• Giá trị sử dụng của  $k$  đồ vật trong túi:

$$\sigma_k = \sum_{i=1}^k c_i x_i \quad (3)$$

• Trọng lượng còn lại của túi:

$$b_k = b - \sum_{i=1}^k a_i x_i \quad (4)$$

• Cận trên của phương án bộ phận cấp  $k$ :

$$g(x_1, x_2, \dots, x_k) = \sigma_k - b_k \cdot \frac{c_{k+1}}{a_{k+1}} \quad (5)$$

**Bước 3 (Trả lại kết quả):** Phương án tối ưu và giá trị tối ưu tìm được.



**Thuật toán Brach\_And\_Bound (k) {**

**for j = 1; j<=0; j--){**

**x[k] = j;**

$$\sigma_k = \sigma_k + c_i x_i; \quad b_k = b_k + a_k x_k;$$

**If (k==n) <Ghi nhận kỷ lục>;**

**else if ( $\delta_k + (c_{k+1} * b_k) / a_{k+1} > \text{FOPT}$ )**

**Branch\_And\_Bound(k+1);**

$$\sigma_k = \sigma_k - c_k x_k; \quad b_k = b_k - a_k x_k;$$

**}**

**}**

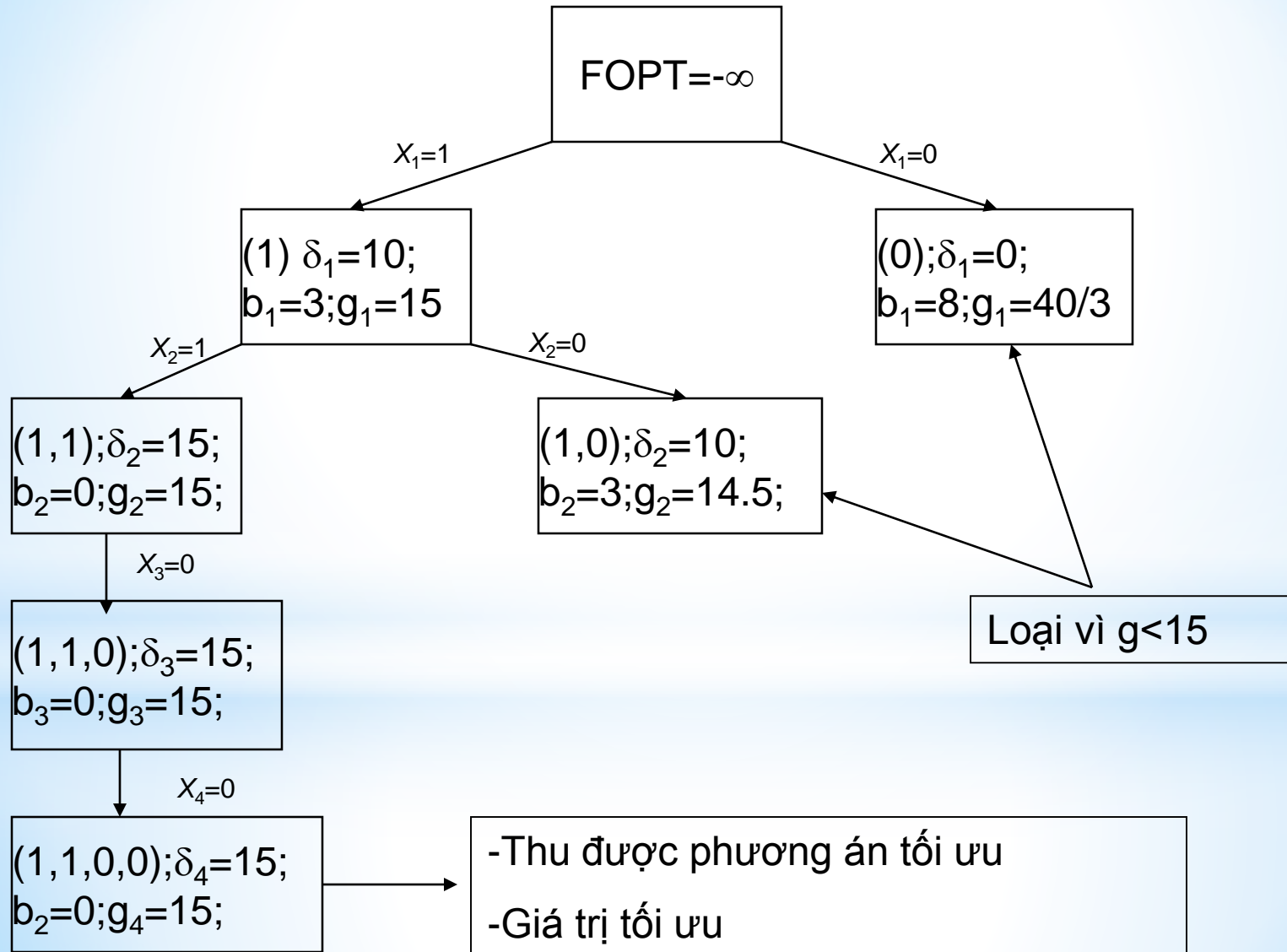
```

void Back_Track(int i){
    int j, t = ((b-weight)/A[i]);
    for(int j= t; j>=0; j--){
        X[i] = j; weight = weight+A[i]*X[i];
        cost = cost + C[i]*X[i];
        if (i==n) Update();
        else if ( cost + ( C[i+1]*((b- weight)/A[i+1]))>FOPT)
            Back_Track(i+1);
        weight = weight-A[i]*X[i];
        cost = cost - C[i]*X[i];
    }
}

```

Ví dụ giải bài toán:

$$f(X) = 10x_1 + 5x_2 + 3x_3 + 6x_4 \rightarrow \max,$$
$$5x_1 + 3x_2 + 2x_3 + 4x_4 \leq 8,$$
$$x_j \in \mathbb{Z}_+, j = 1, 2, 3, 4.$$



## **2.8. Thuật toán qui hoạch động(Dynamic Programming)**

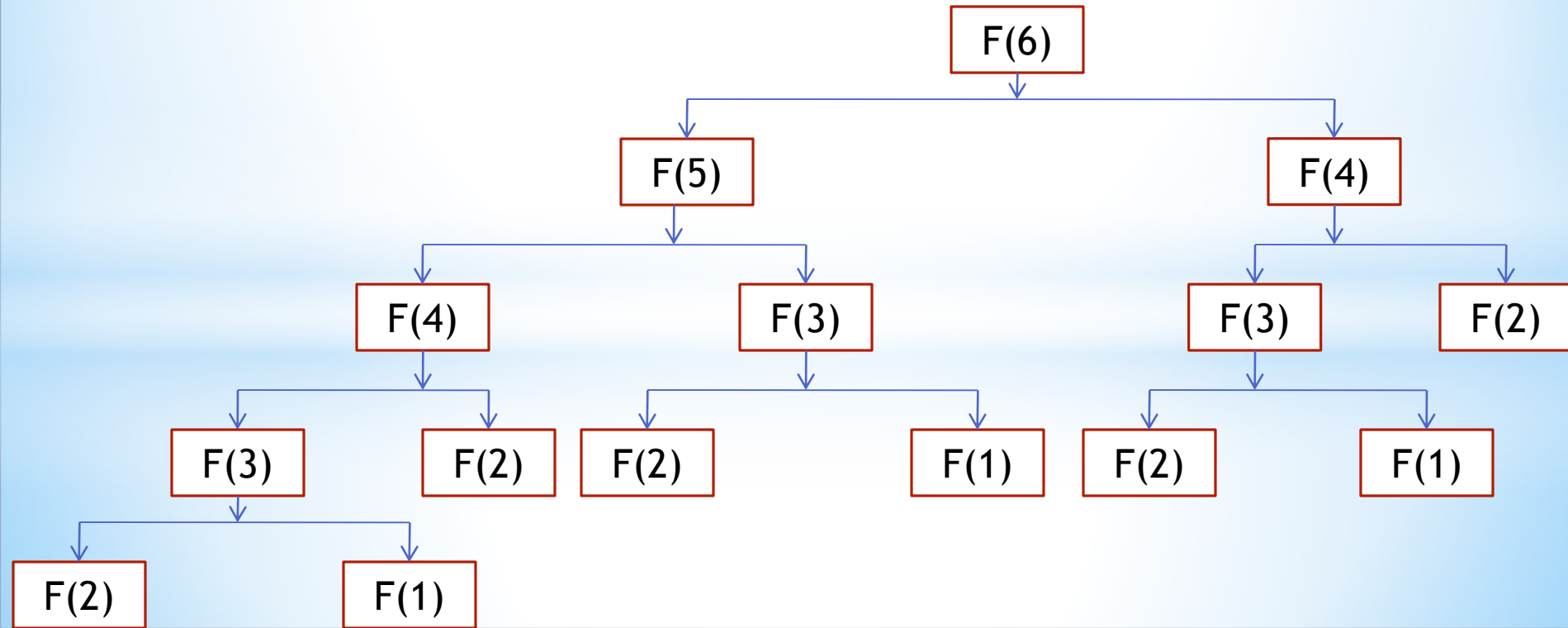
Phương pháp qui hoạch động dùng để giải lớp các bài toán thỏa mãn những điều kiện sau:

- Bài toán lớn cần giải có thể phân rã được thành nhiều bài toán con. Trong đó, sự phối hợp lời giải của các bài toán con cho ta lời giải của bài toán lớn. Bài toán con có lời giải đơn giản được gọi là cơ sở của qui hoạch động. Công thức phối hợp nghiệm của các bài toán con để có nghiệm của bài toán lớn được gọi là công thức truy hồi của qui hoạch động.
- Phải có đủ không gian vật lý lưu trữ lời giải các bài toán con (Bảng phương án của qui hoạch động). Vì qui hoạch động đi giải quyết tất cả các bài toán con, do vậy nếu ta không lưu trữ được lời giải các bài toán con thì không thể phối hợp được lời giải giữa các bài toán con.
- Quá trình giải quyết từ bài toán cơ sở (bài toán con) để tìm ra lời giải bài toán lớn phải được thực hiện sau hữu hạn bước dựa trên bảng phương án của qui hoạch động.

**Ví dụ.** Tìm số Fibonacci thứ  $n$ . Cho  $F_1 = F_2 = 1$ ;  $F_n = F_{n-1} + F_{n-2}$  với  $n \geq 3$ . Hãy tìm  $F_6 = ?$ .

```
#include <iostream.h>
int F( int i ) {
    if ( i < 3 ) return (1);
    else return (F(i-1) + F(i-2))
}
void main(void) {
    cout<<"\n F[6] "<<F(6);
}
```

```
#include <iostream.h>
int main()
{
    int F[100]; F[1]=1; F[2] =1;
    for (int i=3; i<=6; i++)
        F[i] = F[i-1] + F[i-2];
    cout<<"\n F[6]="<<F[6];
}
```



## MỘT VÀI NHẬN XÉT

- 1. Cách giải thứ nhất:** Sử dụng lời gọi đệ qui đến  $F(6)$ . Để tính được  $F(6)$  cách giải thứ nhất cần phải tính 1 lần  $F(5)$ , 2 lần  $F(4)$ , 3 lần  $F(3)$ , 5 lần  $F(2)$ , 3 lần  $F(1)$ .
- 2. Cách giải thứ hai (Qui hoạch động):** về bản chất cũng là đệ qui. Tuy nhiên phương pháp thực hiện theo các bước sau:
  - **Bước cơ sở** (*thường rất đơn giản*): Tính  $F[1] = 1$ ,  $F[2] = 1$ .
  - **Công thức truy hồi** (*thường không đơn giản*): Lưu lời giải các bài toán con biết trước vào bảng phương án (*ở đây là mảng một chiều  $F[100]$* ). Sử dụng lời giải của bài toán con trước để tìm lời giải của bài toán con tiếp theo. Trong bài toán này là  $F[i] = F[i-1] + F[i-2]$  với  $n \geq 3$ . Bằng cách lưu trữ vào bảng phương án, ta chỉ cần giải mỗi bài toán con một lần. Tuy vậy, cái giá phải trả là liệu ta có đủ không gian nhớ để lưu trữ lời giải các bài toán con hay không?
  - **Truy vết** (*thường đơn giản*): Đưa ra nghiệm của bài toán bằng việc truy lại dấu vết phối hợp lời giải các bài toán con để có được nghiệm của bài toán lớn. Trong bài toán này, vết của ta chính là  $F[6]$  do vậy ta không cần phải làm gì thêm nữa.

**Ví dụ.** Tìm số các cách chia số tự nhiên  $n$  thành tổng các số tự nhiên nhỏ hơn  $n$ . Các cách chia là hoán vị của nhau chỉ được tính là một cách.

**Lời giải.** Gọi  $F[m, v]$  là số cách phân tích số  $v$  thành tổng các số nguyên dương nhỏ hơn hoặc bằng  $m$ . Khi đó, bài toán trên tương ứng với bài toán tìm  $F[n, n]$  (số các cách chia số  $n$  thành tổng các số nguyên dương nhỏ hơn hoặc bằng  $n$ ).

Ta nhận thấy, số các cách chia số  $v$  thành tổng các số nhỏ hơn hoặc bằng  $m$  được chia thành hai loại:

- **Loại 1:** Số các cách phân tích số  $v$  thành tổng các số nguyên dương không chứa số  $m$ . Điều này có nghĩa số các cách phân tích Loại 1 là số các cách chia số  $v$  thành tổng các số nguyên dương nhỏ hơn hoặc bằng  $m-1$ . Như vậy số Loại 1 chính là  $F[m-1, v]$ .
- **Loại 2:** . Số các cách phân tích số  $v$  thành tổng các số nguyên dương chứa ít nhất một số  $m$ . Nếu ta xem sự xuất hiện của một hay nhiều số  $m$  trong phép phân tích  $v$  chỉ là 1, thì theo nguyên lý nhân số lượng các cách phân tích loại này chính là số cách phân tích số  $v$  thành tổng các số nhỏ hơn  $m-v$  hay  $F[m, v-m]$ .

Trong trường hợp  $m > v$  thì  $F[m, v-m] = 0$  nên ta chỉ có các cách phân tích loại 1. Trong trường hợp  $m \leq v$  thì ta có cả hai cách phân tích loại 1 và loại 2. Vì vậy:

- $F[m, v] = F[m-1, v]$  nếu  $m > v$ .
- $F[m, v] = F[m-1, v] + F[m, v-m]$  nếu  $m \leq v$ .



**Bảng phương án:** Sử dụng công thức truy hồi cho phép ta tính toán  $F[m,v]$  thông qua  $F[m-1, v]$  và  $F[m, v-m]$ . Tập các giá trị tính toán theo phương pháp lặp của hệ thức truy hồi được gọi là bảng phương án của bài toán. Ví dụ với  $n=5$  ta sẽ có bảng phương án như sau:

- Dòng 0 ghi lại  $F[0, v]$ . Trong đó,  $F[0,0] = 1$ ;  $F(0,v) = 0$ , với  $v > 1$ .
- Dựa vào dòng 0, các dòng tiếp theo trong bảng được tính toán như bảng dưới đây:
  - $F[m, v] = F[m-1, v]$  nếu  $m > v$ ;
  - $F[m, v] = F[m-1, v] + F[m, v-m]$  nếu  $m \leq v$ .
  - Từ đó ta có  $F[5, 5] = F[4,5] + F[5,0] = 7$  (cách).

	0	1	2	3	4	5
0	1	0	0	0	0	0
1	1	1	1	1	1	1
2	1	1	2	2	3	3
3	1	1	2	3	4	5
4	1	1	2	3	5	6
5	1	1	2	3	5	7

## Tổng quát.

- **Bước cơ sở:**  $F[0,0] = 1$ ;  $F[0,v] = 0$  với  $v > 0$ .
- **Tính toán giá trị bằng phương án dựa vào công thức truy hồi:**
  - $F[m, v] = F[m-1, v]$  nếu  $v > m$ ;
  - $F[m, v] = F[m-1, v] + F[m, v-m]$  nếu  $m \geq v$ .
- **Lưu vết.** tìm  $F[5, 5] = F[4, 5] + F[5, 0] = 6 + 1 = 7$ .

	0	1	2	3	4	5	
$F[m,v]$	0	1	0	0	0	0	$\rightarrow v$
1	1	1	1	1	1	1	
2	1	1	2	2	3	3	
3	1	1	2	3	4	5	
4	1	1	2	3	5	6	
5	1	1	2	3	5	7	
$\downarrow m$							

**Bài tập 1.** Hãy cho biết kết quả thực hiện chương trình dưới đây?

```
#include <iostream.h>

void main (void ) {
    int F[100][100], n, m, v;
    cout<<"Nhập n="; cin>>n; //Nhập n=5
    for ( int j=0; j <=n; j++) F[0][j] =0; //Thiết lập dòng 0 là 0.
    F[0][0] = 1; //Duy chỉ F[0][0] thiết lập là 1.
    for (m =1; m<=n; m++) {
        for (v= 0; v<=n; v++){
            if ( m > v ) F[m][v] = F[m-1][v];
            else F[m][v] = F[m-1][v] + F [m][v-m];
        }
    }
    cout<<" Kết quả:"<<F[n][n]<<endl;
}
```

## Bài tập 2. Hãy cho biết kết quả thực hiện chương trình dưới đây?

```
#include <iostream.h>
int F0[100], F[100], n, m, v, dem=0;
void Init(void){
    cout<<"Nhap n="; cin>>n;
    for(int i=0; i<=n; i++) F0[i]=0;
    F0[0]=1;
}
void Result(void) {
    cout<<"\n Ket qua buoc "<<++dem<<":";
    for(int i=0; i<=n; i++)
        cout<<F0[i]<<" ";
}
void Replace(void ) {
    for(int i=0; i<=n; i++)
        F0[i]=F[i];
}
```

```
int main (void ) {
    Init();
    for (m=1; m<=n; m++) {
        for (v= 0; v<=n; v++){
            if ( v < m ) F[v] = F0[v];
            else F[v] = F0[v] + F[v-m];
        }
        Result();Replace();
    }
    Result();
    cout<<"\n Ket qua:"<<F[n]<<endl;
    system("PAUSE");return 0;
}
```

### Bài tập 3. Hãy cho biết kết quả thực hiện chương trình dưới đây?

```
#include <iostream.h>
int F[100], n, m, v, dem=0;
void Init(void){
    cout<<"Nhap n="; cin>>n; //Nhap n=5
    for(int i=0; i<=n; i++) F[i]=0; //Thiet lap F0[i]=0; i=0, 1, 2, ...,n;
    F[0]=1;
}
void Result(void) {
    cout<<"\n Ket qua buoc "<<dem<<": ";
    for(int i=0; i<=n; i++)
        cout<<F[i]<<" ";
}
int main (void ) { Init();
    for (m=1; m<=n; m++) { Result();
        for (v= m; v<=n; v++){
            F[v] = F[v] + F[v-m];
        }
    }
    Result();
    cout<<"\n Ket qua:"<<F[n]<<endl;
    system("PAUSE");return 0;
}
```

#### Bài tập 4. Hãy cho biết kết quả thực hiện chương trình dưới đây?

```
#include <iostream.h>
int n, dem=0;
int F(int m, int v){
    if (m==0) {
        if (v==0) return(1);
        else return(0);
    }
    else {
        if (m>v ) return (F(m-1, v));
        else return(F(m-1,v)+F(m,v-m));
    }
}
int main (void ) {
    cout<<"\n Nhap n=";<<cin>>n;
    cout<<"\n Ket qua:"<<F(n,n)<<endl;
    system("PAUSE");return 0;
}
```

## 2.9. Tìm kiếm mẫu

Đối sánh xâu (String matching) là một chủ đề quan trọng trong lĩnh vực xử lý văn bản. Các thuật toán đối sánh xâu được xem là những thành phần cơ sở được cài đặt cho các hệ thống thực tế đang tồn tại trong hầu hết các hệ điều hành. Hơn thế nữa, các thuật toán đối sánh xâu cung cấp các mô hình cho nhiều lĩnh vực khác nhau của khoa học máy tính: xử lý ảnh, xử lý ngôn ngữ tự nhiên, tin sinh học và thiết kế phần mềm.

String-matching được hiểu là việc tìm một hoặc nhiều xâu mẫu (pattern) xuất hiện trong một văn bản (có thể là rất dài). Ký hiệu xâu mẫu hay xâu cần tìm là  $X = (x_0, x_1, \dots, x_{m-1})$  có độ dài  $m$ . Văn bản  $Y = (y_0, y_1, \dots, y_{n-1})$  có độ dài  $n$ . Cả hai xâu được xây dựng từ một tập hữu hạn các ký tự Alphabet ký hiệu là  $\Sigma$  với kích cỡ là  $\sigma$ . Như vậy một xâu nhị phân có độ dài  $n$  ứng dụng trong mật mã học cũng được xem là một mẫu. Một chuỗi các ký tự ABD độ dài  $m$  biểu diễn các chuỗi AND cũng là một mẫu.

### Input:

- Xâu mẫu  $X = (x_0, x_1, \dots, x_m)$ , độ dài  $m$ .
- Văn bản  $Y = (y_0, x_1, \dots, y_n)$ , độ dài  $n$ .

### Output:

- Tất cả vị trí xuất hiện của  $X$  trong  $Y$ .



## Phân loại các thuật toán đối sánh mẫu

Thuật toán đối sánh mẫu đầu tiên được đề xuất là Brute-Force. Thuật toán xác định vị trí xuất hiện của X trong Y với thời gian  $O(m.n)$ . Nhiều cải tiến khác nhau của thuật toán Brute-Force đã được đề xuất nhằm cải thiện tốc độ tìm kiếm mẫu. Ta có thể phân loại các thuật toán tìm kiếm mẫu thành các lớp:

- **Tìm kiếm mẫu từ bên trái qua bên phải:** Harrison Algorithm, Karp-Rabin Algorithm, Morris-Pratt Algorithm, Knuth- Morris-Pratt Algorithm, Forward Dawg Matching algorithm , Apostolico-Crochemore algorithm, Naive algorithm.
- **Tìm kiếm mẫu từ bên phải qua bên trái:** Boyer-Moore Algorithm , Turbo BM Algorithm, Colussi Algorithm, Sunday Algorithm, Reverse Factorand Algorithm, Turbo Reverse Factor, Zhu and Takaoka and Berry-Ravindran Algorithms.
- **Tìm kiếm mẫu từ một vị trí cụ thể:** Two Way Algorithm, Colussi Algorithm , Galil-Giancarlo Algorithm, Sunday's Optimal Mismatch Algorithm, Maximal Shift Algorithm, Skip Search, KMP Skip Search and Alpha Skip Search Algorithms.
- **Tìm kiếm mẫu từ bất kỳ:** Horspool Algorithm, Boyer-Moore Algorithm, Smith Algorithm , Raita Algorithm.

## Một số khái niệm và định nghĩa cơ bản về tìm kiếm mẫu:

Giả sử Alphabet là tập hợp (hoặc tập con) các mã ASCII. Một từ  $w = (w_0, w_1, \dots, w_l)$  có độ dài  $l$ ,  $w_l = \text{null}$  giống như biểu diễn của ngôn ngữ C. Khi đó ta định nghĩa một số thuật ngữ sau:

- *Prefix (tiền tố)*. Từ  $u$  được gọi là tiền tố của từ  $w$  nếu tồn tại một từ  $v$  để  $w = uv$  ( $v$  có thể là rỗng). Ví dụ:  $u = \text{“AB”}$  là tiền tố của  $w = \text{“ABCDEF”}$  và  $u = \text{“com”}$  là tiền tố của  $w = \text{“communication”}$ .
- *Suffix (hậu tố)*. Từ  $v$  được gọi là hậu tố của từ  $w$  nếu tồn tại một từ  $u$  để  $w = uv$  ( $u$  có thể là rỗng). Ví dụ:  $v = \text{“EF”}$  là hậu tố của  $w = \text{“ABCDEF”}$  và  $v = \text{“tion”}$  là hậu tố của  $w = \text{“communication”}$ .
- *Factor (substring, subword)*. Một từ  $z$  được gọi là một xâu con, từ con hay nhân tố của từ  $w$  nếu tồn tại hai từ  $u, v$  ( $u, v$  có thể rỗng) sao cho  $w = uzv$ . Ví dụ từ  $z = \text{“CD”}$  là factor của từ  $w = \text{“ABCDEF”}$  và  $z = \text{“muni”}$  là factor của  $w = \text{“communication”}$ .
- *Period (đoạn)*. Một số tự nhiên  $p$  được gọi là đoạn của từ  $w$  nếu với mọi  $i$  ( $0 \leq i < m-1$ ) thì  $w[i] = w[i+p]$ . Giá trị đoạn nhỏ nhất của  $w$  được gọi là đoạn của  $w$  ký hiệu là  $\text{pre}(w)$ . Ví dụ  $w = \text{“ABABCDEF”}$ , khi đó tồn tại  $p=2$ .
- *Periodic (tuần hoàn)*. Từ  $w$  được gọi là tuần hoàn nếu đoạn của từ nhỏ hơn hoặc bằng  $l/2$ . Trường hợp ngược lại được gọi là không tuần hoàn. Ví dụ từ  $w = \text{“ABAB”}$  là từ tuần hoàn. Từ  $w = \text{“ABABCDEF”}$  là không tuần hoàn.

## Một số khái niệm và định nghĩa cơ bản về tìm kiếm mẫu:

- *Basic word* (từ cơ sở). Từ  $w$  được gọi là từ cơ sở nếu nó không thể viết như lũy thừa của một từ khác. Không tồn tại  $z$  và  $k$  để  $z^k = w$ .
- *Boder word* (từ biên). Từ  $z$  được gọi là boder của  $w$  nếu tồn tại hai từ  $u, v$  sao cho  $w = uz = zv$ . Khi đó  $z$  vừa là tiền tố vừa là hậu tố của  $w$ . Trong tình huống này  $|u| = |v|$  là một đoạn của  $w$ .
- *Reverse word* (Từ đảo). Từ đảo của từ  $w$  có độ dài  $l$  ký hiệu là  $wR = (w_{r-1}, w_{r-2}, \dots, w_1, w_0)$ .
- *Deterministic Finite Automata (DFA)*. Một automat hữu hạn  $A$  là bộ bốn  $(Q, q_0, T, E)$  trong đó:
  - $Q$  là tập hữu hạn các trạng thái.
  - $q_0$  là trạng thái khởi đầu.
  - $T$  là tập con của  $Q$  là tập trạng thái dừng.
  - $E$  là tập con của  $(Q, \Sigma, T)$  tập các chuyển dịch.

Ngôn ngữ  $L(A)$  đoán nhận bởi  $A$  được định nghĩa:

$$\{w \in \Sigma^* : \exists q_0, \dots, q_n, n = |w|, q_n \in T, \forall 0 \leq i < n, (q_i, w[i], q_{i+1}) \in \delta\}$$

## 2.9.1. Thuật toán Brute-Force

### Đặc điểm:

- Không có pha tiền xử lý.
- Sử dụng không gian nhớ phụ hằng số.
- Quá trình so sánh thực hiện theo bất kỳ thứ tự nào.
- Độ phức tạp thuật toán là  $O(n.m)$ ;

### Thuật toán Brute-Force:

#### Input :

- Xâu mẫu  $X = (x_0, x_1, \dots, x_m)$ , độ dài  $m$ .
- Văn bản nguồn  $Y = (y_1, y_2, \dots, y_n)$  độ dài  $n$ .

#### Output:

- Mọi vị trí xuất hiện của  $X$  trong  $Y$ .

**Formats:** Brute-Force( $X, m, Y, n$ );

#### Actions:

```
for ( j = 0; j <= (n-m); j++) { //duyet từ trái qua phải xâu X
    for (i =0; i<m && X[i] == Y[i+j]; i++) ; //Kiểm tra mẫu
    if (i>=m) OUTPUT (j);
}
```

**EndActions.**

## 2.9.2. Thuật toán Knuth-Morris-Pratt

### Đặc điểm:

- Thực hiện từ trái sang phải.
- Có pha tiền xử lý với độ phức tạp  $O(m)$ .
- Độ phức tạp thuật toán là  $O(n + m)$ ;

**Thuật toán PreKmp:** //thực hiện bước tiền xử lý

### Input :

- Xâu mẫu  $X = (x_0, x_1, \dots, x_m)$ , độ dài  $m$ .

**Output:** Mảng giá trị kmpNext[].

### Formats:

- PreKmp( $X, m, \text{kmpNext}$ );

### Actions:

```
i = 1; kmpNext[0] = 0; len = 0; //kmpNex[0] luôn là 0
while (i < m) {
    if (X[i] == X[len] ) { //Nếu X[i] = X[len]
        len++; kmpNext[i] = len; i++;
    }
    else { // Nếu X[i] != X[len]
        if ( len != 0 ) { len = kmpNext[len-1]; }
        else { kmpNext[i] = 0; i++; }
    }
}
```

**EndActions.**

## Kiểm nghiệm PreKmp (X, m, kmpNext):

- $X[] = \text{"ABABCABAB"} , m = 9$ .

i=?	(X[i]== X[Len])?	Len =?	kmpNext[i]=?
		Len =0	kmpNext[0]=0
i=1	('B'=='A'): No	Len =0	kmpNext[1]=0
i=2	('A'=='A'): Yes	Len =1	kmpNext[2]=1
i=3	('B'=='B'): Yes	Len=2	kmpNext[3]=2
i=4	('C'=='A'): No	Len=0	kmpNext[4]=0
i=5	('A'=='A'): Yes	Len=1	kmpNext[5]=1
i=6	('B'=='B'): Yes	Len=2	kmpNext[6]=2
i=7	('A'=='A'): Yes	Len=3	kmpNext[6]=3
i=8	('B'=='B'): Yes	Len=4	kmpNext[6]=4

Kết luận:  $kmpNext[] = \{0, 0, 1, 2, 0, 1, 2, 3, 4\}$ .



**Kiểm nghiệm PreKmp (X, m, kmpNext) với  $X[] = \text{"AABAACAABAA"}\text{, } m = 11\text{.}$**

i=?	(X[i]== X[len])?	Len =?	kmpNext[i]=?
		Len =0	kmpNext[0]=0
i=1	('A'=='A'): Yes	Len =1	kmpNext[1]=1
i=2	('B'=='A'): No	Len =0	kmpNext[2]=chưa xác định
i=2	('B'=='A'): No	Len=0	kmpNext[2]=0
i=3	('A'=='A'): Yes	Len=1	kmpNext[3]=1
i=4	('A'=='A'): Yes	Len=2	kmpNext[4]=2
i=5	('C'=='B'): No	Len=1	kmpNext[5]=chưa xác định
i=5	('C'=='A'): No	Len=0	kmpNext[5]=chưa xác định
i=5	('C'=='A'): No	Len=0	kmpNext[5]=0
i=6	('A'=='A'): Yes	Len =1	kmpNext[6]=1
i=7	('A'=='A'): Yes	Len =2	kmpNext[7]=2
i=8	('B'='B'):Yes	Len=3	kmpNext[8] = 3
i=9	('A'='A'):Yes	Len=4	kmpNext[9] = 4
i=10	('A'='A'):Yes	Len=5	kmpNext[10] = 5
Kết luận: kmpNext = {0, 1, 0, 1, 2, 0, 1, 2, 3, 4, 5}			



## Thuật toán Knuth-Morris-Partt:

### Input :

- Xâu mẫu  $X = (x_0, x_1, \dots, x_m)$ , độ dài  $m$ .
- Văn bản  $Y = (y_0, y_1, \dots, y_n)$ , độ dài  $n$ .

### Output:

- Tất cả vị trí xuất hiện  $X$  trong  $Y$ .

**Formats:** Knuth-Morris-Partt( $X, m, Y, n$ );

### Actions:

**Bước 1** (Tiền xử lý):

preKmp( $x, m, kmpNext$ ); //Tiền xử lý với độ phức tạp  $O(m)$

**Bước 2** (Lặp):

$i = 0; j = 0;$

```
while (i < n) {  
    if ( X[j] == Y[i] ) { i++; j++; }  
    if ( i == m ) {  
        < Tìm thấy mẫu ở vị trí i-j>;  
        j = kmpNext[j-1];  
    }  
    else if ( i < n && X[j] != Y[i] ) {  
        if ( j != 0 ) j = kmpNext[ j-1];  
        else i = i + 1;  
    }  
}
```

}

**EndActions.**

## Kiểm nghiệm Knuth-Moriss-Patt (X, m, Y, n):

- $X[] = \text{"ABABCABAB"} , m = 9.$
- $Y[] = \text{"ABABDABACDABABCABAB"} , n = 19$

Bước 1 (Tiền xử lý). Thực hiện Prekmp(X, m, kmpNext) ta nhận được:

$kmpNext[] = \{ 0, 0, 1, 2, 0, 1, 2, 3, 4 \}$

Bước 2 (Lặp):

$(X[i]==Y[i])?$	$(J == 9)?$	$I = ? J = ?$	
$(X[0]==Y[0]): \text{Yes}$	No	$i=1, j=1$	
$(X[1]==Y[1]): \text{Yes}$	No	$i=2, j=2$	
$(X[2]==Y[2]): \text{Yes}$	No	$i=3, j=3$	
$(X[3]==Y[3]): \text{Yes}$	No	$i=4, j=4$	
$(X[4]==Y[4]): \text{No}$	No	$i=4, j=2$	
$(X[2]==Y[4]): \text{No}$	No	$i=4, j=0$	
$(X[0]==Y[4]): \text{No}$	No	$i=5, j=0$	
$(X[0]==Y[5]): \text{Yes}$	No	$i=6, j=1$	
$(X[1]==Y[6]): \text{Yes}$	No	$i=7, j=2$	
.....			

## 2.9.3. Thuật toán Boyer-More

### Đặc điểm:

- Thực hiện từ phải sang trái.
- Pha tiền xử lý có độ phức tạp  $O(m + \sigma)$ .
- Pha tìm kiếm có Độ phức tạp thuật toán là  $O(n.m)$ ;

### Thuật toán Boyer-More:

#### Input :

- Xâu mẫu  $X = (x_0, x_1, \dots, x_m)$ , độ dài  $m$ .
- Văn bản  $Y = (y_1, y_2, \dots, y_n)$  độ dài  $n$ .

#### Output:

- Đưa ra mọi vị trí xuất hiện của  $X$  trong  $Y$ .

**Formats:**  $k = \text{Boyer-More}(X, m, Y, n)$ ;

#### Actions:

Bước 1 (Tiền xử lý):

- Xây dựng tập hậu tố tốt của  $X$ :  $\text{bmGs}[]$ .
- Xây dựng tập ký tự không tốt của  $X$ :  $\text{bmBc}[]$ .

Bước 2 (Tìm kiếm):

- Tìm kiếm dựa vào  $\text{bmGs}[]$  và  $\text{bmBc}[]$ .

**EndActions.**

## 2.3. Thuật toán Boyer-More

```
Void BM(char *x, int m, char *y, int n) {  
    int i, j, bmGs[XSIZE], bmBc[ASIZE];  
    /* Bước tiền xử lý */  
    preBmGs(x, m, bmGs); //xây dựng good suffix  
    preBmBc(x, m, bmBc); // xây dựng bad character shift  
    /* Bước tìm kiếm */  
    j = 0;  
    while (j <= n - m) {  
        for (i = m - 1; i >= 0 && x[i] == y[i + j]; --i);  
        if (i < 0) {  
            cout<<"Vi tri:"<<(j)<<endl;  
            j += bmGs[0];  
        }  
        else  
            j += MAX(bmGs[i], bmBc[y[i + j]] - m + 1 + i);  
    }  
}
```

## 2.9.4. Thuật toán Rabin-Karp

### Đặc điểm:

- Sử dụng hàm băm (hash function).
- Thực hiện pha tiền xử lý với độ phức tạp  $O(m)$ .
- Độ phức tạp thuật toán là  $O(n + m)$ ;

### Thuật toán Rabin-Karp:

#### Input :

- Xâu mẫu  $X = (x_0, x_1, \dots, x_m)$ , độ dài  $m$ .
- Văn bản  $Y = (y_0, y_1, \dots, y_n)$ , độ dài  $n$ .

#### Output:

- Tất cả vị trí của  $X$  trong  $Y$ .

**Formats:** Rabin-Karp( $X, m, Y, n$ );

#### Actions:

```
#define REHASH(a, b, h) (((h) - (a)*d) << 1) + (b)) //Hàm băm
int d, hx, hy, i, j;
for (d = i = 1; i < m; ++i) d = (d<<1);
for (hy = hx = i = 0; i < m; ++i) { hx = ((hx<<1) + x[i]); hy = ((hy<<1) + y[i]);}
j = 0;
while (j <= n-m) {
    if (hx == hy && memcmp(x, y + j, m) == 0) OUT(j);
    hy = REHASH(y[j], y[j + m], hy);
    ++j;
}
```

**EndActions.**

## 2.9.5. Thuật toán Automat hữu hạn

### Đặc điểm:

- Xây dựng Automat hữu hạn đoán nhận ngôn ngữ  $\Sigma^*x$ .
- Sử dụng không gian nhớ phụ  $O(m\sigma)$ .
- Thực hiện pha tiền xử lý với thời gian  $O(m\sigma)$ ;
- Pha tìm kiếm có độ phức tạp tính toán  $O(n)$ .

### Mô tả phương pháp:

Một Automat hữu hạn đoán nhận từ  $x$  là  $A(x) = (Q, q_0, T, E)$  đoán nhận ngôn ngữ  $\Sigma^*x$  được định nghĩa như sau:

- $Q$  là tập các tiền tố của  $x$ :  $Q = \{\phi, x[0], x[0..1], \dots, x[0, \dots, m-2], x\}$ .
- $q_0 = \phi$ .
- $T = x$ .
- Với mỗi  $q \in Q$  ( $q$  là một tiền tố của  $x$ ) và  $a \in \Sigma$  thì  $(q, a, qa) \in E$  khi và chỉ khi  $qa$  cũng là tiền tố của  $x$ . Trong trường hợp khác  $(q, a, p) \in E$  chỉ khi  $p$  là hậu tố dài nhất của  $qa$  cũng là một tiền tố của  $x$ .
- Automat hữu hạn  $A(x)$  xây dựng cần không gian nhớ  $O(m\sigma)$  và thời gian  $O(m + \sigma)$ .

## 2.9.6. Thuật toán Shift-Or

### Đặc điểm:

- Sử dụng các toán tử thao tác bit (Bitwise).
- Hiệu quả trong trường hợp độ dài mẫu nhỏ hơn một từ máy.
- Thực hiện pha tiền xử lý với thời gian  $O(m + \sigma)$ ;
- Pha tìm kiếm có độ phức tạp tính toán  $O(n)$ .

### Thuật toán Shift-Or:

#### Input :

- Xâu mẫu  $X = (x_0, x_1, \dots, x_m)$ , độ dài  $m$ .
- Văn bản  $Y = (y_1, y_2, \dots, y_n)$  độ dài  $n$ .

#### Output:

- Đưa ra mọi vị trí xuất hiện của  $X$  trong  $Y$ .

**Formats:**  $k = \text{Shift-Or}(X, m, Y, n)$ ;

#### Actions:

Bước 1 (Tiền xử lý):

PreSo( $X, m, S$ ); //chuyển  $X$  thành tập các số.

Bước 2 (Tìm kiếm):

SO( $X, n, Y, m$ ); //Tìm kiếm mẫu  $X$  trong  $Y$

#### EndActions.



## Bước tiền xử lý: PreSo

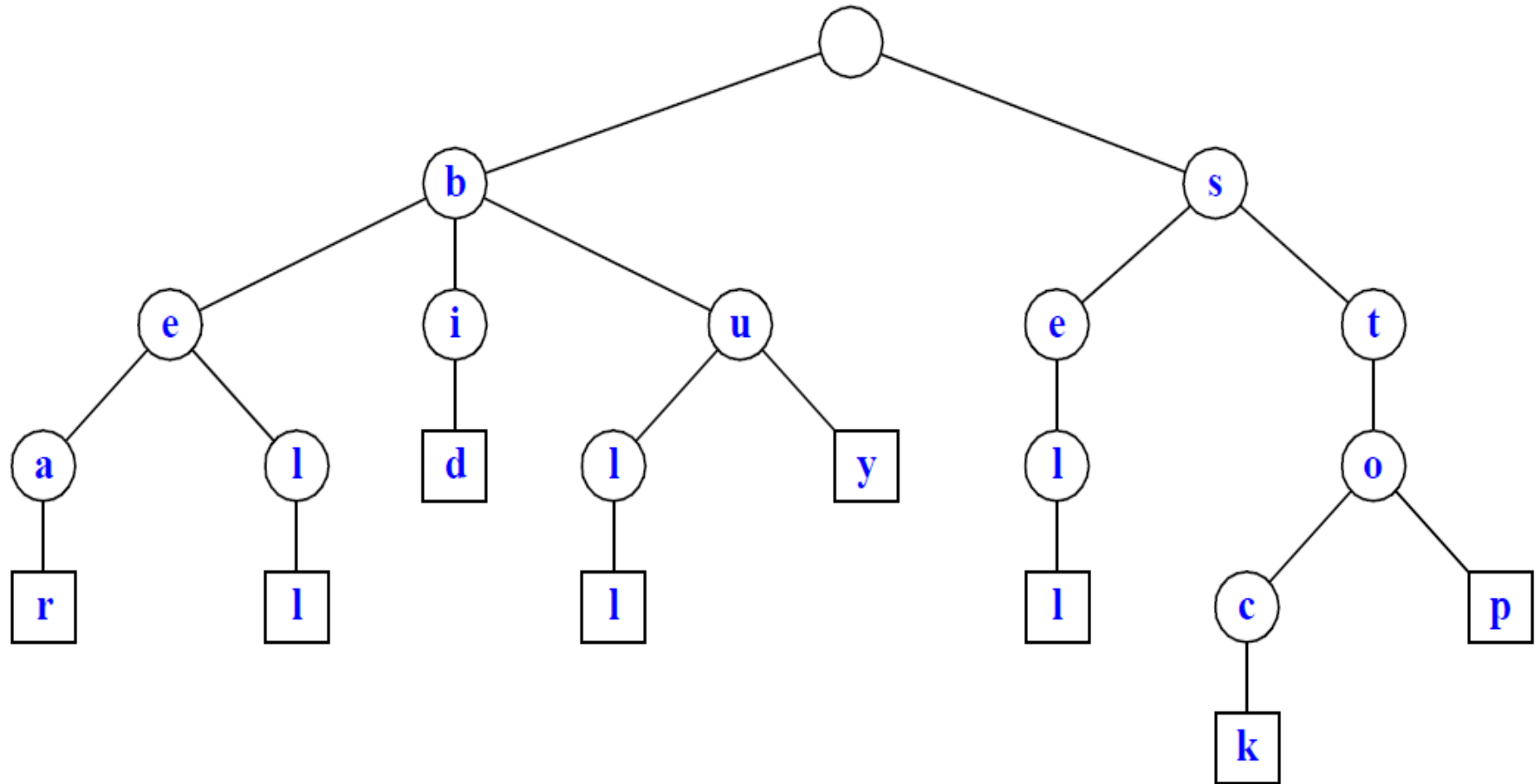
```
int preSo( char  *x, int  m, unsigned int S[]) {  
    unsigned int j, lim;  
    int i;  
    for (i = 0; i < ASIZE; ++i)  
        S[i] = ~0;  
    for (lim = i = 0, j = 1; i < m; ++i, j <=& 1) {  
        S[x[i]] &= ~j;  
        lim |= j;  
    }  
    lim = ~(lim>>1);  
    return(lim);  
}
```

## Bước tìm kiếm: SO

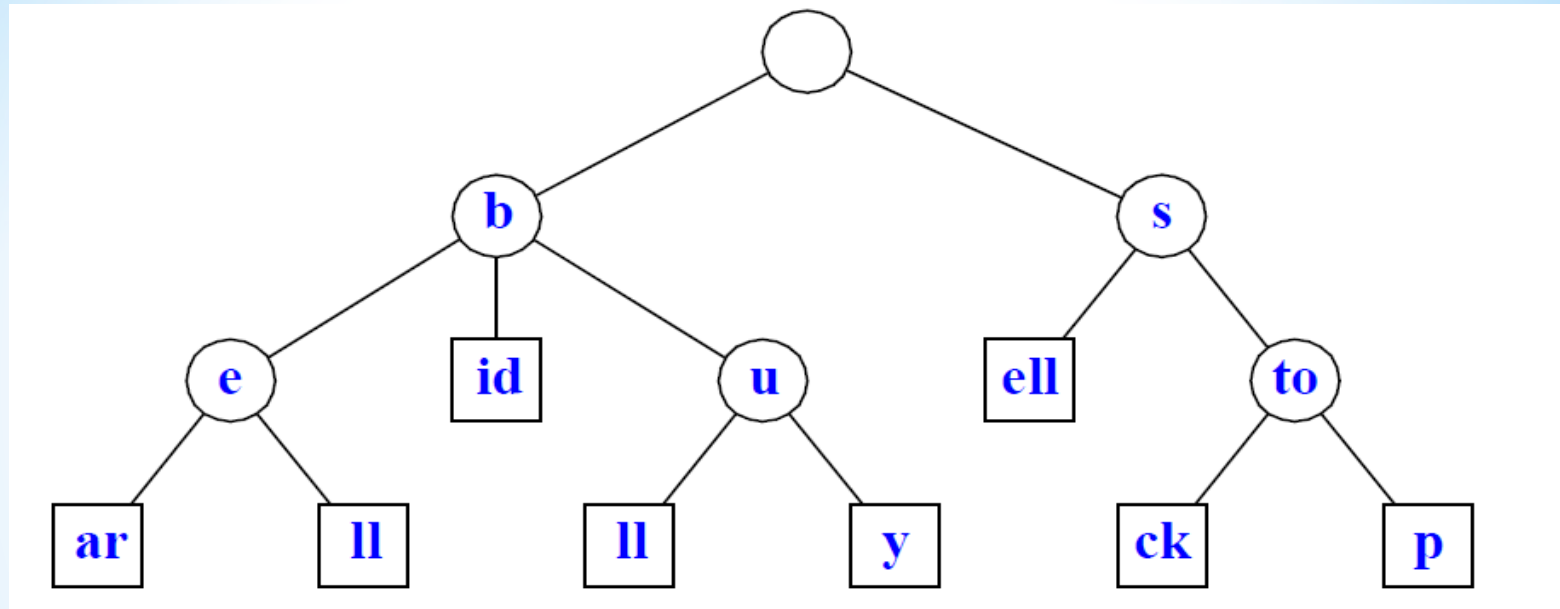
```
void SO(char *x, int m, char *y, int n) {  
    unsigned int lim, state;  
    unsigned int S[ASIZE];  
    int j;  
    if (m > WORD)  
        cout<<("SO: Use pattern size <= word size");  
    /* Preprocessing */  
    lim = preSo(x, m, S);  
    /* Searching */  
    for (state = ~0, j = 0; j < n; ++j) {  
        state = (state<<1) | S[y[j]];  
        if (state < lim)  
            cout<<"Vi tri:"<<(j - m + 1)<<endl;  
    }  
}
```

## 2.9.7. Cây hậu tố

**Cây hậu tố:** Một cây hậu tố của văn bản X là cây được nén cho tất cả các hậu tố của X. Ví dụ  $X = \{\text{bear, bell, bid, bull, buy, sell, stock, stop}\}$ . Khi đó cây hậu tố ban đầu của X được gọi là cây hậu tố chuẩn như dưới đây sau:



**Cây hậu tố nén:** Từ cây hậu tố chuẩn ta thực hiện nối các node đơn lẻ lại với nhau ta được một cây nén (compresses tree).



### Phương pháp tạo cây hậu tố:

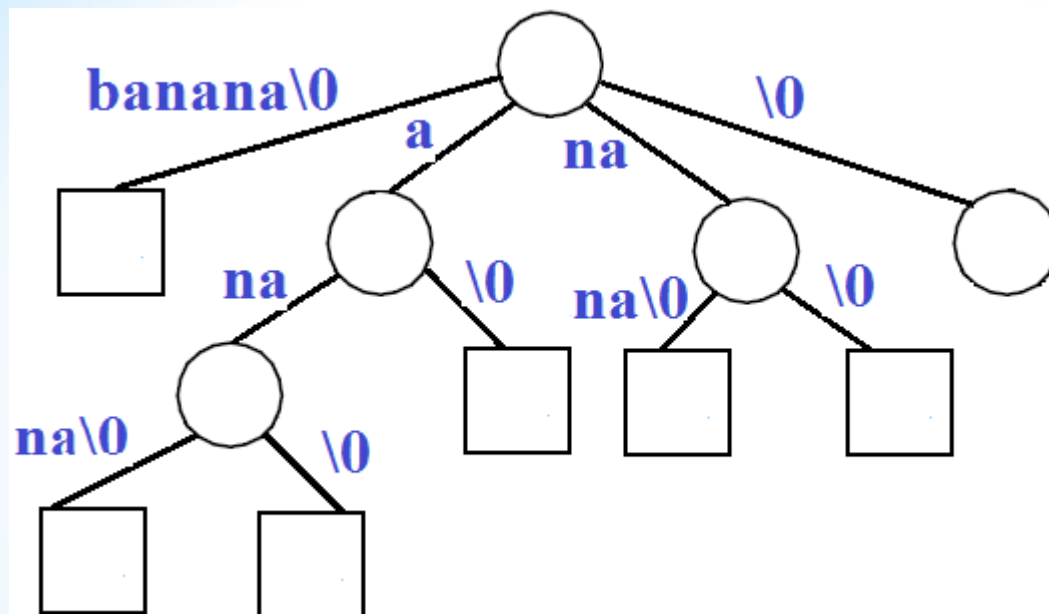
- Bước 1. Sinh ra tất cả các hậu tố của văn bản đã cho.
- Bước 2. Xem xét tất cả các hậu tố như một từ độc lập và xây dựng cây nén.

Ví dụ. Với X = "Banana" ,khi đó thực hiện bước 1 ta nhận được các hậu tố:

Banana;	na;
anana;	a;
nana;	$\phi$
ana;	

[illegible]

## Bước 2. Kết hợp các node đơn để được cây hậu tố



**CASE STUDY:** Biểu diễn, đánh giá và cài đặt những thuật toán tìm kiếm mẫu dưới đây.

- 2.1. Giới thiệu vấn đề
- 2.2. Thuật toán Brute-Force
- 2.3. Thuật toán Knuth-Morris-Partt
- 2.4. Thuật toán Karp-Rabin
- 2.5. Thuật toán Shift or
- 2.6. Thuật toán Morris-Partt
- 2.7. Thuật toán Automat hữu hạn
- 2.8. Thuật toán Simon
- 2.9. Thuật toán Colussi
- 2.10. Thuật toán Galil-Giancarlo
- 2.11. Apostolico-Crochemore algorithm
- 2.12. Not So Naive algorithm
- 2.13. Turbo BM algorithm
- 2.14. Apostolico-Giancarlo algorithm
- 2.15. Reverse Colussi algorithm
- 2.16. Reverse Colussi algorithm
- 2.17. Horspool algorithm
- 2.18. Quick Search algorithm
- 2.19. Tuned Boyer-Moore algorithm
- 2.30. Zhu-Takaoka algorithm
- 2.31. Berry-Ravindran algorithm
- 2.32. Smith algorithm
- 2.33. Raita algorithm
- 2.34. Reverse Factor algorithm
- 2.35. Turbo Reverse Factor algorithm
- 2.36. Forward Dawg Matching algorithm
- 2.37. Backward Nondeterministic Dawg
- 2.38. Matching algorithm
- 2.39. Backward Oracle Matching algorithm
- 2.40 Galil-Seiferas algorithm
- 2.41. Two Way algorithm
- 2.42. String Matching on Ordered
- 2.43. Alphabets algorithm
- 2.44. Optimal Mismatch algorithm
- 2.45. Maximal Shift algorithm
- 2.46. Skip Search algorithm
- 2.47. KMP Skip Search algorithm
- 2.48. Alpha Skip Search algorithm



**Bài tập 1. Bài toán cái túi.** Một nhà thám hiểm cần đem theo một cái túi trọng lượng không quá  $B$ . Có  $N$  đồ vật cần đem theo. Đồ vật thứ  $i$  có trọng lượng  $a_i$ , có giá trị sử dụng  $c_i$  ( $i=1, 2, \dots, N$ ;  $a_i, c_i \in \mathbb{Z}^+$ ). Hãy tìm cách đưa đồ vật vào túi cho nhà thám hiểm sao cho tổng giá trị sử dụng các đồ vật trong túi là lớn nhất

• **Tập phương án của bài toán:** Mỗi phương án của bài toán là một xâu nhị phân có độ dài  $N$ . Trong đó,  $x_i=1$  tương ứng với đồ vật  $i$  được đưa vào túi,  $x_i=0$  tương ứng với đồ vật  $i$  không được đưa vào túi. Tập các xâu nhị phân  $X=(x_1, \dots, x_N)$  còn phải thỏa mãn điều kiện tổng trọng lượng không vượt quá  $B$ . Nói cách khác, tập phương án  $D$  của bài toán được xác định như công thức dưới đây.

$$D = \left\{ X = (x_1, x_2, \dots, x_N) : g(X) = \sum_{i=1}^N a_i x_i \leq B; x_i = 0, 1 \right\}$$

• **Hàm mục tiêu của bài toán:** Ứng với mỗi phương án  $X=(x_1, \dots, x_N) \in D$ , ta cần tìm phương án  $X^*=(x_1^*, \dots, x_N^*)$  sao cho tổng giá trị sử dụng các đồ vật trong túi là lớn nhất. Do vậy, hàm mục tiêu của bài toán được xác định như sau:

$$f(X) = \sum_{i=1}^N c_i x_i \rightarrow \max$$

**Bài tập 2. Bài toán người du lịch.** Một người du lịch muốn đi tham quan  $N$  thành phố  $T_1, T_2, \dots, T_N$ . Xuất phát tại một thành phố nào đó, người du lịch muốn qua tất cả các thành phố còn lại mỗi thành phố đúng một lần rồi trở lại thành phố ban đầu. Biết  $c_{ij}$  là chi phí đi lại từ thành phố  $T_i$  đến thành phố  $T_j$ . Hãy tìm một hành trình cho người đi du lịch có tổng chi phí là nhỏ nhất.

• **Tập phương án của bài toán:** Không hạn chế tính tổng quát của bài toán, ta cố định xuất phát là thành phố  $T_1 = 1$ . Khi đó, mỗi hành trình của người du lịch  $T_1 \rightarrow T_2 \rightarrow \dots \rightarrow T_N \rightarrow T_1$  được xem như một hoán vị của  $2, \dots, N$  là  $X = (x_1, x_2, \dots, x_N)$ , trong đó  $x_1 = 1$ . Như vậy, tập phương án  $D$  của bài toán là tập các hoán vị  $X = (x_1, x_2, \dots, x_N)$  với  $x_1 = 1$ .

$$D = \{X = (x_1, x_2, \dots, x_N) : x_1 = 1 \wedge (\forall i \neq j) : x_i \neq x_j ; i, j = 1, 2, \dots, N\}$$

• **Hàm mục tiêu của bài toán:** Ứng với mỗi phương án  $X = (x_1, \dots, x_N) \in D$ , chi phí đi lại từ thành phố thứ  $i$  đến thành phố  $j$  là  $C[X[i]][X[i+1]]$  ( $i=1, 2, \dots, N-1$ ). Sau đó ta quay lại thành phố ban đầu với chi phí là  $C[X[N]][X[1]]$ . Như vậy, hàm mục tiêu của bài toán được xác định như sau:

$$f(X) = \sum_{i=1}^{N-1} c_{x_i x_{i+1}} + c_{x_N x_1} \rightarrow \min$$

**Bài tập 3. Bài toán cho thuê máy.** Một ông chủ có một chiếc máy cho thuê. Đầu tháng ông nhận được yêu cầu của  $M$  khách hàng thuê máy cho  $N$  ngày kế tiếp. Mỗi khách hàng  $i$  cho biết tập  $N_i$  ngày họ cần thuê máy. Ông chỉ có quyền hoặc từ chối yêu cầu của khách hàng, hoặc nếu chấp nhận yêu cầu của khách ông phải bố trí máy theo đúng những ngày mà khách yêu cầu. Hãy tìm phương án thuê máy giúp ông chủ sao cho tổng số ngày thuê máy là nhiều nhất.

- **Tập phương án của bài toán:** Gọi  $I = \{ 1, 2, \dots, M \}$  là tập chỉ số khách hàng,  $S$  là tập của tất các tập con của  $I$ . Khi đó, tập các phương án cho thuê máy là:

$$D = \left\{ J \subset S : N_k \cap N_p = \emptyset, \forall k, p \in J \right\}$$

- **Hàm mục tiêu:** Ứng với mỗi phương án  $J \in D$ , tổng số ngày cho thuê máy là:

$$f(J) = \sum_{j \in J} |N_j| \rightarrow \max$$

**Bài tập 4. Bài toán phân công công việc.** Một hệ gồm có  $N$  quá trình thực hiện  $N$  việc song hành. Biết mỗi quá trình đều có thể thực hiện được  $N$  việc kể trên nhưng với chi phí thời gian khác nhau. Biết  $c_{ij}$  là thời gian quá trình  $i$  thực hiện việc  $j$ . Hãy tìm phương án giao việc cho mỗi quá trình sao cho tổng thời gian thực hiện  $N$  việc kể trên là ít nhất.

• **Tập phương án của bài toán:** Gọi  $X = (x_1, x_2, \dots, x_N)$  là một hoán vị của  $1, 2, \dots, N$ . Nếu  $x_i = j$  thì ta xem quá trình thứ  $i$  được thực hiện việc  $j$ . Như vậy, tập phương án của bài toán chính là tập các hoán vị của  $1, 2, \dots, N$ .

$$D = \{X = (x_1, x_2, \dots, x_N) : (\forall i \neq j) | x_i \neq x_j, i, j = 1, 2, \dots, N\}$$

• **Hàm mục tiêu của bài toán:** Ứng với mỗi phương án  $X \in D$ , thời gian thực hiện của mỗi phương án là:

$$f(X) = \sum_{i=1}^N c[i, x[i]] \rightarrow \min$$