CLB CÔNG NGHỆ THÔNG TIN
ĐẠI HỌC KINH TẾ QUỐC DÂN

TÀI LIỆU
KHÓA THUẬT TOÁN

(LƯU HÀNH NỘI BỘ)

Người biên soạn: Vũ Thành Công

Tháng 3 - 2024

Lời nói đầu

Lời đầu tiên xin chúc mừng các bạn đã trải qua những khóa học và bài kiểm tra khốc liệt để được lựa chọn trở thành thành viên chính thức của Ban Học thuật - CLB Công nghệ thông tin Đại học Kinh tế Quốc dân NITC.

Sau một thời gian học tập và thử thách bản thân với Coding Challenge, chắc hẳn mọi người đã dần thành thạo hơn với lập trình thi đấu rồi phải không? Vậy thi trong thời gian sắp tới đây, các bạn sẽ tiếp tục được mài giũa khả năng tư duy cũng như nắm vững nhiều dạng bài khó hơn với **Khóa Thuật toán**!

Tài liệu này được chia làm 4 phần, sắp xếp theo thứ tự từ cơ bản đến nâng cao và theo các nhóm dạng bài liên quan đến nhau. Mỗi phần sẽ có các ví dụ bài tập mẫu về thuật toán đó cũng như giải thích lý thuyết. Một số bài tập có dẫn thêm đường link trên phần đề bài để các bạn có thể thực hành và kiểm tra code của bản thân xem đã tối ưu hay chưa…

Tất nhiên, tài liệu này chỉ là một phần rất nhỏ, được góp nhặt và chắt lọc lại các nội dung ở trên Internet về Cấu trúc dữ liệu & Giải thuật. Bởi giải thuật là muôn hình vạn trạng, không có bất kỳ thuật toán nào là tối ưu với mọi tất cả các vấn đề cả. Vì vậy khuyến khích các bạn đối với mỗi bài toán ở trong tài liệu này nên tìm kiếm thêm theo các keyword có trong bài đồng thời mở rộng vấn đề ra xem có cách nào giải tối ưu hơn nữa đối với vấn đề đó không.

Về phía biên soạn, tài liệu này vẫn còn nhiều điểm thiếu sót. Vì vậy mong các Gen tiếp theo của NITC sẽ tiếp tục phát triển và mở rộng thêm nữa các dạng bài hay, khó, mới lạ hơn để chất lượng học viên CLB ngày càng được nâng cao, nhắm đến các giải thưởng top đầu của ICPC/ACM và Olympic Tin học Sinh viên.

Cuối cùng, chúc các bạn sẽ học tập tốt, sẽ không nản chí trên chặng đường giải thuật, và sẽ đạt được thành tích cao trong các cuộc thi sắp tới!

Mục lục

PHẦN 1: CÁC THUẬT TOÁN CƠ BẢN

I. Số học

Bài 1: Greatest Common Prime Divisor

GCPD (Greatest Common Prime Divisor) được định nghĩa là số nguyên tố lớn nhất là ước của các số nguyên dương cho trước. Nhiệm vụ của bạn là tìm GCPD của hai số nguyên a và b.

Ví dụ:

Với a = 12 và b = 18, đầu ra là greatestCommonPrimeDivisor(a, b) = 3;

Với a = 12 và b = 13, đầu ra là greatestCommonPrimeDivisor(a, b) = -1.

- [đầu vào] integer a: $2 \leq a \leq 150$.
- [đầu vào] integer b: $2 \leq b \leq 150$.
- [đầu ra] integer: số GCPD của a và b, hoặc -1 nếu nó không tồn tại.

Lý thuyết:

Sàng nguyên tố Eratosthenes:

- Tư tưởng của phương pháp trên là ta sẽ tìm mọi số nguyên tố có giá trị bé hơn hoặc bằng n. Từ đó có thể kết luật được số n có phải là một số nguyên tố hay không ?
- Thuật toán được miêu tả như sau:
    - Lập một danh sách các số liên tiếp từ 2 đến n
    - Bước đầu tiên ta đặt số a = 2
    - Lần lượt đánh dấu các số a*a, a*(a+1), a*(a+2), ... có trong danh sách mình đã tạo trước. Nếu như a*a > n thì ta kết thúc thuật toán.
    - Tìm số đầu tiên lớn hơn a chưa được đánh dấu trong danh sách. Nếu không tìm thấy thì kết thúc thuật toán, ngược lại thì gán a bằng số đó và lặp lại bước 3
- Ví dụ minh họa:

Một danh sách các số từ 2 đến 20:

	2	3	4	5
6	7	8	9	10
11	12	13	14	15
16	17	18	19	20

Ta gán a = 2 và đánh dấu các số 2*2, 2*3, 2*4, ..., 2*10. Ta được bảng:

	2	3	**4**	5
6	7	**8**	9	**10**
11	**12**	13	**14**	15
16	17	**18**	19	**20**

Tiếp tục tìm số lớn hơn 3 mà chưa được đánh dấu trong danh sách và gán a bằng số đó ⇒ a = 5. Tiếp tục đánh dấu các số chia hết cho 5 mà lớn hơn 5*5 = 25. Không tồn tại số đó trong danh sách. Ta dừng lại quá trình làm của mình ở đây. Cuối cùng bảng ta thu được sẽ là:

	2	3	**4**	5
6	7	**8**	**9**	**10**
11	**12**	13	**14**	**15**
16	17	**18**	19	**20**

Hướng dẫn:

```
bool arr[1000001];
void snt(int n){
    for (int i = 2; i <= n; i++)
        arr[i] = 1;
    arr[0] = arr[1] = 0;
    for (int i = 2; i <= sqrt(n); i++)
    if (arr[i])
        for (int j = 2 * i; j <= n; j += i)
            arr[j] = 0;
}
int greatestCommonPrimeDivisor(int a, int b)
{
    snt(min(a, b));
    int d = 0;
    for (int i = min(a, b); i >= 2; i--)
        if (arr[i] && a % i == 0 && b % i == 0)
            return i;
    return -1;
}
```

Bài 2: Tìm chữ số khác không cuối cùng của n!(giai thừa)

Ví dụ:

Với n = 5, kết quả lastDigitDiffZero(n) = 2.

5! = 1 · 2 · 3 · 4 · 5 = 120.

Với n = 6, kết quả lastDigitDiffZero(n) = 2.

6! =1 · 2 · 3 · 4 · 5 · 6 = 720.

Với n = 10, kết quả lastDigitDiffZero(n) = 8.

10! = 3628800.

- [input] integer n: $1 \leq n \leq 10^6$.
- [output] integer: Chữ số cuối cùng khác 0 của n!

Lý thuyết:

- Một số X như thế nào thì chia hết cho 10 ? Đó là khi X phân tích ra thừa số nguyên tố, sẽ luôn chứa 2 thừa số là 2 và 5 (Vì 10 = 2 * 5, mà 2 và 5 nguyên tố cùng nhau)
- Một số X có k chữ số 0 tận cùng, tức là X chia hết cho 10k, và không chia hết cho 10k+1. Khi đó, bậc lũy thừa của 2 và 5, giá trị nhỏ hơn sẽ phải đúng bằng k (10k = 2k * 5k)
- Vậy ta cần xác định số chữ số 0 của số X, thì chỉ cần xác định bậc lũy thừa của 2 và 5 mà X có, rồi lấy ra số nhỏ hơn
- Ví dụ : X = 27 * 3 * 55 ⇒ X có 5 chữ số 0 tận cùng. Điều đó đúng, vì giá trị X = 1200000
- Chúng ta đã tìm hiểu được cách đếm số chữ số 0 tận cùng của 1 số. Vậy muốn tìm chữ số tận cùng khác 0 đầu tiên của 1 số, thì làm như thế nào ?
- Đơn giản lắm, chúng ta chỉ cần xác định được số chữ số 0 tận cùng của X. Nếu X chia hết cho 10k, thì bạn chỉ việc chia X cho 10k là sẽ tìm ra thôi :)
- Các bạn sẽ thắc mắc làm sao xác định được số chữ số 0 tận cùng của X nếu như X quá lớn (ví dụ : X = n!, X = nk với n, k đều là giá trị lớn ...)
- Thực ra thì, những số X này đều được cấu thành từ những giá trị đủ để ta tính toán. Nên thay vì gộp chung chúng lại rồi tìm 1 lần, các bạn có thể tìm bậc lũy thừa của 2 và 5 cho từng số, rồi cộng tổng lại
- Tính đúng đắn của nó, kế thừa từ định lý toán cơ bản : khi nhân 2 lũy thừa cùng cơ số, được 1 lũy thừa mới có bậc bằng tổng bậc 2 lũy thừa kia

Code đếm số 0 trong giai thừa của n:

```
int NumOf0s(vector  a, int n) {
    int sum2 = 0, sum5 = 0;
    for (int i = 0; i < n; ++i) {
        int num2 = 0, num5 = 0;
        while (a[i] % 2 == 0) {
            num2++;
            a[i] /= 2;
        }
        while (a[i] % 5 == 0) {
            num5++;
            a[i] /= 5;
        }
        sum2 += num2, sum5 += num5;
    }
    return min(sum2, sum5);
}
```

Hướng dẫn:

```
int lastDigitDiffZero(int n)
{
    long long res = 1;
    for (int i=2;i<=n;i++)
    {
        res *=i;
        while ( res%10 == 0 ) res /= 10;
        res = res % 100;
    }
    while ( res%10 == 0 ) res /= 10;
    return res% 10;
}
```

Bài 3: Cho số tự nhiên product. Hãy tìm số nguyên dương nhỏ nhất (lớn hơn 0) mà tích các chữ số của số đó bằng product. Nếu không có số thỏa mãn, trả ra -1.

Ví dụ:

Với product = 12, thì kết quả digitsProduct(product) = 26;

Với product = 19, thì kết quả digitsProduct(product) = -1.

- [Đầu vào] integer product: $0 \leq$ product ≤ 600.
- [Đầu ra] integer

Lý thuyết:

Số A nhỏ hơn số B khi :

- số chữ số của A ít hơn số chữ số của B
- số chữ số của A bằng của B, và vị trí đầu tiên (từ trái sang) khác nhau của 2 số, chữ số của A có giá trị bé hơn

Vậy, để thu được số nhỏ nhất, ta cần tối ưu về số lượng chữ số, rồi sau đó là thứ tự các chữ số.

Ta lấy theo cách sau : lấy chữ số to nhất lớn hơn 1 mà product chia hết, đặt ở cuối, rồi tiếp tục làm thế với product mới (sau khi đã chia cho chữ số đó)

- TH1 : product = 1 ⇒ Ta đã tách được product thành mọi chữ số, nên sẽ có kết quả cho bài toán
- TH2 : product > 1. Lúc này product sẽ gồm các số nguyên tố > 9, nên ko còn cách nào có thể tách ra nữa. Ko có số nào thỏa mãn đề bài trong trường hợp này

Tính đúng đắn :

- Giá trị ta tách ra càng to, thì càng tách ra ít lần. Nên số chữ số chắc chắn là tối ưu nhất
- Giả sử có nhiều cách tách ra có cùng số chữ số, thì việc tách ra chữ số càng to, sẽ càng tốt hơn cho phần ở trước

Ví dụ : product = 108 ⇒ 9 * 6 * 2 ⇒ kết quả : 269

- Ta nên lấy { 9, 6, 2 } thay vì { 2, 2, 3, 3, 3 } để tối ưu về số chữ số
- Ta nên lấy { 9, 6, 2 } thay vì { 9, 4, 3 } vì sau khi lấy 6, product còn lại là 2 thay vì 3. Và 2 tốt hơn khi đặt ở hàng đầu tiên (từ trái sang)

Bài toán tương tự : Cho số product. Tìm số có tổng các chữ số nhỏ nhất. Nếu có nhiều số thỏa mãn, in ra số bé nhất.

Hướng dẫn:

```
int digitsProduct(int product)
{
    if (product == 1) return 1;
    if (product == 0) return 10;
    int ans = 0;
    for (int i = 9; i >= 2; i--){
        while(product % i == 0){
            ans = ans * 10 + i;
            product /= i;
        }
    }
    // số ans bây giờ là kết quả, nhưng nó đang bị
    int ans2 = 0;
    while (ans > 0){
        ans2 = ans2 * 10 + ans % 10;
        ans /= 10;
    }
    return (product == 1) ? ans2 : -1;
}
```

Bài 4: Cho số nguyên n, hãy tính tổng các số nguyên tố nhỏ hơn hoặc bằng n

(Bởi vì tổng các số nguyên tố nhỏ hơn hoặc bằng n có thể rất lớn, nên hãy trả ra kết quả của tổng trên dưới dạng là số dư trong phép chia của tổng trên cho 22082018)

Lý thuyết:

- Kiểm tra và tính tổng các số nguyên tố từ 1 tới n có thể tốn nhiều thời gian
- Để tăng tốc tìm các số nguyên tố, có thể sử dụng sàng Eratosthenes

Hướng dẫn:

```
bool a[10000001];
void snt(int n){
    for (int i = 2; i <= n; i++)
        a[i]=1;
    a[0] = a[1] = 0;
    for (int i = 2; i <= sqrt(n); i++)
    if (a[i])
        for (int j = 2 * i; j <= n; j += i)
            a[j] = 0;
}
int primeSum(int n)
{
    int mod = 22082018;
    snt(n);
    int d = 0;
    for (int i = 2; i <= n; i++)
        if (a[i]) d = (d + i % mod) % mod;
    return d;
}
```

II. Xử lý xâu

Bài 1: Một xâu được gọi là palindrome nếu viết xuôi hay viết ngược xâu đó đều cho ra kết quả giống nhau. Cho một xâu kí tự, kiểm tra nó có phải xâu palindrome không.

Ví dụ:

Với inputString = "aabaa", kết quả checkPalindrome(inputString) = true;

Với inputString = "abac", kết quả checkPalindrome(inputString) = false;

Với inputString = "a", kết quả checkPalindrome(inputString) = true.

- [Đầu vào] string inputString: Xâu không rỗng chứa các kí tự chữ cái in thường, 1 ≤ inputString.length ≤ 105.
- [Đầu ra] boolean: true nếu inputString là xâu palindrome, false nếu không.

Lý thuyết:

Để kiểm tra 1 xâu S độ dài n có là xâu palindrome không, thì ta chỉ việc kiểm tra xâu S có đối xứng không là được, tức là S[i] = S[n - i - 1] với i = [0 .. n/2). ĐPT : O(N)

Hướng dẫn:

```
bool isPalindrome(string S)
{
    int n = S.size();
    for (int i = 0; i < n/2; ++i)
        if (S[i] != S[n - i - 1]) return false;
    return true;
}
```

Mở rộng vấn đề:

Nếu như ta muốn kiểm tra 1 lượng lớn xâu con của S có phải là xâu palindrome không, vậy có cách nào để kiểm tra nhanh không ?

Nhận xét : với xâu S, nếu xâu con S[L, R] là xâu palin

TH1 : L = R. Điều này hiển nhiên phải không các bạn ?

TH2 : L < R

kí tự S[L] = S[R]

hoặc L = R - 1, hoặc S[L + 1, R - 1] cũng phải là xâu palin

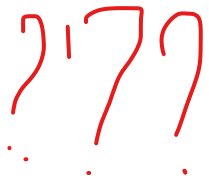
Từ nhận xét trên, ta có thể dùng 1 mảng CheckPalin[][] để lưu lại kết quả khi kiểm tra với các xâu con của S. Ta sẽ kiểm tra CheckPalin[i][j] dựa vào S[i], S[j] và CheckPalin[i + 1][j - 1]

Khi cần kiểm tra 1 đoạn [L, R] có là xâu palin không, ta chỉ việc lấy ra CheckPalin[L][R] là được. 1 bước, quá nhanh gọn.

```
int CheckPalin[1003][1003];

void buildCheck(string S) {
    int n = S.size();
    for (int i = n - 1; i >= 0; --i) {
        CheckPalin[i][i] = 1; // day la TH1
        for (int j = i + 1; j < n; ++j)
            CheckPalin[i][j] = (S[i] == S[j]) && (
            // day la TH2
    }
}

bool checkSubS(int L, int R) {
    return CheckPalin[L][R];
}
```



Bài 2: Cho một xâu kí tự chứa các chữ số từ 0 tới 9. Người ta áp dụng phép biến đổi xâu dựa trên các nguyên tắc sau:

- **Nếu chữ số ngoài cùng bên trái chia hết cho 3, xóa nó khỏi xâu kí tự**
- **Nếu không thỏa mãn điều kiện trên, và nếu chữ số ngoài cùng bên tay phải chia hết cho 3, xóa nó khỏi xâu kí tự**

- **Nếu không thỏa mãn 2 điều kiện trên, và nếu tổng chữ số ngoài cùng bên trái và ngoài cùng bên phải chia hết cho 3, xóa cả hai chữ số trên khỏi xâu**

Các phép toán trên được áp dụng vào xâu ban đầu cho tới khi xâu trở thành rỗng, hoặc ko đáp ứng cả 3 điều kiện trên

Cho một xâu kí tự, hãy tìm xâu kết quả cuối cùng khi áp dụng liên tục các phép toán trên.

Ví dụ:

Với s = "312248", thì kết quả truncateString(s) = "2".

Các bước thực hiện để ra kết quả trên như sau:

Kí tự đầu tiên bên trái 3 chia hết cho 3 và bị xóa. Xâu s trở thành "12248";

Cả 1 và 8 đều không chia hết cho 3, nhưng tổng của chúng là 9 thì chia hết cho 3, do đó cả 1 và 8 bị xóa khỏi xâu. Xâu s trở thành "224";

Cả 2 và 4 đều không chia hết cho 3, nhưng tổng của chúng là 6 thì chia hết cho 3, do đó cả 2 và 4 bị xóa khỏi xâu. Xâu s trở thành "2";

Xâu "2" không thỏa mãn điều kiện nào trong các điều kiện trên, do đó đây là xâu đáp án cuối cùng.

- [Đầu vào] string s: $1 \leq s.length \leq 15$.
- [Đầu ra] string.

Hướng dẫn:

```
std::string truncateString(std::string s)
{
    bool kt = true;
    while (kt && s.length() > 0){
        kt = false;
        if ((s[0] - '0') % 3 == 0){
            s.erase(0,1);
            kt = true;
        } else
            if ((s[s.length() - 1] - '0') % 3 == 0){
                s.erase(s.length()-1,1);
                kt = true;
            } else
                if ((s[0]+s[s.length() - 1] - 96) % 3 == 0){
                    kt = true;
                    s.erase(s.length() - 1, 1);
                    s.erase(0, 1);
                }
    }
    return s;
}
```

Bài 3: Một hệ thống hỏi đáp trực tuyến cần chuẩn hóa câu hỏi của người dùng

Một câu hỏi đã được chuẩn hóa cần tuân thủ các luật sau:

- Luật chung: Câu hỏi chỉ chứa kí tự chữ cái (a-zA-Z), chữ số (0-9), dấu phẩy (,), dấu cách (' '), dấu hỏi (?). Các kí tự khác đều được thay thế bằng dấu cách

- Luật dấu cách: Không có dấu cách ở đầu hay ở cuối câu. Giữa các từ chỉ có 1 dấu cách duy nhất. Sau mỗi dấu cách là 1 chữ cái hoặc chữ số?

- Luật dấu phẩy: Trước dấu phẩy luôn là 1 chữ cái hoặc chữ số. Sau dấu phẩy luôn là một dấu cách. Trường hợp đứng trước dấu phẩy là dấu cách, hãy xóa dấu cách này đi.

- Luật chữ hoa/chữ thường: Chữ cái bắt đầu câu luôn được viết hoa. Các chữ cái khác đều viết thường

- Luật dấu hỏi: Luôn có 1 dấu ? kết thúc câu. Trước dấu ? luôn là kí tự chữ cái hoặc chữ số. Trường hợp có các dấu ? xuất hiện khi chưa kết thúc câu, hãy thay thế nó bằng dấu cách. Trường hợp trước dấu cách là dấu phẩy và dấu cách, hãy xóa dấu cách và dấu phẩy

Ví dụ:

Với đầu vào s="this is not a relevant question , is it???", thì kết quả questionCorrection(s) = "This is not a relevant question, is it?"

Với đầu vào s="who are you,,???", thì kết quả questionCorrection(s) = "Who are you?"

Hướng dẫn:

```
string q(string s){
    string v="";
    for (int i = 0; i<s.size(); i++){
        if (s[i]>='A'&&s[i]<='Z')v = v+char(s[i]+32);
        else if(s[i]>='a'&&s[i]<='z') v= v+s[i];
        else if (s[i]>='0' && s[i]<= '9') v= v+s[i];
        else if (s[i]=='?') v = v+ " ";
        else if (s[i]==',' &&s[i+1]==','){
            while(s[i]==',')i++;
            v = v+ ",";
            i--;
        }
        else if( s[i]==',') v= v+", ";
        else if (s[i]==' '&&s[i+1]==','){
            v = v+",";
            i++;
        }
        else if (s[i]==' '){
            while(s[i]==' ')i++;
            v = v+ " ";
            i--;
        }
        else v=v+' ';

    }
    while (v[0] == ' ' || v[0]==',')v.erase(0, 1);
    while (v.find("  ") != -1)v.erase(v.find("  "), 1);
    while (v[v.length()-1]==' '||v[v.length()-1]==','){
        v.erase(v.length()-1, 1);
    }
    if (v[0]<'0'||v[0]>'9')v[0]=v[0]-32;
    v = v+"?";
    return v;
}
string questionCorrection(string s){
    string a = q(s);
    string b = q(a);
    return q(b);
}
```

Bài 4: Cho một xâu s gồm các ký tự từ ‘a’ -> ‘z’ in thường . Tìm các ký tự bị trùng lặp trong xâu đó.

Ví dụ:

Với đầu vào s="findduplicates", thì kết quả findDuplicates(s) = " d i"

- [Đầu vào] string s.
- [Đầu ra] string: gồm các ký tự bị trùng lặp trong xâu. Mỗi ký tự ngăn cách bằng 1 khoảng trắng.

Lý thuyết:

Với bài toàn này sẽ có nhiều cách giải khác nhau. Tuy nhiên trong hướng dẫn này, chúng ta sẽ sử dụng toán tử bitwise để tìm ký tự trùng lặp trong một xâu cho trước.

Lưu ý: Cách này chỉ sử dụng được đối với xâu gồm các ký tự từ ‘a’ – ‘z’. Đối với các xâu có dải ký tự rộng hơn, nó sẽ làm đưa ra kết quả sai.

Ta sử dụng một biến **checker** để theo dõi những ký tự nào đã trùng lặp. Đối với mỗi ký tự trong chuỗi đầu vào, nó tính toán bit tương ứng bằng cách sử dụng giá trị ASCII của ký tự đó (ký tự được trừ cho ‘a’ để lấy chỉ số từ 0 đến 25 theo thứ tự của chữ cái đó trong bảng chữ cái). Nếu bit đã được đặt trong **checker**, điều đó có nghĩa là ký tự đó bị trùng lặp và nó được in. Nếu không, bit sẽ được đặt vào **checker**.

Tìm hiểu thêm trên internet với keyword: “Finding Duplicates in a String using Bitwise Operations”

Hướng dẫn giải:

```
void findDuplicates(const std::string& str) {
    int checker = 0;

    for (char ch : str) {
        int charValue = ch - 'a';
        int charBit = 1 << charValue;

        // Check if the bit corresponding to the character is already set
        if ((checker & charBit) > 0) {
            std::cout << ch << " ";
        } else {
            // Set the bit corresponding to the character
            checker |= charBit;
        }
    }
}
```

III. Đếm

Bài 1: Cho một xâu kí tự, tìm số lượng xâu con khác nhau của xâu đó (không tính xâu rỗng)

Ví dụ:

Với inputString = "abac", thì kết quả differentSubstringsTrie(inputString) = 9.

9 xâu con khác nhau của xâu đầu vào là:

"a", "b", "c", "ab", "ac", "ba", "aba", "bac", "abac"

- [Đầu vào] string inputString: Xâu kí tự chỉ chứa kí tự chữ cái in thường, 3 ≤ inputString.length ≤ 15.
- [Đầu ra] integer

Hướng dẫn:

```
int differentSubstringsTrie(std::string inputString)
{
    int d = 0;
    string  s = inputString;
    string p = " ";
    for (int i = 0; i < s.length(); i++){
        string h = "";
        for (int j = i; j < s.length(); j++){
            h = h + s[j];
            string k = " " + h + " ";
            if (p.find(k) < 0 || p.find(k) > p.length() - 1) {
                d++;
                p = p + h + " ";
            }
        }
    }
    return d;
}
```

Bài 2: Cho một ma trận chữ nhật chứa các chữ số (0-9)

Hãy tính số lượng các hình vuông 2 × 2 khác nhau tồn tại trong ma trận.

Ví dụ:

Với matrix =

[[1, 2, 1],

[2, 2, 2],

[2, 2, 2],

[1, 2, 3],

[2, 2, 1]]

thì kết quả differentSquares(matrix) = 6.

Dưới đây là 6 hình vuông 2 × 2 khác nhau:

- 1 2

  2 2
- 2 1

  2 2
- 2 2

  2 2
- 2 2

  1 2
- 2 2

  2 3
- 2 3

  2 1

- [Đầu vào] array.array.integer matrix: 1 ≤ matrix.length ≤ 100, 1 ≤ matrix[i].length ≤ 100, 0 ≤ matrix[i][j] ≤ 9
- [Đầu ra] integer: Số các hình vuông 2 × 2 khác nhau tồn tại trong ma trận.

Hướng dẫn:

```
int differentSquares(std::vector<std::vector<int>> matrix)
{
    string s = "@";
    string p = "";
    int d = 0;
    for (int i = 0; i < matrix.size() - 1; i++)
        for (int j = 0; j < matrix[0].size() - 1; j++)
        {
            p = "";
            p = p + "@" + char(matrix[i][j] + 48) + " " + char(matrix[i][j + 1] + 48)
            + " " + char(matrix[i + 1][j] + 48) + " " + char(matrix[i + 1][j + 1] + 48) + "@ ";
            if (s.find(p) < 0 || s.find(p) > s.length() - 1)
            {
                d++;
                s = s + p;
            }
        }
    return d;
}
```

IV. Dãy số

Bài 1: Cho một mảng gồm các số tự nhiên đã sắp xếp tăng dần. Nhập vào một số nguyên bất kỳ. Kiểm tra xem số nguyên đó có nằm trong dãy số đã cho hay không?

Ví dụ:

Mảng arr = {2, 3, 4, 10, 40}, x = 10; Kết quả trả về là "true"

Mảng arr = {2, 3, 4, 10, 40}, x = 15; Kết quả trả về là "false"

Lý thuyết:

- **Tìm kiếm nhị phân** là một thuật toán tìm kiếm được sử dụng trong một mảng đã được sắp xếp bằng cách chia đôi mảng cần tìm kiếm nhiều lần.
- Chúng ta chia đôi mảng và gọi 2 phần chia đôi đó là left và right
- Phần tử đứng ở giữa left và Right được gọi là Mid.

- Chúng ta sẽ dựa vào Mid để tìm xem giá trị chúng ta cần tìm nó nằm trên mảng left hay right
- Nếu giá trị cần tìm nằm ở trên left thì chúng ta sẽ loại bỏ mảng right và chỉ thực hiện tìm kiếm trên left và ngược lại!
- Độ phức tạp trung bình: O(log N)

Hướng dẫn:

```
int binarySearch(int arr[], int l, int r, int x)
{
    if (r >= l) {
        int mid = l + (r - l) / 2;

        // If the element is present at the middle
        // itself
        if (arr[mid] == x)
            return mid;

        // If element is smaller than mid, then
        // it can only be present in left subarray
        if (arr[mid] > x)
            return binarySearch(arr, l, mid - 1, x);

        // Else the element can only be present
        // in right subarray
        return binarySearch(arr, mid + 1, r, x);
    }

    // We reach here when element is not
    // present in array
    return -1;
}

int main(void)
{
    int arr[] = { 2, 3, 4, 10, 40 };
    int x = 10;
    int n = sizeof(arr) / sizeof(arr[0]);
    int result = binarySearch(arr, 0, n - 1, x);
    (result == -1) ? cout << "false"
                   : cout << "true";
    return 0;
}
```

Bài 2: Dãy con có tổng lớn nhất

Cho dãy gồm n số nguyên a1, a2, …, an. Tìm dãy con gồm một hoặc một số phần tử liên tiếp của dãy đã cho với tổng các phần tử trong dãy là lớn nhất.

Ví dụ:

Mảng arr = { 6, -8, -2, 5, 7, -3, 10}, Kết quả là 3 6 19

Mảng arr = { -1, 3, 8, -3, -2, 5, 0} , Kết quả là 2 3 11

- [Đầu vào] int: Dòng đầu tiền chứa số nguyên dương n (n < 106). Dòng thứ i trong số n dòng tiếp theo chứa số ai (|ai| <= 1000).
- [Đầu ra] int: gồm 3 số nguyên là vị trí đầu tiên, vị trí cuối cùng và tổng của đoạn con có tổng lớn nhất trong dãy số.

Lý thuyết:

Để giải bài toán này, ta sử dụng thuật toán **Kadane**.

Thuật toán Kadane là một thuật toán cực kỳ đơn giản và hiệu quả để tìm dãy con liên tiếp có tổng lớn nhất trong một dãy số nguyên cho trước. Thuật toán này được đặt theo tên của nhà khoa học người Ấn Độ – Subramanian Kadane.

Ý tưởng của thuật toán là duyệt qua các phần tử của dãy một cách tuần tự, với mỗi phần tử ta tính tổng của dãy con liên tiếp kết thúc bằng phần tử đó. Nếu tổng này lớn hơn tổng lớn nhất đã biết trước đó thì ta cập nhật tổng lớn nhất và vị trí của dãy con liên tiếp đó.

Cụ thể, ta duyệt qua dãy số một cách tuần tự và sử dụng hai biến để lưu trữ thông tin về dãy con liên tiếp có tổng lớn nhất. Biến **max_sum** lưu trữ tổng lớn nhất đã tìm được cho đến thời điểm hiện tại, còn biến **cur_sum** lưu trữ tổng của dãy con liên tiếp kết thúc bằng phần tử hiện tại. Ta cũng sử dụng hai biến **start_idx** và **end_idx** để lưu trữ vị trí bắt đầu và kết thúc của dãy con liên tiếp tương ứng.

Thuật toán Kadane có độ phức tạp thời gian là O(n), nghĩa là nó có thể tìm ra dãy con liên tiếp có tổng lớn nhất trong thời gian tuyến tính với số phần tử của dãy.

Hướng dẫn:

```cpp
int main() {
    int n;
    cin >> n;
    vector<int> a(n);
    for (int i = 0; i < n; i++) {
        cin >> a[i];
    }

    int max_sum = a[0];
    int cur_sum = a[0];
    int start_idx = 0;
    int end_idx = 0;
    int cur_start_idx = 0;

    for (int i = 1; i < n; i++) {
        if (cur_sum < 0) {
            cur_sum = a[i];
            cur_start_idx = i;
        } else {
            cur_sum += a[i];
        }

        if (cur_sum > max_sum) {
            max_sum = cur_sum;
            start_idx = cur_start_idx;
            end_idx = i;
        }
    }

    cout << start_idx + 1 << " " << end_idx + 1 << " " << max_sum;

    return 0;
}
```

Bài 3: Cho một mảng các số nguyên

Hãy tìm ra dãy con có độ dài lớn nhất và tạo thành cấp số cộng.

Ví dụ:

Với a = [1, 7, 3, 5, 4, 2], thì kết quả longestSequence(a) = 3.

Dãy [1, 3, 5] là dãy con không liên tiếp (giữ nguyên thứ tự xuất hiện) có độ dài lớn nhất tạo thành 1 cấp số cộng

- [Đầu vào] array.integer a: Mảng chứa số nguyên: $1 \leq a.length \leq 20$, $-250 \leq a[i] \leq 250$.

- [Đầu ra] integer: Độ dài lớn nhất của dãy con tạo thành cấp số cộng

Hướng dẫn:

```
int find(vector<int> a, int i, int j)
{
    int k = a[j] - a[i], count = 2;
    for (int x = j + 1; x < a.size(); x++)
        if (a[x] - a[i] == k * count)
        {
            count++;
        }
    return count;
}
int longestSequence(vector<int> a)
{
    int max = INT_MIN;
    for (int i = 0; i < a.size(); i++)
    {
        for (int j = i + 1; j < a.size(); j++)

        {
            if (max < find(a, i, j))
                max = find(a, i, j);
        }
    }
    return max;
}
```

Bài 4: [*https://cses.fi/problemset/task/1070*] Hoán vị của các số nguyên 1,2,.. N được gọi là *đẹp* nếu không có phần tử liền kề nào cách nhau 1 đơn vị.

Cho số nguyên N, hãy xây dựng một hoán vị *đẹp* nếu tồn tại một hoán vị như vậy.

Ví dụ:

Nếu n = 5 ta có kết quả trả về là {4, 2, 5, 3, 1}

Nếu n = 3 ta có kết quả trả về là “NO SOLUTION”

- [Đầu vào] Dòng đầu vào duy nhất chứa số nguyên N, 1 <= N <= 10^6
- [Đầu ra] In một hoán vị đẹp các số 1, 2,…N. Nếu có nhiều kết quả, bạn có thể in bất kỳ kết quả nào trong số đó**.** Nếu không có giải pháp nào thì in ra "NO SOLUTION".

Gợi ý: Tìm ra quy luật để sắp xếp các số chẵn lẻ sao cho phù hợp.

V. Ma trận

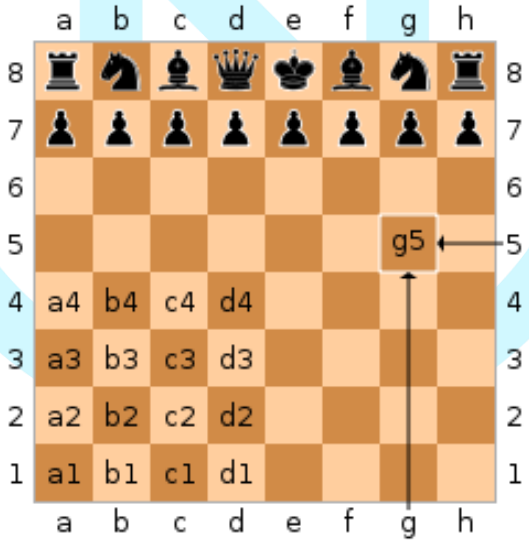
Bài 1: Bàn cờ vua là một bảng có 8*8 ô vuông

Mỗi ô trên bàn cờ được kí hiệu bằng 2 kí tự - 1 kí tự chữ cái và 1 kí tự số

Các cột hàng dọc được gán nhãn từ trái sang phải bằng các kí tự chữ cái từ 'a' tới 'h'

Các hàng ngang được đánh số từ 1 tới 8 từ phía dưới lên trên

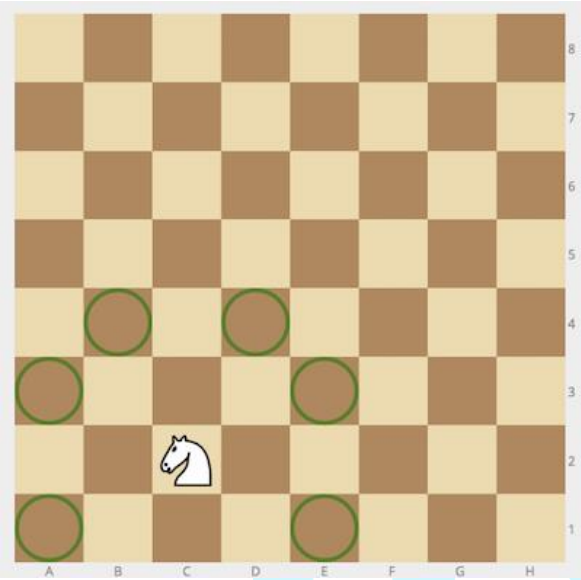
Vị trí mỗi một ô trên bàn cờ được thể hiện bằng xâu có 2 kí tự: kí tự đầu tiên thể hiện cột, kí tự thứ hai thể hiện hàng. Ví dụ như a8, b3, c2, ...



Cho biết vị trí của con mã trên bàn cờ vua, hãy tìm số vị trí khác nhau mà con mã có thể thực hiện nước nhảy

Một con mã có thể nhảy tới 1 vị trí cách vị trí hiện tại 2 ô theo chiều ngang và 1 ô theo chiều dọc, hoặc 2 ô theo chiều dọc và 1 ô theo chiều ngang. Nước đi của con mã tạo ra hình dáng giống như 1 chữ L

Ví dụ: Với cell = "c2", thì kết quả chessKnight(cell) = 6.



- [Đầu vào] string cell: Vị trí hiện tại của con mã, cell.length = 2, 'a' ≤ cell[0] ≤ 'h', 1 ≤ cell[1] ≤ 8.
- [Đầu ra] integer: Số nước đi mà con mã có thể thực hiện

Lý thuyết:

Cách để duyệt các tọa độ sắp truy cập từ ô hiện tại (được áp dụng rất nhiều sau này, khi các bạn làm đến những bài có BFS trên bảng) : Dùng 2 bảng lưu độ chênh về giá trị hàng, cột của ô sắp đến so với ô hiện tại. Rồi for và duyệt, xử lí chung với các ô tọa độ mới thay vì xử lí tay với từng ô.

Hướng dẫn:

```
int chessKnight(string cell)
{
    int di2[] = {2, -2}, di1[] = {1, -1}, dem = 0, x = cell[0] - 96, y = cell[1] - 48;

    for (int i = 0; i <= 1; i++)
        for (int j = 0; j <= 1; j++)
        {
            if ((x - di2[i] <= 8 && x - di2[i] >= 1) && (y - di1[j] >= 1 && y - di1[j] <= 8))

            {
                dem++;
            }
        }

    for (int i = 0; i <= 1; i++)
        for (int j = 0; j <= 1; j++)
        {
            if (((x - di1[i] <= 8 && x - di1[i] >= 1) && (y - di2[j] >= 1 && y - di2[j] <= 8)))

            {
                dem++;
            }
        }
    return dem;
}
```

Bài 2: Cho số nguyên dương n. Hãy tạo ra ma trận vuông kích thước n*n chứa các số từ 1 tới n*n tăng dần theo hình xoắn ốc, xuất phát từ ô trên trái và đi theo theo theo chiều kim đồng hồ

Ví dụ:

Với n = 3, kết quả spiralNumbers(n) = [[1, 2, 3],

[8, 9, 4],

[7, 6, 5]]

- [Đầu vào] integer n: Độ dài của ma trận, 3 ≤ n ≤ 100.
- [Đầu ra] matrix.integer

Gợi ý:

- Tạo ra ma trận kích thước n*n
- Sử dụng 2 biến để lưu trữ hàng và cột hiện tại, bắt đầu xuất phát từ ô (0,0) (trên trái)
- Sử dụng 1 biến để lưu trữ giá trị hiện tại của số nằm trong ma trận. Biến này sẽ tăng dần từ 1 cho tới n*n
- Điền số vào ma trận theo nguyên tắc:
    - Với cạnh đi từ trái qua phải: tăng dần cột, giữ nguyên hàng
    - Với cạnh đi từ trên xuống dưới: tăng dần hàng, giữ nguyên cột
    - Với cạnh đi từ phải qua trái: giảm dần cột, giữ nguyên hàng
    - Với cạnh đi từ dưới lên trên: giảm dần hàng, giữ nguyên cột.

Hướng dẫn:

```
vector< vector<int> > spiralNumbers(int n)
{   vector< vector<int> > a;
    vector<int> b;
    int x=0,y=n-1,k=0;
    for (int i=0; i<n; i++) b.push_back(1);

    for (int i=0; i<n; i++) a.push_back(b);

    while (k<n*n)
    { {for (int i=x;i<=y;i++) {k++;a[x][i]=k;}}
      {for (int i=x+1;i<=y;i++) {k++;a[i][y]=k;}}
      {for (int i=y-1;i>=x;i--) {k++;a[y][i]=k;}}
      {for (int i=y-1;i>=x+1;i--) {k++;a[i][x]=k;}}
      x++;y--;
    }
    return a;

}
```

Bài 3: Cho ma trận n x m chứa tất cả các số nguyên từ 1 tới n*m, mỗi số chỉ xuất hiện một lần. Bạn xuất phát từ ô chứa số 1. Mỗi nước đi, bạn được quyền đi tới 4 ô xung quanh chung cạnh. Mỗi ô bạn chỉ có thể đi qua một lần

Hãy kiểm tra xem có tồn tại phương án đi qua tất cả các ô với thứ tự tăng dần của các ô trong ma trận? (xuất phát từ ô số 1, tới ô số 2, rồi tới ô số 3, ... cuối cùng kết thúc ở ô n*m)

Ví dụ:

Với matrix = [[1, 4, 5], [2, 3, 6]] thì kết quả findPath(matrix) = true. Bạn có thể đi lần lượt từ 1 tới 6.

Với matrix = [[1, 3, 6], [2, 4, 5]] thì kết quả findPath(matrix) = false. Bạn có thể đi từ ô 1 sang ô 2, nhưng không thể đi từ 2 sang 3 do hai ô này không chung cạnh.

- [Đầu vào] matrix.integer matrix: Mảng 2 chiều không rỗng chứa các số nguyên thể hiện ma trận hình chữ nhật. $1 \leq$ matrix.length ≤ 5, $1 \leq$ matrix[0].length ≤ 10, $1 \leq$ matrix[i][j] ≤ 25.
- [Đầu ra] boolean

Hướng dẫn:

```cpp
bool findPath(std::vector<std::vector<int>>& matrix) {
    int column = matrix[0].size();
    int row = matrix.size();
    int columnOne = -1;
    int rowOne = -1;
    int count = 2;

    // Find the position of number 1
    for (int i = 0; i < row; i++) {
        for (int j = 0; j < column; j++) {
            if (matrix[i][j] == 1) {
                columnOne = j;
                rowOne = i;
            }
        }
    }

    if (columnOne == -1 && rowOne == -1) {
        return false;
    }

    // Check if the path is valid
    while (count <= column * row) {
        if (rowOne + 1 >= 0 && rowOne + 1 < row && matrix[rowOne + 1][columnOne] == count) {
            count++;
            rowOne++;
        } else if (rowOne - 1 >= 0 && rowOne - 1 < row && matrix[rowOne - 1][columnOne] == count) {
            count++;
            rowOne--;
        } else if (columnOne - 1 >= 0 && columnOne - 1 < column && matrix[rowOne][columnOne - 1] == count) {
            count++;
            columnOne--;
        } else if (columnOne + 1 >= 0 && columnOne + 1 < column && matrix[rowOne][columnOne + 1] == count) {
            count++;
            columnOne++;
        } else {
            return false;
        }
    }
    return true;
}
```

VI. Sắp xếp (Cơ bản)

Sắp xếp là quá trình bố trí lại các phần tử trong một tập hợp theo một trình tự nào đó nhằm mục đích giúp quản lý và tìm kiếm các phần tử dễ dàng và nhanh chóng hơn.

Sau đây sẽ là phần giới thiệu một số thuật toán sắp xếp nổi tiếng và thông dụng nhất!

Sắp xếp nổi bọt (Bubble Sort):

Duyệt qua danh sách, làm cho các phần tử lớn nhất hoặc nhỏ nhất dịch chuyển về phía cuối danh sách, tiếp tục lại làm phần tử lớn nhất hoặc nhỏ nhất kế đó dịch chuyển về cuối hay chính là làm cho phần tử nhỏ nhất (hoặc lớn nhất) nổi lên, cứ như vậy cho đến hết danh sách.

Độ phức tạp trung bình: $O(n^2)$

```
void bubbleSort(int arr[], int n) {
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                // Swap arr[j] and arr[j+1]
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }
}
```

Sắp xếp chọn (Selection Sort):

Duyệt từ đầu đến phần tử kề cuối danh sách, duyệt tìm phần tử nhỏ nhất từ vị trí kế phần tử đang duyệt đến hết, sau đó đổi vị trí của phần tử nhỏ nhất đó với phần tử đang duyệt và cứ tiếp tục như vậy.

Độ phức tạp trung bình: $O(n^2)$

```
void selectionSort(int arr[], int n) {
    for (int i = 0; i < n - 1; i++) {
        int minIndex = i;
        for (int j = i + 1; j < n; j++) {
            if (arr[j] < arr[minIndex]) {
                minIndex = j;
            }
        }
        swap(arr[i], arr[minIndex]);
    }
}
```

Sắp xếp chèn (Insertion Sort)

Ý tưởng của thuật toán này như sau: ta có mảng ban đầu gồm phần tử A[0] xem như đã sắp xếp, ta sẽ duyệt từ phần tử 1 đến n – 1, tìm cách chèn những phần tử đó vào vị trí thích hợp trong mảng ban đầu đã được sắp xếp.

Độ phức tạp trung bình: $O(n^2)$

```cpp
void insertionSort(std::vector<int>& arr) {
    int n = arr.size();
    for (int i = 1; i < n; ++i) {
        int key = arr[i];
        int j = i - 1;

        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
    }
}
```

Bài 1: Cho mảng số nguyên. Hãy sắp xếp chúng theo nguyên tắc:

a đứng trước b nếu tổng các chữ số của a nhỏ hơn b

Nếu hai số có tổng các chữ số bằng nhau, số nào nhỏ hơn sẽ đứng trước. Ví dụ 4 và 13 có tổng chữ số bằng nhau, nhưng 4 < 13 nên 4 sẽ đứng trước 13 trong mảng kết quả.

Ví dụ

Ví dụ a = [13, 20, 7, 4], kết quả digitalSumSort(a) = [20, 4, 13, 7].

- [Đầu vào] array.integer a: Mảng các số nguyên, $4 \leq$ a.length ≤ 20, $1 \leq$ a[i] $\leq$ 1000
- [Đầu ra] array.integer: Mảng đã được sắp xếp theo yêu cầu đề bài

Gợi ý: Sử dụng các thuật toán sắp xếp ở phần trên.

Hướng dẫn:

```cpp
int sum(int n){
    int d = 0;
    while(n > 0){
        d += n % 10;
        n /= 10;
    }
    return d;
}
std::vector<int> digitalSumSort(std::vector<int> a)
{
    for (int i = 0; i < a.size()-1; i++)
    for (int j = i + 1; j < a.size(); j++)
    if (sum(a[i]) > sum(a[j]) || (sum(a[i]) == sum(a[j]) && a[i] > a[j])){
        swap(a[i] ,a[j]);
    }
    return a;
}
```

PHẦN 2: CÁC THUẬT TOÁN NÂNG CAO

I. Đệ quy

Trong lập trình, một hàm được gọi là đệ quy nếu bên trong thân hàm có một lời gọi đến chính nó.

Hàm đệ quy luôn có điều kiện dừng được gọi là "điểm neo". Khi đạt tới điểm neo, hàm sẽ không gọi chính nó nữa.

Khi được gọi, hàm đệ quy thường được truyền cho một tham số, thường là kích thước của bài toán lớn ban đầu. Sau mỗi lời gọi đệ quy, tham số sẽ nhỏ dần, nhằm phản ánh bài toán đã nhỏ hơn và đơn giản hơn. Khi tham số đạt tới một giá trị cực tiểu (tại điểm neo), hàm sẽ chấm dứt.

```
void countDown(int count)
{
    cout << "push " << count << '\n';

    if (count > 1) // điều kiện dừng
        countDown(count - 1);

    cout << "pop " << count << '\n';
}

int main()
{
    countDown(3);
    return 0;
}
```

Hàm đệ quy phải có một điều kiện kết thúc đệ quy, nếu không chương trình sẽ lặp vô hạn (đến khi tràn bộ nhớ ngăn xếp). **Điều kiện dừng** của hàm đệ quy gọi là điều kiện cơ sở.

Bài 1: Cho n là một số tự nhiên (n>=0). Hãy tính giai thừa của n (n!) biết rằng 0! = 1 và n! = (n-1)! * n. Chú ý kết quả sẽ được lấy mod của 1000000007.

Ví dụ: GiaiThua(3) = 6, GiaiThua(8) = 40320

- [Đầu vào] integer n: 0 <= n <= 1000
- [Đầu ra] integer: kết quả của phép tình giai thừa

Hướng dẫn:

```
long GiaiThua(long long n)
{
    if (n == 0)
    {
        return 1;
    }
    return (n * GiaiThua(n - 1)) % 1000000007;
}
```

Bài 2: Dãy Fibonaci là dãy vô hạn các số tự nhiên. Số Fibonaci thứ n, ký hiệu F(n), được định nghĩa như sau :

- **F(n) = 0, nếu n = 0**
- **F(n) = 1, nếu n = 1**
- **F(n) = F(n-1) + F(n-2), nếu n > 1**

Yêu cầu: Tính số fibonaci thứ n với n cho trước. Chú ý kết quả sẽ được lấy mod của 1000000007.

Ví dụ: fibonacci(4) = 3, fibonacci(15) = 610

- [Đầu vào] integer n: 0 <= n <= 100
- [Đầu ra] integer: số Fibonacci thứ n

Lý thuyết:

Ta thấy số fibonaci là một đối tượng có bản chất đệ quy, do đó ta có thể tìm được số fibonaci thứ *n* bằng một hàm đệ quy nhị phân như sau:

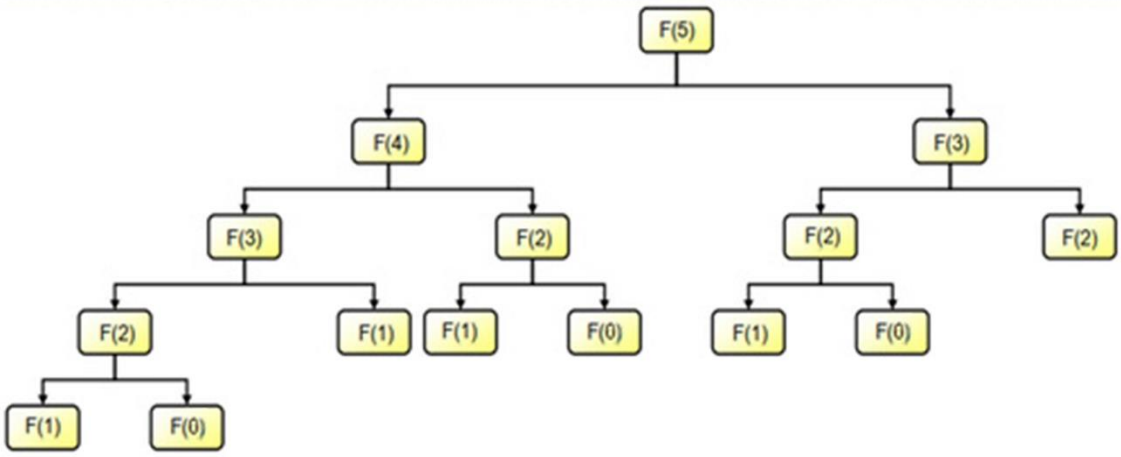
```
long fibonacciRecursion(long number){
    // fibo with recursion
    if (number <= 1)
    {
        return number;
    }
    return (fibonacciRecursion(number-1) + fibonacciRecursion(number-2)) % 1000000007;

}
```

Mặc dù rất đơn giản, nhưng cách cài đặt đệ quy này có một số nhược điểm khiến cho nó không được khuyến khích trong việc lập trình.

Mọi bài toán có bản chất đệ quy đều có thể chuyển về cách cài đặt khử đệ quy, bởi vì thực tế các hàm đệ quy sẽ được chương trình dịch chuyển về những mã lệnh không đệ quy trước khi thực hiện. Vì thế, bài toán Fibonaci có thể viết được lại ở dạng không đệ quy sử dụng **mảng một chiều**.

Nếu cùng tính giá trị *f*(5), thì giải thuật không đệ quy chỉ cần mất đúng 6 bước tính toán, vì kết quả các bài toán sau khi tính xong sẽ được lưu lại luôn trên mảng một chiều, chỉ cần sử dụng luôn để tính toán. Còn đối với giải thuật đệ quy, thì việc phân tách các bài toán con sẽ diễn ra như sau:



Ta thấy để tính được bài toán f(5), chương trình phải tính lại 1 lần f(4), 2 lần f(3), 3 lần f(2), 5 lần f(1) và 3 lần f(0). Đó gọi là sự xuất hiện của các bài toán con gối nhau, khiến cho chương trình phải tính toán lại nhiều lần một bài toán trong khi đáng lẽ nó chỉ cần tính một lần, từ đó làm cho chương trình chạy rất chậm. Đối với những hàm

đệ quy có nhiều lời gọi hơn thì độ phức tạp tính toán sẽ còn tăng lên gấp nhiều lần hơn nữa!

Hướng dẫn:

```
long fibonacci(long number)
{
    vector<long> fibo(number+1);
    fibo[0] = 0;
    fibo[1] = 1;
    for (long i = 2; i <= number; i++)
    {
        fibo[i] = (fibo[i-1] + fibo[i-2]) % 1000000007;
    }
    return fibo[number];
}
```

II. Sắp xếp (Nâng cao)

Sắp xếp trộn (Merge Sort)

Sắp xếp trộn (merge sort) là một thuật toán dựa trên kỹ thuật chia để trị, ý tưởng của thuật toán này như sau: chia đôi mảng thành hai mảng con, sắp xếp hai mảng con đó và trộn lại theo đúng thứ tự, mảng con được sắp xếp bằng cách tương tự.

Giả sử left là vị trí đầu và right là cuối mảng đang xét, cụ thể các bước của thuật toán như sau:

- Nếu mảng còn có thể chia đôi được (tức left < right)
- Tìm vị trí chính giữa mảng
- Sắp xếp mảng thứ nhất (từ vị trí left đến mid)
- Sắp xếp mảng thứ 2 (từ vị trí mid + 1 đến right)
- Trộn hai mảng đã sắp xếp với nhau

Độ phức tạp trung bình: O(nlog(n))

```
void merge(int arr[], int left, int mid, int right) {
    int n1 = mid - left + 1;
    int n2 = right - mid;

    int L[n1], R[n2];

    for (int i = 0; i < n1; i++)
        L[i] = arr[left + i];
    for (int j = 0; j < n2; j++)
        R[j] = arr[mid + 1 + j];

    int i = 0, j = 0, k = left;

    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k] = L[i];
            i++;
        } else {
            arr[k] = R[j];
            j++;
        }
        k++;
    }

    while (i < n1) {
        arr[k] = L[i];
        i++;
        k++;
    }

    while (j < n2) {
        arr[k] = R[j];
        j++;
        k++;
    }
}

void mergeSort(int arr[], int left, int right) {
    if (left < right) {
        int mid = left + (right - left) / 2;

        mergeSort(arr, left, mid);
        mergeSort(arr, mid + 1, right);

        merge(arr, left, mid, right);
    }
}
```

Sắp xếp nhanh (Quick Sort)

Sắp xếp nhanh (quick sort) hay sắp xếp phân đoạn (Partition) là là thuật toán sắp xếp dựa trên kỹ thuật chia để trị, cụ thể ý tưởng là: chọn một điểm làm chốt (gọi là pivot), sắp xếp mọi phần tử bên trái chốt đều nhỏ hơn chốt và mọi phần tử bên phải đều lớn hơn chốt, sau khi xong ta được 2 dãy con bên trái và bên phải, áp dụng tương tự cách sắp xếp này cho 2 dãy con vừa tìm được cho đến khi dãy con chỉ còn 1 phần tử.

Cụ thể áp dụng thuật toán cho mảng như sau:

- Chọn một phần tử làm chốt
- Sắp xếp phần tử bên trái nhỏ hơn chốt
- Sắp xếp phần tử bên phải nhỏ hơn chốt
- Sắp xếp hai mảng con bên trái và bên phải pivot

Phần tử được chọn làm chốt rất quan trọng, nó quyết định thời gian thực thi của thuật toán. Phần tử được chọn làm chốt tối ưu nhất là phần tử **trung vị**, phần tử này làm cho số phần tử nhỏ hơn trong dãy bằng hoặc sấp xỉ số phần tử lớn hơn trong dãy. Tuy nhiên, việc tìm phần tử này rất tốn kém, phải có thuật toán tìm riêng, từ đó làm giảm hiệu suất của thuật toán tìm kiếm nhanh, do đó, để đơn giản, người ta thường sử dụng phần tử chính giữa làm chốt.

Độ phức tạp trung bình: O(nlog(n))

```c
// Function to swap two elements
void swap(int* a, int* b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

// Function to partition the array and return the pivot index
int partition(int arr[], int left, int right) {
    int pivot = arr[left + (right - left) / 2]; // Choosing the middle element as the pivot
    int i = left - 1;

    for (int j = left; j <= right - 1; j++) {
        if (arr[j] < pivot) {
            i++;
            swap(&arr[i], &arr[j]);
        }
    }

    swap(&arr[i + 1], &arr[right]);
    return i + 1;
}

// Function to perform quicksort
void quicksort(int arr[], int left, int right) {
    if (left < right) {
        int pivot = partition(arr, left, right);

        quicksort(arr, left, pivot - 1);
        quicksort(arr, pivot + 1, right);
    }
}
```

Bài 1: Một nhóm người đứng thành hàng trong công viên. Giữa họ có một số cây không thể di chuyển

Nhiệm vụ của bạn là thay đổi vị trí của họ, sao cho chiều cao của họ tạo thành một dãy tăng dần (không tính cây). Chú ý rằng cây không thể di chuyển

Ví dụ

Với a = [-1, 150, 190, 170, -1, -1, 160, 180], thì kết quả sortByHeight(a) = [-1, 150, 160, 170, -1, -1, 180, 190].

- [Đầu vào] array.integer a, $1 \leq a.length \leq 1000$, $-1 \leq a[i] \leq 1000$.
- [Đầu ra] array.integer: Mảng đầu ra thể hiện độ cao của nhóm người sau khi đã đổi chỗ

Gợi ý: Sử dụng các thuật toán sắp xếp ở phần trên.

III. Các thuật toán sinh

Bài 1: Sinh nhị phân.

Hãy hiển thị tất cả các cấu hình nhị phân có n bit.

Ví dụ: n = 3. Kết quả sẽ là:

000

001

010

011

100

101

110

111

- [Đầu vào] integer n: 0<n<20
- [Đầu ra] Các số nhị phân n bit. Mỗi số hiển thị trên một dòng.

Lý thuyết

Một chương trình muốn sử dụng được **thuật toán sinh** phải thỏa mãn được 2 điều kiện sau:

- Xác định trình tự, điểm đầu và điểm cuối của tập hợp cấu hình. Thật vậy, trong khi điểm đầu có ý nghĩa là cơ sở để hàm sinh chạy được, điểm cuối của tập hợp cấu hình sẽ đảm bảo thuật toán có điểm dừng.
- Xây dựng được thuật toán mà từ một cấu hình chưa phải cuối thì ta luôn sinh ra một cấu hình kế tiếp nó.

Trong bài tập đầu tiên, ta sẽ đi tìm hiểu về **thuật toán sinh nhị phân** là một trong những thuật toán sinh đơn giản nhất. Lưu ý có rất nhiều ý tưởng về giải thuật sinh. Với bài này chúng ta sẽ áp dụng đệ quy vào chương trình.

Hướng dẫn:

```
void generateBinary(int n, std::vector<int>& binary) {
    if (n == 0) {
        for (int bit : binary) {
            std::cout << bit;
        }
        std::cout << std::endl;
    } else {
        binary.push_back(0);
        generateBinary(n - 1, binary);
        binary.pop_back();

        binary.push_back(1);
        generateBinary(n - 1, binary);
        binary.pop_back();
    }
}
```

Dễ nhận thấy cấu hình đầu sẽ có dạng 0000…0 và cấu hình cuối là 1111…1.

Hàm generateBinary nhận đầu vào là số bit cần sinh và một vector để lưu trữ chuỗi nhị phân. Trong mỗi lần gọi đệ quy, chúng ta thêm một bit vào vector và giảm số bit cần sinh đi 1. Sau đó, chúng ta đệ quy gọi hàm generateBinary với số bit giảm đi 1. Khi số bit cần sinh đạt đến 0, chúng ta in ra chuỗi nhị phân đã tạo. Cụ thể thuật toán như sau:

- Nếu n bằng 0, tức là không còn bit nào cần sinh nữa, ta in ra chuỗi nhị phân đã tạo bằng cách duyệt qua từng phần tử trong vector và in chúng ra màn hình.
- Ngược lại, nếu n khác 0, ta thêm bit 0 vào cuối vector và gọi đệ quy hàm generateBinary với n giảm đi 1.
- Tiếp theo, ta xóa bit 0 vừa thêm vào để thử thêm bit 1.
- Sau đó, ta thêm bit 1 vào cuối vector và gọi đệ quy hàm generateBinary với n giảm đi 1.
- Khi đã thử tất cả các trường hợp (thêm bit 0 và thêm bit 1), ta xóa bit 1 vừa thêm vào để các hàm đệ quy có thể hoạt động đúng.

Bài 2: Sinh hoán vị

Cho một mảng gồm n phần từ là các số tự nhiên 1,2, 3,…n . Hãy hiển thị tất cả các hoàn vị của mảng trên.

Ví dụ: n = 3. Kết quả:

123

132

213

231

312

321

- [Đầu vào] integer: 0<n<10.
- [Đầu ra] Các hoán vị của mảng. Mỗi hoán vị in trên 1 dòng.

Lý thuyết:

Đối với bài này, chúng ta sẽ sử dụng phương pháp xử lý trên mảng truyền thống để sinh các hoán vị. Cụ thể như sau:

- Khởi tạo cấu hình ban đầu : 1,2,3,..n.
- Xét từ cuối dãy lên đầu dãy, tìm phần tử đầu tiên làm mất tính không giảm của dãy, đánh dấu phần tử đó. Nếu không tìm được, kết thúc quá trình sinh.
- Xét từ cuối lên đầu dãy, tìm phần tử đầu tiên lớn hơn phần tử được dánh dấu, đổi chỗ 2 phần tử đó.
- Từ vị trí phần tử tìm được ở bước 2, sắp xếp tăng dần từ vị trí đó cho đến cuối dãy. In cấu hình và quay lại bước 2.

```
int n, stop = 0, a[9];

void initialize() {
    for (int i = 1; i <= n; i++)
        a[i] = i;
}

void generateNextPermutation() {
    int i = n - 1;
    while (i > 0 && a[i] > a[i + 1])
        i--;
    if (i == 0)
        stop = 1;
    else {
        int k = n;
        while (a[i] > a[k])
            k--;
        swap(a[k], a[i]);
        int c = n, r = i + 1;
        while (r < c) {
            swap(a[c], a[r]);
            r++;
            c--;
        }
    }
}

void printPermutation() {
    for (int i = 1; i <= n; i++)
        cout << a[i];
    cout << endl;
}

void generatePermutations() {
    cin >> n;
    initialize();
    do {
        printPermutation();
        generateNextPermutation();
    } while (!stop);
}
```

Bài 3: Sinh tổ hợp

Hiển thị các tổ hợp chập k của n phần từ 1, 2, 3,…n.

Ví dụ: Với n = 4, k = 2. Ta được kết quả:

1 2

1 3

1 4

2 3

2 4

3 4

- [Đầu vào] integer: 1<=n<=10, 1<=k<=n
- [Đầu ra] Các tổ hợp chập k của n phần tử. Mỗi tổ hợp in trên một dòng.

Hướng dẫn:

Với sinh các tổ hợp, ta sẽ thấy cấu hình đầu tiên sẽ có dạng 1, 2,…k và cấu hình cuối cùng là n – k + 1, n – k + 2, … n.

Từ đó, ta cần cài đặt một cấu trúc có thể xử lý một mảng theo dạng như sau:

- Khởi tạo một mảng rỗng, số start ban đầu sẽ là 1.
- Tiếp tục thêm các số tiếp theo vào mảng cho đến khi đủ k số.
- Khi đã đến số thứ k, ta hiển thị cấu hình kết quả, sau đó loại 1 số cuối đi là k ra khỏi mảng và thêm k + 1 vào vị trí cuối mảng để hiển thị cấu hình tiếp theo.
- Lặp lại bước trên cho đến khi số cần thêm vào mảng bằng n, ta sẽ bỏ 2 số cuối đi và tiếp tục làm lại việc thêm – bỏ 2 số cuối trên mảng.

Nhận thấy đây là cấu trúc lặp có thể áp dụng đệ quy, ta sẽ triển khai như sau:

```
void generateCombinations(vector<int>& combination, int start, int k, int n) {
    if (k == 0) {
        // Print the combination
        for (int num : combination) {
            cout << num << " ";
        }
        cout << endl;
        return;
    }

    for (int i = start; i <= n; i++) {
        combination.push_back(i);
        generateCombinations(combination, i + 1, k - 1, n);
        combination.pop_back();
    }
}
```

Nếu k bằng 0, tức là đã tạo được một tổ hợp, hàm sẽ in ra các phần tử trong combination và kết thúc hàm.

Nếu k không bằng 0, hàm sẽ lặp qua các số từ start đến n. Với mỗi số i, nó sẽ thêm i vào combination, sau đó đệ quy gọi lại hàm generateCombinations với các tham số mới: i+1 để đảm bảo không lặp lại các phần tử, k-1 để giảm số lượng phần tử cần chọn, và n không thay đổi. Sau khi hàm đệ quy gọi xong, phần tử i sẽ được loại bỏ khỏi combination để chuẩn bị cho lần lặp tiếp theo, giúp sinh ra tất cả các tổ hợp có thể.

Bài 4: [*https://www.spoj.com/PTIT/problems/BCCOW*] Nông dân John đang đưa các con bò của anh ta đi xem phim! Xe tải của anh ta thì có sức chứa có hạn thôi, là C (100 <= C <= 5000) kg, anh ta muốn đưa 1 số con bò đi xem phim sao cho tổng khối lượng của đống bò này là lớn nhất, đồng thời xe tải của anh ta vẫn chịu được.

Cho N (1 <= N <= 16) con bò và khối lượng W_i của từng con, hãy cho biết khối lượng bò lớn nhất mà John có thể đưa đi xem phim là bao nhiêu.

Ví dụ: với C = 259, N = 5, Arr = {81, 58, 42, 33, 61}; ta được kết quả là maxWeight = 242 (81+58+42+61)

- [Đầu vào] integer, array integer: 100<=C<=5000; 1<=N<=16.
- [Đầu ra] integer: Khối lượng bò lớn nhất có thể chở.

Gợi ý: Sinh nhị phân dãy có độ dài n để kiểm soát tất cả các trường hợp.

IV. Tham lam

Giải thuật tham lam (Greedy algorithm) là một thuật toán giải quyết một bài toán theo kiểu metaheuristic (tối ưu hóa) để tìm kiếm lựa chọn tối ưu địa phương ở mỗi bước đi với hy vọng tìm được tối ưu toàn cục.

Tư tưởng chung của phương pháp là chấp nhận tìm ra các nghiệm gần đúng với nghiệm tối ưu (nghĩa là có thể sai), rồi tìm cách xây dựng một hàm tính toán độ tối ưu cho phương án sao cho khả năng ra được nghiệm tối ưu là lớn nhất có thể. Ưu điểm của phương pháp này là độ phức tạp khá nhỏ, và nếu triển khai đúng cách có thể cho ra nghiệm tối ưu nhanh hơn nhiều so với *Quay lui* hay *Nhánh và Cận* (sẽ được đề cập ở các phần sau). Thực tế, có nhiều thuật toán sử dụng chiến lược giải

thuật này và vẫn cho được kết quả tối ưu, chẳng hạn như Giải thuật Prim hay Kruskal để tìm cây khung nhỏ nhất trên đồ thị.

Bài 1: Mỗi phân số dương đều có thể được biểu diễn dưới dạng tổng của các phân số đơn vị khác nhau (phân số đơn vị là phân số có tử số bằng 1, và mẫu số là một số nguyên dương). Cách biểu diễn phân số như vậy được gọi là biểu diễn theo *Phân số Ai Cập*, và mỗi phân số có rất nhiều cách biểu diễn như vậy. Cho trước một phân số a/b , hãy tìm biểu diễn phân số Ai Cập của nó với số lượng số hạng là ít nhất có thể?

Ví dụ: egyptian_representation(6, 14), ta có kết quả là:

1 3

1 11

1 231

- [Đầu vào] integer : 1 <= a < b <= 1000
- [Đầu ra] In ra các phân số trong phân tích tìm được, mỗi phân số trên một dòng theo thứ tự giảm dần về giá trị.

Lý thuyết:

Nghiệm của bài toán được biểu diễn dưới dạng một vector $X = (x_1, x_2, .., x_n)$ sao cho:

$$\frac{a}{b} = \frac{1}{x_1} + \frac{1}{x_2} + \cdots \frac{1}{x_n}$$

Với $x_1 < x_2 < \ldots < x_n$ và n nhỏ nhất có thể.

Mỗi phân số dương có tử số nhỏ hơn mẫu số đều có thể được rút gọn về một phân số tối giản. Vì thế, ta có thể áp dụng giải thuật tham lam như sau:

- Nếu $\frac{a}{b}$ có mẫu số chia hết cho tử số thì bài toán đã có lời giải, vì khi đó nó có thể viết dưới dạng $\frac{a}{b \div a}$

- Với một phân số $\frac{a}{b}$ $(1 < a < b)$, tìm phân số đơn vị lớn nhất không vượt quá $\frac{a}{b}$ bằng cách tính giá trị $x = \lceil \frac{a}{b} \rceil$, phân số đơn vị tìm được sẽ là $\frac{1}{x}$. Sở dĩ ta tìm phân số $\frac{1}{x}$ lớn nhất là để cho số lượng số hạng tạo ra sẽ nhỏ nhất có thể.
- Tiếp tục lặp lại quá trình trên với hiệu $\frac{a}{b} - \frac{1}{x}$, cho tới khi phân tích xong.

Giải thuật có độ phức tạp không ổn định, tùy thuộc vào việc lựa chọn các phân số đơn vị, tuy nhiên vẫn sẽ chạy rất nhanh.

Hướng dẫn:

```
void egyptian_representation(int a, int b)
{

    if (a == 0 || b == 0)
        return;


    if (b % a == 0)
    {
        cout << 1 << ' ' << b / a;
        return;
    }


    int x = b / a + 1;
    cout << 1 << ' ' << x << endl;


    egyptian_representation(a * x - b, b * x);
}

main()
{
    int a, b;
    cin >> a >> b;

    egyptian_representation(a, b);
}
```

Bài 2: Một khách hàng muốn rút số tiền *T* từ một cây ATM bên đường. Bên trong cây ATM hiện đang có *n* tờ tiền có mệnh giá $a_1,a_2,\ldots,a_n$. Hãy tìm số tờ tiền tối thiểu cần sử dụng để ATM trả tiền cho khách hàng?

Ví dụ: n = 5, T = 10, arr = {1, 4, 2, 3, 6} thì kết quả là 2 (tờ 4 và tờ 6)

Ví dụ: n = 6, T = 100, arr = {50, 20, 20, 20, 20, 20} thì kết quả là 5 (5 tờ 20)

- [Đầu vào] integer: 1 <= n, T <= 1000; 1 <= a_i <= 1000.
- [Đầu ra] integer: In ra số nguyên duy nhất là số tờ tiền tối thiểu cần sử dụng. Nếu không có phương án trả tiền thì in ra −1.

Lý thuyết:

Với giới hạn này, bài toán không thể giải quyết bằng giải pháp Quay lui hay Nhánh và Cận. Phương pháp tốt nhất sẽ là **Quy hoạch động** (sẽ được nhắc tới ở phần sau), tuy nhiên ở đây tôi sẽ phân tích một ý tưởng Tham lam đơn giản như sau:

- Sắp xếp các tờ tiền theo mệnh giá giảm dần.
- Tại mỗi bước lựa chọn, luôn luôn chọn tờ tiền có mệnh giá lớn nhất và không vượt quá số tiền còn lại phải trả.

Giải thuật trên sẽ tìm ra nghiệm rất nhanh, **tuy nhiên không phải luôn luôn tối ưu** và **cũng có thể sẽ không tìm được nghiệm**. Đó chính là nhược điểm của phương pháp Tham lam mà chúng ta buộc phải chấp nhận. Mặt khác, phương pháp này phù hợp với chiến thuật vét các test khi thi thể thức OLP.

Hướng dẫn:

Chú ý: Code mẫu dưới làm theo giải thuật Tham lam nên chưa tối ưu. Trong phần sau ở giải thuật Quy hoạch động, ta sẽ quay trở lại problem này.

Thử test n = 6, T = 100, arr = {50, 20, 20, 20, 20, 20} bằng cách giải dưới sẽ bị đưa ra kết quả sai.

```cpp
main()
{
    int n, T;
    cin >> n >> T;

    vector < int > a(n + 1);
    for (int i = 1; i <= n; ++i)
        cin >> a[i];

    sort(a.begin() + 1, a.end(), greater < int >());

    int res = 0;
    for (int i = 1; i <= n; ++i)
        if (T >= a[i])
        {
            T -= a[i];
            ++res;
        }

    if (T == 0)
        cout << res;
    else
        cout << -1;

    return 0;
}
```

Bài 3: [*https://www.geeksforgeeks.org/problems/activity-selection-1587115620/1*] Cho N hoạt động với ngày bắt đầu và ngày kết thúc được cho trong mảng start[ ] và end[ ]. Chọn số lượng hoạt động tối đa mà một người có thể thực hiện, giả sử rằng một người chỉ có thể thực hiện một hoạt động duy nhất vào một ngày nhất định. Lưu ý: Thời lượng của hoạt động bao gồm cả ngày bắt đầu và ngày kết thúc.

Ví dụ:

- N = 2, start[] = {2, 1}, end[] = {2, 2} → Kết quả là 1 (người đó chỉ thực hiện được 1 công việc)
- N = 4, start[] = {1, 3, 2, 5}, end[] = {2, 4, 3, 6} → Kết quả là 3 (người đó thực hiện được công việc 1, 2 và 4.

- [Đầu vào] integer, array integer: $1 \leq N \leq 2*10^5$, $1 \leq start[i] \leq end[i] \leq 10^9$
- [Đầu ra] integer: số lượng công việc tối đa người đó có thể hoàn thành.

Gợi ý: Sử dụng thuật toán tham lam để làm bài tập này.

V. Quay lui

Quay lui là một kĩ thuật thiết kế giải thuật dựa trên đệ quy. Ý tưởng của quay lui là tìm lời giải từng bước, mỗi bước chọn một trong số các lựa chọn khả dĩ và đệ quy.

Dùng để giải bài toán liệt kê các cấu hình. Mỗi cấu hình được xây dựng bằng từng phần tử. Mỗi phần tử lại được chọn bằng cách thử tất cả các khả năng.

Các bước trong việc liệt kê cấu hình dạng X[1...n]:

- Xét tất cả các giá trị X[1] có thể nhận, thử X[1] nhận các giá trị đó. Với mỗi giá trị của X[1] ta sẽ:
- Xét tất cả giá trị X[2] có thể nhận, lại thử X[2] cho các giá trị đó. Với mỗi giá trị X[2] lại xét khả năng giá trị của X[3]...tiếp tục như vậy cho tới bước:
- ...
- Xét tất cả giá trị X[n] có thể nhận, thử cho X[n] nhận lần lượt giá trị đó.
- Thông báo cấu hình tìm được.

Bản chất của quay lui là một quá trình tìm kiếm theo chiều sâu(Depth-First Search).

Mô hình thuật toán:

```
Backtracking(k) {
        for([Mỗi phương án chọn i(thuộc tập D)]) {
                if ([Chấp nhận i]) {
                        [Chọn i cho X[k]];
                        if ([Thành công]) {
                                [Đưa ra kết quả];
                        } else {
                                Backtracking(k+1);
                                [Bỏ chọn i cho X[k]];
                        }
                }
        } }
```

Bài 1: Cho bàn cờ vua có kích thước n × n (4 ≤ n ≤ 10), các cột được đánh số từ 1 đến n theo chiều từ trái qua phải, các hàng được đánh số từ 1 đến n theo chiều từ trên xuống dưới. Hãy tìm cách đặt n quân hậu trên bàn cờ sao cho không có 2 quân hậu nào ở cùng một hàng, cột hay đường chéo (không thể "ăn" nhau)

Ví dụ: n = 4, kết quả sẽ là:

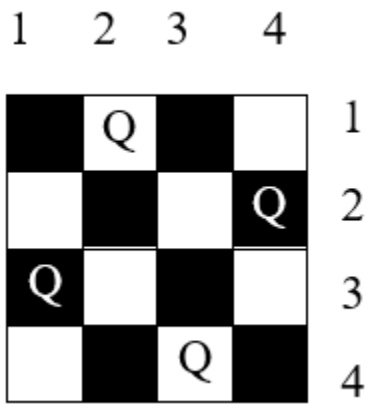
1 2

2 4

3 1

4 3

Giải thích kết quả:



- [Đầu vào] integer: 2 <= n <= 10
- [Đầu ra] array integer: các vị trí của các quân hậu. In ra -1 nếu không có cách giải.

Hướng dẫn:

Trong giải thuật này, chúng ta sử dụng một mảng col để lưu cột mỗi quân hậu được đặt ở hàng thứ i. Ví dụ, col[1] sẽ chứa số cột của quân hậu được đặt ở hàng thứ 1, col[2] chứa số cột của quân hậu được đặt ở hàng thứ 2, và cứ tiếp tục như vậy.

Hàm isSafe() được sử dụng để kiểm tra xem có thể đặt quân hậu vào ô (x, y) hay không. Nó kiểm tra xem cột y có trùng với cột của bất kỳ quân hậu nào ở các hàng trước đó hay không, cũng như kiểm tra đường chéo.

Hàm solveNQueens() là hàm quay lui chính. Nó thử tất cả các cách đặt quân hậu từ hàng thứ 1 đến hàng thứ n, và nếu không thể đặt quân hậu vào một hàng cụ thể, nó sẽ quay lui lại và thử một cách đặt khác.

```
const int MAX_N = 10;
int n;
vector<int> col; // Lưu cột mỗi quân hậu đặt ở hàng thứ i
bool isSafe(int x, int y) {
    // Kiểm tra cột
    for (int i = 1; i < x; ++i) {
        if (col[i] == y || (abs(i - x) == abs(col[i] - y))) {
            return false;
        }
    }
    return true;
}
// Hàm quay lui để đặt quân hậu từ hàng thứ x trở đi
bool solveNQueens(int x) {
    if (x > n) {
        // Đã đặt n quân hậu thành công
        for (int i = 1; i <= n; ++i) {
            cout << i << " " << col[i] << endl;
        }
        return true;
    }
    for (int y = 1; y <= n; ++y) {
        if (isSafe(x, y)) {
            col[x] = y;
            if (solveNQueens(x + 1)) {
                return true; // Đã tìm được cách đặt quân hậu
            }
            // Nếu không, thử đặt quân hậu vào ô tiếp theo
        }
    }
    return false; // Không thể đặt quân hậu vào hàng này
}
int main() {
    cin >> n;
    col.resize(MAX_N);
    if (!solveNQueens(1)) {
        cout << -1 << endl;
    }
}
```

Bài 2: Cho ma trận a: mxn (1<=m,n<=20). Mỗi phần tử trên ma trận chỉ nhận 2 giá trị 0 hoặc 1 với ý nghĩa: a[i][j] = 1 nếu có thể đi qua ô (i,j) trên ma trận, a[i][j] = 0 nếu tại ô (i,j) có chướng ngại vật không thể đi qua. Cho trước m,n và giá trị các phần tử của a. Hãy liệt kê các đường đi từ ô (1,1) đến ô (m,n), với

mỗi lần đi chỉ được di chuyển sang phải hoặc xuống dưới. (a[1][1] và a[m][n] luôn bằng 1). Mỗi đường đi thỏa mãn ghi ra trên một dòng một xâu chỉ gồm 2 kí tự "R" (sang phải) và "D" (xuống dưới).

Ví dụ: m x n = 3 x 4, ma trận cho là:

1 0 1 0

1 1 1 0

0 1 1 1

Ta được kết quả:

DRRDR

DRDRR

1	0	1	0
1	1	1	0
0	1	1	1

1	0	1	0
1	1	1	0
0	1	1	1

Hướng dẫn:

```cpp
void findPaths(vector<vector<int>>& matrix, int m, int n, int i, int j, string path) {
    // Nếu thấy đường đi đến ô cuối cùng thì in ra đường đi
    if (i == m - 1 && j == n - 1) {
        cout << path << endl;
        return;
    }
    // Di chuyển sang phải nếu có thể
    if (j + 1 < n && matrix[i][j + 1] == 1) {
        findPaths(matrix, m, n, i, j + 1, path + "R");
    }
    // Di chuyển xuống dưới nếu có thể
    if (i + 1 < m && matrix[i + 1][j] == 1) {
        findPaths(matrix, m, n, i + 1, j, path + "D");
    }
}

int main() {
    int m, n;
    cin >> m >> n;
    vector<vector<int>> matrix(m, vector<int>(n));
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {
            cin >> matrix[i][j];
        }
    }
    findPaths(matrix, m, n, 0, 0, "");
}
```

Bài 3: [*https://www.spoj.com/PTIT/problems/BCTSP/*] Cho n thành phố đánh số từ 1 đến n và các tuyến đường giao thông hai chiều giữa chúng, mạng lưới giao thông này được cho bởi mảng C[1…n, 1…n] ở đây C[i][j] = C[j][i] là chi phí đi đoạn đường trực tiếp từ thành phố I đến thành phố j.

Một người du lịch xuất phát từ thành phố 1, muốn đi thăm tất cả các thành phố còn lại mỗi thành phố đúng 1 lần và cuối cùng quay lại thành phố 1. Hãy chỉ ra chi phí ít nhất mà người đó phải bỏ ra.

Ví dụ: n = 4, ma trận C là:

0 20 35 10

20 0 90 50

35 90 0 12

10 50 12 0

Kết quả chi phí ít nhất người ấy phải bỏ ra là 117.

- [Đầu vào]: integer n: số thành phố (n <= 15); n dòng sau, mỗi dòng chứa n số nguyên thể hiện cho mảng 2 chiều C.
- [Đầu ra] integer: chi phí ít nhất người ấy phải bỏ ra.

Lý thuyết:

Trong lập trình cũng như trong thực tế, chắc hẳn các bạn đều đã gặp những bài toán với yêu cầu tìm kết quả tốt nhất thỏa mãn một hoặc một số điều kiện nào đó.

Có rất nhiều bài toán như vậy trong Tin học, và chúng được gọi là các **bài toán tối ưu**. Thông thường, người ta sẽ hay nghĩ đến phương pháp Quy hoạch động khi giải các bài toán tối ưu, tuy nhiên, phương pháp này chỉ có thể áp dụng nếu như bài toán đang xét có bản chất đệ quy (sẽ nói ở phần sau). Thực tế có nhiều bài toán tối ưu không có thuật toán nào thực sự hữu hiệu để giải quyết, mà vẫn phải sử dụng mô hình xem xét tất cả các phương án rồi đánh giá chúng để chọn ra phương án tốt nhất.

Phương pháp **Nhánh và Cận** (*Branch and Bound*) chính là một phương pháp cải tiến từ phương pháp Quay lui, được sử dụng để tìm nghiệm của bài toán tối ưu.

Các bước của một bài toán Quay lui – Nhánh cận:

- Bước đầu tiên của phương pháp vẫn giống với ý tưởng của quay lui: Tìm cách biểu diễn nghiệm của bài toán dưới dạng một $(x_1, x_2, \ldots, x_n)$
- Bước tiếp theo sẽ hơi khác một chút: Nếu như ở phương pháp Quay lui, chỉ cần tuần tự chọn các ứng cử viên cho từng thành phần của vector nghiệm, thì ở phương pháp Nhánh và cận, mỗi nghiệm $X=(x_1, x_2, \ldots, x_n)$ của bài toán sẽ được đánh giá **độ tốt** bằng một hàm $f(X)$. Vì đây là bài toán tối ưu, nên mục tiêu của chúng ta là đi tìm nghiệm có hàm $f(X)$ tốt nhất, thường là **lớn nhất** hoặc **nhỏ nhất**.
- Bước thứ 3 là xây dựng nghiệm của bài toán.

Bằng phương pháp trên, ta sẽ loại bỏ được những nhánh không cần thiết để không duyệt vào các phương án đó, từ đó việc tìm ra nghiệm tối ưu sẽ nhanh hơn. Tuy nhiên, việc đánh giá được "độ tốt" của các nghiệm mở rộng không phải việc đơn giản, nhưng nếu làm được như vậy thì giải thuật sẽ thực thi nhanh hơn nhiều so với quay lui.

Ở bài này, ta có thể áp dụng Nhánh và Cận để giảm độ phức tạp khi chỉ sử dụng quay lui bình thường như sau:

- Gọi chi phí tốt nhất hiện tại là best_cost.
- Với mỗi bước thử chọn x_i, kiểm tra xem chi phí đường đi tính tới lúc đó có lớn hơn hoặc bằng chi phí tốt nhất hiện tại hay không. Nếu đã lớn hơn thì chọn ngay giá trị khác cho x_i, bởi vì có đi tiếp theo nhánh này cũng sẽ chỉ tạo ra chi phí lớn hơn mà thôi.
- Tới khi chọn được một giá trị x_n thì cần kiểm tra xem chi phí tới x_n cộng thêm chi phí từ x_n về 1 có tốt hơn chi phí tốt nhất hiện tại không? Nếu có thì cập nhật lại cách đi tốt nhất.

Hướng dẫn:

```cpp
#include <bits/stdc++.h>
#define int long long
#define task "tsp."
#define inf 1e9 + 7

using namespace std;

const int maxn = 21;
int n, current_cost, best_cost;
int visited[maxn], x_best[maxn], x[maxn], c[maxn][maxn];

void enter()
{
    cin >> n;

    for (int i = 1; i <= n; ++i)
        for (int j = 1; j <= n; ++j)
            cin >> c[i][j];

    // Khởi tạo trước thành phố đầu tiên là 1, đồng thời đánh dấu nó đã thăm.
    x[1] = 1;
    visited[1] = 1;

    // Khởi tạo chi phí tối ưu bằng +oo, giả sử phương án hiện tại đang rất tệ.
    best_cost = inf;
}

// Cập nhật kết quả tốt nhất.
void update_best_solution(int current_cost)
{
    if (current_cost + c[x[n]][1] < best_cost)
    {
        best_cost = current_cost + c[x[n]][1];

        for (int i = 1; i <= n; ++i)
            x_best[i] = x[i];
    }
}
```

```cpp
// In ra phương án tốt nhất tìm được.
void print_best_solution()
{
    cout << best_cost << endl;
    for (int i = 1; i <= n; ++i)
        cout << x_best[i] << "->";
    cout << 1;
}

// Giải thuật nhánh và cận.
void branch_and_bound(int i)
{
    if (current_cost >= best_cost)
        return;

    for (int j = 2; j <= n; ++j)
        if (!visited[j])
        {
            visited[j] = 1;
            x[i] = j;
            current_cost += c[x[i - 1]][j];

            // Đã sinh xong một cấu hình, cập nhật chi phí tốt nhất.
            if (i == n)
                update_best_solution(current_cost);
            // Chưa sinh xong, tiếp tục sinh thành phần tiếp theo với chi phí tăng thêm.
            else
                branch_and_bound(i + 1);

            visited[j] = 0;
            current_cost -= c[x[i - 1]][j];
        }
}

main()
{
    ios_base::sync_with_stdio(false);
    cin.tie(nullptr);

    enter();
    branch_and_bound(2);
    print_best_solution();

    return 0;
}
```

VI. Chia để trị

Chia để trị (*Divide and Conquer*) là một trong các thiết kế giải thuật rất phổ biến thường được sử dụng trong những bài toán có kích thước lớn. Trong lập trình thi đấu, chúng ta thường nghe đến chiến lược này gắn liền cùng với những thuật toán như Sắp xếp nhanh (*Quicksort*), Sắp xếp trộn (*Mergesort*),..., hay thuật toán lũy thừa a^n. Tuy nhiên, đó chỉ là bề nổi của vấn đề. Ý nghĩa thực sự của Chia để trị nằm ở

chỗ, nó giúp cho chúng ta giải quyết những bài toán lớn trong thời gian nhỏ, mà kĩ thuật lập trình lại không quá phức tạp.

Tư tưởng của chiến lược Chia để trị có thể được chia làm ba bước theo đúng tên gọi của nó như sau:

- Divide: Chia bài toán lớn ban đầu thành các bài toán nhỏ.
- Conquer: Gọi đệ quy về các bài toán con tới khi thu được bài toán con hoặc đã có lời giải, hoặc có thể giải một cách dễ dàng.
- Combine: Kết hợp nghiệm của các bài toán con lại để thu được nghiệm của bài toán lớn hơn, từ đó tiến tới nghiệm của bài toán gốc.

Nghe qua thì có vẻ khá giống với Giải thuật đệ quy đúng không nào? Kỳ thực, chiến lược Chia để trị chính là một sự phát triển của Giải thuật đệ quy, áp dụng kĩ thuật đệ quy để giải bài toán một cách nhanh hơn, hiệu quả hơn. Đối với Giải thuật đệ quy, việc giải các bài toán con có thể sinh ra bất lợi về mặt thời gian thực thi do gặp phải rất nhiều bài toán con gối nhau (trùng lặp), nhưng với chiến lược Chia để trị thì điều đó không xảy ra, vì những bài toán con trong chiến lược này thường được thu nhỏ với tốc độ rất nhanh, có thể chỉ tương đương với hàm log. Ngoài ra, do đặc điểm của phương pháp có sử dụng tới lời gọi đệ quy, nên các bài toán áp dụng Chia để trị cũng không có hoặc xuất hiện rất ít những bài toán con gối nhau gây tốn kém thời gian tính toán.

Chiến lược Chia để trị có thể được mô tả bằng một mô hình đệ quy như sau:

```
divide_and_conquer(A, x) // Tìm nghiệm x của bài toán A.
{
   if (A_đủ_nhỏ)
      {Giải_bài_toán_A};
   else
   {
      {Phân_rã_A_thành_các_bài_toán_con: A_1, A_2,..., A_m}
      for (i = 1 to m)
         divide_and_conquer(A_i, x_i); // Gọi đệ quy để tìm nghiệm x_i của
bài toán con A_i.

      {Kết_hợp_nghiệm_của_m_bài_toán_con                    ->
Thu_được_nghiệm_bài_toán_A}
   }
}
```

Bài 1: Cho một mảng A[] gồm các số nguyên, hãy viết chương trình tìm hiệu lớn nhất giữa hai phần tử bất kỳ sao cho phần tử lớn hơn xuất hiện sau phần tử nhỏ hơn. Nói cách khác, chúng ta cần tìm max(A[j] - A[i]), trong đó A[j] > A[i] và j > i.

Ví dụ:

Với A[] = [1, 4, 9, 5, 3, 7], kết quả: 8 (Hiệu lớn nhất tìm được là giữa 9 và 1)

Với A[] = [9, 8, 6, 3, 2], kết quả là -1 (vì đây là mảng giảm dần)

- [Đầu vào] array integer A
- [Đầu ra] interger: Hiệu lớn nhất của mảng

Lý thuyết:

Chúng ta có thể giải quyết vấn đề này bằng cách sử dụng phương pháp chia để trị. Giả sử chúng ta chia mảng thành hai phần bằng nhau và tính toán đệ quy độ chênh lệch tối đa giữa phần bên trái và bên phải. Khi đó, chênh lệch lớn nhất của toàn bộ mảng sẽ là giá trị lớn nhất trong ba khả năng sau:

- Hiệu lớn nhất của phần bên trái (Bài toán con 1).
- Hiệu lớn nhất của phần bên phải (Bài toán con 2).
- Chênh lệch tối đa qua phần bên trái và bên phải.

Hướng dẫn:

```cpp
int maxDifference(const vector<int>& arr, int left, int right) {
    if (left == right) {
        return 0;
    }

    int mid = (left + right) / 2;

    int leftMaxDiff = maxDifference(arr, left, mid);
    int rightMaxDiff = maxDifference(arr, mid + 1, right);

    int maxLeft = arr[mid];
    int minRight = arr[mid + 1];

    for (int i = left; i <= mid; i++) {
        maxLeft = max(maxLeft, arr[i]);
    }

    for (int i = mid + 1; i <= right; i++) {
        minRight = min(minRight, arr[i]);
    }

    return max(leftMaxDiff, max(rightMaxDiff, minRight - maxLeft));
}
```

Bài 2: [*https://leetcode.com/problems/longest-common-prefix/description/*] Viết hàm tìm chuỗi tiền tố chung dài nhất trong một mảng các chuỗi. Nếu không có tiền tố chung, trả về một chuỗi trống "".

Ví dụ: Với a = ["flower","flow","flight"], ta có kết quả là “fl”

- [Đầu vào] string array: mảng xâu ký tự chỉ gồm chữ cái alphabet in thường
- [Đầu ra] string: chuỗi tiền tố chung dài nhất

Lý thuyết:

Mặc dù có thuật toán tốt hơn để giải nó là Tìm kiếm nhị phân, nhưng đây vẫn là một ví dụ rất hay về cách áp dụng Chia để trị:

- Đầu tiên, xét bài toán tìm tiền tố chung dài nhất giữa hai chuỗi x và y: Duyệt qua các kí tự x_i và y_j của hai chuỗi, nếu như xuất hiện một cặp chuỗi x_i != $y_j$thì tiền tố chung dài nhất sẽ là $x_{0...i-1} = y_{0...j-1}$ (vì hai chuỗi sẽ khác nhau khi tồn tại một kí tự khác nhau). Hàm *lcp_two_strings(x, y)* dùng để tìm tiền tố chung dài nhất của hai chuỗi x và y.
- Áp dụng ý tưởng trên với n chuỗi, ta có công thức:

$$LCP(s_1, s_2, ..., s_n)=LCP(...LCP(LCP(s_1, s_2), s_3),...,s_n)$$

- Kế đến, chia đôi tập hợp các chuỗi ban đầu, tìm tiền tố chung dài nhất của hai nửa rồi gộp lại với nhau để tạo được tiền tố chung dài nhất của tất cả các chuỗi. Việc tìm tiền tố chung dài nhất của hai tập hợp con trái phải lại được thực hiện bằng cách chia đôi tập hợp đó ra cho tới khi thu được tập hợp chỉ gồm một chuỗi duy nhất. Hàm *lcp_n_strings(a, l, r)* dùng để tìm tiền tố chung dài nhất của tập hợp các chuỗi $\{a_l, a_{l+1}, ..., a_r\}$.

Hướng dẫn:

```
string lcp_two_strings(string x, string y)
{
    int lcp_length = 0;
    for (int i = 0, j = 0; i < x.size() && j < y.size(); ++i, ++j)
    {
        if (x[i] != y[j])
            break;
        ++lcp_length;
    }

    return x.substr(0, lcp_length);
}

string lcp_n_strings(vector<string> a, int l, int r)
{
    if (l == r)
        return a[l];
    if (l < r)
    {
        int mid = (l + r) / 2;
        string lcp_left = lcp_n_strings(a, l, mid);
        string lcp_right = lcp_n_strings(a, mid + 1, r);

        return lcp_two_strings(lcp_left, lcp_right);
    }
}
```

VII. Quy hoạch động

Gần một nửa các bài thi trong các cuộc thi code cần đến **quy hoạch động** (*Dynamic Programming*). Tất nhiên, có những cách khác để giải bài toán đó. Nhưng vì các cuộc thi code đều có giới hạn về thời gian, cũng như bộ nhớ của chương trình, nên một thuật toán hiệu quả là cực kỳ cần thiết. Và trong những trường hợp như vậy, quy hoạch động là một trong những thuật toán xuất hiện nhiều nhất.

Thuật toán quy hoạch động được ưa chuộng bởi vì ban đầu, bài toán có muôn hình vạn trạng và bạn phải suy nghĩ rất nhiều mới tìm ra được lời giải. Không có một công thức chuẩn mực nào áp dụng được cho mọi bài toán. Bởi vì sự phổ biến của nó, bạn bắt buộc phải cực kỳ thuần thục thuật toán này nếu muốn có kết quả tốt trong các cuộc thi.

Khi nào thì chúng ta cần đến quy hoạch động? Đó là một câu hỏi rất khó trả lời. Không có một công thức nào cho các bài toán như vậy. Tuy nhiên, có một số tính chất của bài toán mà bạn có thể nghĩ đến quy hoạch động. Dưới đây là hai tính chất nổi bật nhất trong số chúng:

- Bài toán có các **bài toán con gối nhau**: Tương tự như thuật toán chia để trị, quy hoạch động cũng chia bài toán lớn thành các bài toán con nhỏ hơn. Quy hoạch động được sử dụng khi các bài toán con này được gọi đi gọi lại. Phương pháp quy hoạch động sẽ lưu kết quả của bài toán con này, và khi được gọi, nó sẽ không cần phải tính lại, do đó làm giảm thời gian tính toán.
- Bài toán có **cấu trúc con tối ưu**: Cấu trúc con tối ưu là một tính chất là lời giải của bài toán lớn sẽ là tập hợp lời giải từ các bài toán nhỏ hơn.

Bài 1: [Làm lại] Một khách hàng muốn rút số tiền T từ một cây ATM bên đường. Bên trong cây ATM hiện đang có n tờ tiền có mệnh giá $a_1,a_2,\ldots,a_n$. Hãy tìm số tờ tiền tối thiểu cần sử dụng để ATM trả tiền cho khách hàng?

Ví dụ: n = 5, T = 10, arr = {1, 4, 2, 3, 6} thì kết quả là 2 (tờ 4 và tờ 6)

Ví dụ: n = 6, T = 100, arr = {50, 20, 20, 20, 20, 20} thì kết quả là 5 (5 tờ 20)

- [Đầu vào] integer: 1 <= n, T <= 1000; 1 <= a_i <= 1000.
- [Đầu ra] integer: In ra số nguyên duy nhất là số tờ tiền tối thiểu cần sử dụng. Nếu không có phương án trả tiền thì in ra −1.

Hướng dẫn:

```cpp
int minNotes(int T, vector<int>& arr) {
    int n = arr.size();

    // Khởi tạo mảng lưu số tờ tiền tối thiểu cần sử dụng
    vector<int> dp(T + 1, INT_MAX);

    // Trường hợp cơ bản: không cần tờ tiền nào để trả 0 đồng
    dp[0] = 0;

    // Duyệt qua các mệnh giá tiền và cập nhật mảng dp
    for (int i = 0; i < n; ++i) {
        for (int j = T; j >= arr[i]; --j) {
            // Cập nhật số tờ tiền tối thiểu cần sử dụng
            if (dp[j - arr[i]] != INT_MAX) {
                dp[j] = min(dp[j], dp[j - arr[i]] + 1);
            }
        }
    }

    // Trả về số tờ tiền tối thiểu cần sử dụng để trả số tiền T
    return dp[T] == INT_MAX ? -1 : dp[T];
}
```

Ở phần trước, chúng ta đã giải quyết bài tập này với giải thuật tham lam, tuy nhiên nó vẫn chưa được tối ưu về mặt kết quả. Vì vậy chúng ta sẽ áp dụng Quy hoạch động để giải bài tập này.

Dễ nhận thấy trong bài này, bài toán con gối nhau là khi ta cần lấy một số tiền k, ta sẽ giải quyết các bài con là tìm cách lấy các số tiền < k. Cấu trúc con tối ưu tất nhiên là phải giải quyết các bài toán con trên sao cho lấy ít số tờ tiền nhất.

Công thức quy hoạch động của bài toán như sau:

- Đầu tiên, ta khởi tạo một mảng dp có kích thước là (T + 1), trong đó dp[i] sẽ lưu số tờ tiền tối thiểu cần sử dụng để trả số tiền i.

  Ta gán dp[0] = 0, vì không cần tờ tiền nào để trả số tiền 0.

- Sau đó, ta duyệt qua từng mệnh giá tiền trong mảng arr và cập nhật mảng dp. Trong quá trình duyệt này, ta tính toán số tờ tiền tối thiểu cần sử dụng để trả các số tiền từ arr[i] đến T.

Cụ thể, với mỗi mệnh giá tiền arr[i], ta duyệt qua các giá trị j từ T đến arr[i], và cập nhật dp[j] theo công thức:

dp[j] = min(dp[j], dp[j - arr[i]] + 1)

Nghĩa là số tờ tiền tối thiểu cần sử dụng để trả số tiền j sẽ là số tờ tiền tối thiểu cần sử dụng để trả số tiền (j - arr[i]) cộng với 1 tờ tiền có mệnh giá arr[i].

- Cuối cùng, sau khi duyệt qua tất cả các mệnh giá tiền, giá trị dp[T] sẽ chứa số tờ tiền tối thiểu cần sử dụng để trả số tiền T. Nếu giá trị này bằng INT_MAX, tức là không thể tạo ra số tiền T từ các mệnh giá tiền đã cho, ta trả về -1. Ngược lại, ta trả về giá trị dp[T].

Bài 2: [*https://leetcode.com/problems/longest-increasing-subsequence/*]Cho một mảng số nguyên, trả về độ dài của dãy con tăng đơn điệu dài nhất. Dãy con tăng đơn điệu là dãy gồm các phần tử $a[i_1]$, $a[i_2]$, ... $a[i_k]$ với $i_1<i_2<...<i_k$ và $a[i_1]<a[i_2]<...<a[i_k]$.

Ví dụ:

Với nums = [10,9,2,5,3,7,101,18] ta có kết quả = 4 (2,3,7,101)

Với nums = [7,7,7,7,7,7,7] ta có kết quả = 1

- [Đầu vào] array integer: 1 <= nums.length <= 2500, -10^4 <= nums[i] <= 10^4
- [Đầu ra] integer: độ dài dãy con tăng đơn điệu dài nhất.

Hướng dẫn:

Do cấu trúc con tối ưu và thuộc tính bài toán con gối nhau, chúng ta cũng có thể sử dụng Quy hoạch động để giải quyết bài toán.

Gọi lengthOfLIS(i) là độ dài của dãy con tăng đơn điệu dài nhất kết thúc tại vị trí i trong mảng nums. Để tính lengthOfLIS(i), chúng ta sẽ xem xét tất cả các vị trí j (trước i) và kiểm tra nếu nums[j] < nums[i]. Điều này có nghĩa là chúng ta có thể mở rộng dãy con tăng đơn điệu từ j đến i. Từ đó, chúng ta có thể tìm lengthOfLIS(j) và cộng thêm 1 vào nó để tính độ dài của dãy con tăng kết thúc tại i. Do đó, công thức đệ quy có thể được viết như sau:

$$\text{lengthOfLIS(i)} = \begin{cases} 1\ (nếu\ i = 0) \\ 1 + \max(\text{lengthOfLIS(j)})\ (nếu\ 0 \le j < i\ và\ nums[j] < nums[i]) \end{cases}$$

```cpp
int lengthOfLIS(vector<int>& nums) {
    int n = nums.size();
    if (n == 0) return 0;

    // lengths[i] lưu độ dài của dãy con tăng đơn điệu dài nhất kết thúc ở nums[i]
    vector<int> lengths(n, 1);

    for (int i = 1; i < n; ++i) {
        for (int j = 0; j < i; ++j) {
            if (nums[i] > nums[j]) {
                lengths[i] = max(lengths[i], lengths[j] + 1);
            }
        }
    }

    // Trả về độ dài lớn nhất tìm được
    return *max_element(lengths.begin(), lengths.end());
}
```

Tuy nhiên, QHĐ không phải là "thần thánh" bởi vì nếu áp dụng cách này, độ phức tạp của mã sẽ là **O(n^2)** - vẫn là một con số cần phải tối ưu hơn nữa.

Như đã nói nhiều lần trong tài liệu này, không có giải thuật nào là tốt với tất cả mọi bài và cũng không có giải thuật nào là vô dụng cả. Vì vậy tùy vào từng bài toán, ta phải có cách nhìn nhận và phân tích đúng để đưa ra phương án giải thuật hoặc cao hơn là kết hợp nhiều thuật toán với nhau nhằm đưa ra cách làm tối ưu nhất.

Gợi ý: Để tối ưu bài này, ta có thể áp dụng Tham Lam và Tìm kiếm nhị phân. Điều này giúp giảm độ phức tạp của thuật toán xuống O(n logn).

Bài 3: [*https://www.spoj.com/PTIT/problems/BCCAITUI/*] Trong siêu thị có n gói hàng (n <= 100), gói hàng thứ i có trọng lượng là Wi <= 100 và giá trị Vi <= 100. Một tên trộm đột nhập vào siêu thị, sức của tên trộm không thể mang được trọng lượng vượt quá M (M <= 100). Hỏi tên trộm sẽ lấy đi những gói hàng nào để được tổng giá trị lớn nhất.

Ví dụ: n = 3, M = 4, Wi và Vi lần lượt là:

3 4

1 4

2 5

3 6

Kết quả là 10 (Tên trộm sẽ lấy gòi hàng thứ 2 và thứ 4)

- [Đầu vào] integer, array integer: n <= 100, M <= 100, Wi, Vi <= 100
- [Đầu ra] integer: Giá trị lớn nhất tên trộm lấy được.

Gợi ý: Nhìn là biết phải tìm công thức Quy hoạch động rồi đúng không?

PHẦN 3: CẤU TRÚC DỮ LIỆU

Trong khóa học C++ của CLB Công nghệ thông tin Đại học Kinh tế Quốc dân, các thành viên đã dược học về các STL C++ như Vector, Set hay Map,… Với tài liệu của này, chúng ta sẽ học thêm 2 cấu trúc dữ liệu quan trọng nữa đó là Stack (Ngăn xếp) và Queue (Hàng chờ)!

I. Stack (Ngăn xếp)

Stack là một cấu trúc dữ liệu hoạt động theo nguyên tắc **Last In First Out** (LIFO). Hiểu đơn giản là phần tử sẽ được thêm vào cuối stack và khi lấy ra ta cũng sẽ lấy phần tử cuối stack (phần tử được thêm vào gần nhất).

Một stack sẽ hỗ trợ các thao tác cơ bản sau:

- Thêm phần tử vào cuối stack.
- Loại bỏ phần tử cuối ra khỏi stack.
- Lấy giá trị cuối trong stack.
- Lấy kích thước stack.

Thông thường để thêm stack vào chương trình, chúng ta sẽ thêm thư viện như sau:

#include<stack>

Tuy nhiên, mình khuyến nghị sử dụng header sau:

#include<bits/stdc++.h>

Header này sẽ giúp chúng ta thêm tất cả các thư viện về các cấu trúc dữ liệu mà chúng ta thường sử dụng nhất.

Ta sẽ khai báo stack như sau:

stack <{kiểu dữ liệu}> {tên stack};

Ví dụ: *stack<int> myStack;*

Các phương thức cơ bản trong STL stack của C++:

- push: Thêm phần tử vào cuối stack
- pop: Loại bỏ phần tử cuối stack
- top: Trả về giá trị là phần tử cuối trong stack
- size: Trả về giá trị nguyên là số phần tử đang có trong stack

- empty: Trả về một giá trị bool, true nếu stack rỗng, false nếu stack không rỗng

Các phương thức trên sẽ đều mất độ phức tạp O(1).

Bài 1: [*https://www.spoj.com/PTIT/problems/PTIT123J/*]Cho các đoạn văn chứa các dấu ngoặc, có thể là ngoặc đơn đơn ("()") hoặc ngoặc vuông ("[]"). Một đoạn văn đúng là đoạn mà với mỗi dấu mở ngoặc thì sẽ có dấu đóng ngoặc tương ứng và đúng thứ tự. Nhiệm vụ của bạn kiểm tra xem đoạn văn có đúng hay không.

Ví dụ:

Với s = "So when I die (the [first] I will see in (heaven) is a score list)." → yes

Với s = "Half Moon tonight (At least it is better than no Moon at all]." → no

- [Đầu vào] Gồm nhiều bộ test, mỗi bộ test trên một dòng chứa đoạn văn cần kiểm tra có thể bao gồm: các kí tự trong bảng chữ cái tiếng Anh, dấu cách, và dấu ngoặc (ngoặc đơn hoặc ngoặc vuông). Kết thúc mỗi bộ test là một dấu chấm. Mỗi dòng có không quá 100 kí tự. Dữ liệu kết thúc bởi dòng chứa duy nhất một dấu chấm.
- [Đầu ra] string array: dòng thứ i in ra "yes" hoặc "no" tương ứng với kết quả của test thứ i đúng hay sai.

Hướng dẫn:

Sử dụng cấu trúc Stack để lưu và kiểm tra các dấu ngoặc xuất hiện trong xâu.

```cpp
int main()
{
    while (1)
    {
        string s;
        getline(cin, s);
        if (s == ".") break;

        stack<char> st;
        st.push('@');

        for (int i = 0; i < s.length(); i++)
        {
           if (s[i] == ')')
                {
                    if (st.top() == '(') st.pop();
                    else st.push(s[i]);
                }
            else if (s[i] == ']')
                {
                    if (st.top() == '[') st.pop();
                    else st.push(s[i]);
                }
            else if (s[i] == '(' || s[i] == '[') st.push(s[i]);
        }

        if (st.top() == '@') cout<<"yes\n";
        else cout<<"no\n";
    }
}
```

Bài 2:[*https://leetcode.com/problems/decode-string/*] Cho một chuỗi được mã hóa, trả về chuỗi đã giải mã của nó. Quy tắc mã hóa là: k[encoded_string], trong đó encoded_string bên trong dấu ngoặc vuông được lặp lại chính xác k lần.

Ví dụ:

Với s = "3[a]2[bc]" thì kết quả là "aaabcbc"

Với s = “3[a2[c]]” thì kết quả là "accaccacc"

- [Đầu vào] string s: 1 <= s.length <= 30.
- [Đầu ra] string: kết quả sau khi giải mã xâu s.

Gợi ý: Sử dụng stack để giải bài tập này.

II. Queue (Hàng chờ)

Queue là một cấu trúc dữ liệu dùng để lưu giữ các đối tượng theo cơ chế FIFO (viết tắt từ tiếng Anh: First In First Out), nghĩa là "vào trước ra trước".

Một cấu trúc queue có các chức năng cơ bản như sau:

- Lấy ra phần tử đầu queue.
- Thêm một phần tử vào cuối của queue.
- Kiểm tra xem queue hiện tại có đang rỗng hay không.
- Truy nhập phần tử ở đầu queue.

Tương tự như Stack, ta sẽ khai báo 1 Queue như sau:

queue <{kiểu dữ liệu}> {tên queue};

Ví dụ: *stack<int> myQueue;*

Cấu trúc queue trong C++ STL hỗ trợ các chức năng sau:

- empty(): Kiểm tra xem hàng đợi hiện tại có đang rỗng hay không.
- size(): Trả về kích thước hàng đợi hiện tại.
- front(): Trả về phần tử đầu tiên của hàng đợi.
- back(): Trả về phần tử cuối cùng của hàng đợi.
- push(): Nạp thêm một phần tử vào hàng đợi, biến cần được khởi tạo trước đó.
- emplace(): Nạp thêm một phần tử vào hàng đợi, có thể khởi tạo biến ngay tại thời điểm nạp.
- pop(): Xóa một phần tử ở đầu của hàng đợi.

Deque

STL C++ còn có một cấu trúc khác khá giống với queue đó là **Deque.**

Cấu trúc deque hỗ trợ tất cả các chức năng mà queue có ở trên, cộng thêm các chức năng của 1 vector. Một vài chức năng nổi bật của deque:

- begin(): Trả về con trỏ ở vị trí đầu tiên của deque.

- end(): Trả về con trỏ ở vị trí cuối cùng của deque.
- size(): Trả về kích thước của deque.
- resize(): Thay đổi kích thước của 1 deque.
- empty(): Kiểm tra xem deque có rỗng hay không.
- operator[]: Truy nhập một phần tử ở một vị trí chỉ định của deque, chức năng này không thể thực hiện được trên queue.
- front(): Trả về phần tử đầu tiên của deque.
- back(): Trả về phần tử cuối cùng của deque.
- push_back(): Thêm 1 phần tử vào cuối deque.
- push_front(): Thêm 1 phần tử vào đầu deque.
- pop_back(): Xóa phần tử ở cuối deque.
- pop_front(): Xóa phần tử ở đầu deque.
- insert(): Thêm 1 phần tử vào 1 vị trí chỉ định của deque.
- erase(): Xóa 1 phần tử ở 1 vị trí chỉ định của deque.
- clear(): Xóa toàn bộ phần tử trong deque.
- emplace_front(), emplace_back(): Tương tự emplace() trong queue.

Như ta có thể thấy, deque là cấu trúc dữ liệu tích hợp từ vector và queue của C++. Ngoại trừ việc hỗ trợ lưu dữ liệu dạng FIFO như queue thì deque cũng có thể được sử dụng dưới dạng LIFO (last in first out – vào sau ra trước) và cũng có thể truy nhập một phần tử bất kì. Và ngoài các chức năng của vector, deque còn hỗ trợ xóa và thêm phần tử ở đầu dãy trong O(1), thay vì O(n) như vector.

Vậy nhược điểm của deque với hai CTDL trên là gì? Thực ra là không có, nếu có thì chẳng qua là nó sinh sau nên không được phổ biến bằng hai CTDL trên mà thôi. Tất cả các bài cần sử dụng queue và vector (thậm chí cả stack) đề có thể sử dụng deque để thay thế.

Cách khai báo: Khai bào tương tự như queue và stack.

Priority Queue

Vẫn tiếp tục là một biến thể khác của queue. **Priority Queue** là một cấu trúc dữ liệu mà các phần tử được quản lý sẽ có "độ ưu tiên" khác nhau gắn với từng phần tử. Phần tử có thứ tự ưu tiên cao hơn trong Priority Queue sẽ được xếp lên trước và truy vấn trước.

Đơn giản hơn, Priority Queue là một cấu trúc cho phép nó tự động sắp xếp các phần tử của nó.

Priority Queue trong C++ sẽ có một số phương thức cơ bản sau:

- push: Thêm phần tử vào priority_queue
- pop: Loại bỏ phần tử đầu tiên (có độ ưu tiên cao nhất) trong priority_queue
- top: Trả về giá trị là phần tử đầu tiên (có độ ưu tiên cao nhất) trong priority_queue
- size: Trả về số nguyên là số phần tử (kích thước) của priority_queue
- empty: Trả về giá trị bool, true nếu priority_queue rỗng, false nếu priority_queue không rỗng

Cách khai báo:

priority_queue <{kiểu dữ liệu}, {class container}, {class compare}> {tên priority_queue};

- Container là một đối tượng cho phép lưu trữ các đối tượng khác. Ví dụ như vector, stack, queue, deque, … đều là các class container. Mặc định, vector là container của priority_queue
- Compare thường sẽ là một phép toán trong thư viện functional. Một số phép toán trong thư viện này là less, greater, less_equal, … Phép toán mặc định trong priority_queue là less. Đối với các kiểu dữ liệu mặc định của C++, các phép toán này đã được xây dựng sẵn. Tuy nhiên, với các class hay struct mà người dùng tạo, ta sẽ phải tự định nghĩa các phép toán này.

Bài 1: [*https://www.spoj.com/PTIT/problems/P175SUME/*] Do những ngày hè quá nóng bức và nhàm chán nên Tide đã nghĩ ra một trò chơi khá thú vị với queue. Ban đầu trong queue có 5 số 1, 2, 3, 4, 5 với mỗi lượt chơi Tide sẽ xóa phần tử ở đầu queue và cho 2 phần tử đó xuống cuối của queue và cứ tiếp tục cho đến khi Tide cảm thấy mệt và không chơi được nữa.

Các bạn hãy giúp Tide xác định xem số đầu tiên của queue tại lượt chơi thứ N nhé.

Ví dụ tại lượt chơi thứ nhất trạng thái của queue là 1, 2, 3 ,4 ,5

Tại lượt chơi thứ 2 trạng thái của queue là 2, 3, 4, 5, 1, 1

- [Đầu vào] integer n: 1 <= n <= 10^9
- [Đầu ra] integer: số đầu tiền của queue sau n lần chơi.

Gợi ý: Áp dụng queue trong STL C++ để giải bài tập.

Bài 2: [*https://www.hackerearth.com/practice/data-structures/queues/basics-of-queues/practice-problems/algorithm/disk-tower-b7cc7a50/*] Nhiệm vụ của bạn là xây dựng một tòa tháp trong N ngày bằng cách tuân theo các điều kiện sau:

- **Mỗi ngày bạn được cung cấp một đĩa có kích thước riêng biệt.**
- **Đĩa có kích thước lớn hơn được đặt về phía đáy tháp. Đĩa có kích thước nhỏ hơn được đặt về phía đỉnh tháp. (Giống với Tháp Hà Nội)**

Bạn không thể đặt một đĩa mới lên đỉnh tháp cho đến khi tất cả các đĩa lớn hơn đĩa đó phải được đặt xong hết phía bên dưới.

In N dòng biểu thị kích thước đĩa có thể được đặt trên tháp vào ngày thứ i.

Ví dụ:

Với N = 5 và arr = {4, 5, 1, 2, 3}

Kết quả sẽ là:

(N1)

(N2) 5 4

(N3)

(N4)

(N5) 3 2 1

- [Đầu vào] integer, integer array: 1 < N < 10^6, 1 < arr[i] < N.
- [Đầu ra] N dòng, dòng thứ i là số đĩa đặt được vào ngày i.

Gợi ý: Áp dụng Priority Queue vào bài tập này.

PHẦN 4: ĐỒ THỊ VÀ DUYỆT ĐỒ THỊ

I. Đồ thị

Từ phần này, chúng ta sẽ tập trung chủ yếu vào lý thuyết để các bạn có thể nắm rõ các khái niệm về giải thuật trên đồ thị cũng như các cấu trúc dữ liệu dùng với duyệt đồ thị trước khi tiến hành áp dụng những phương pháp duyệt cây vào lập trình thi đấu.

1. Định nghĩa

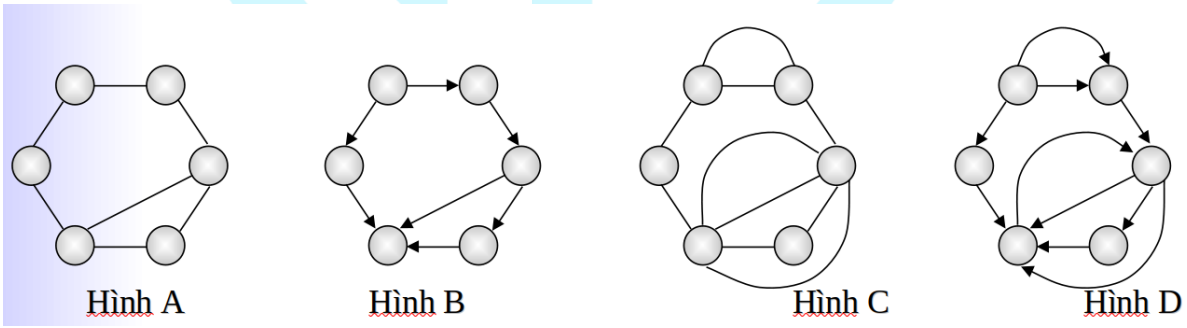
Định nghĩa đồ thị:

- Là một cấu trúc rời rạc: G = <V, E>.
- V là tập các đỉnh (vertices)
- E là tập các cạnh (Edges) - tập các cặp (u,v) mà u,v là hai đỉnh thuộc V.

Dựa vào đặc tính của tập E:

- G là đơn đồ thị nếu giữa hai đỉnh u và v bất kì có nhiều nhất một cạnh.
- G là đa đồ thị nếu giữ hai đỉnh u và v bất kì có thể có nhiều hơn một cạnh.
- G-undirected (đồ thị vô hướng) graph nếu (u,v)=(v,u) gọi là cạnh
- G-directed graph (đcó hướng nếu (u,v)<>(v,u), gọi là các cung.
- Nếu là đồ thị có trọng số: Là đồ thị trê mỗi cung (cạnh) có kèm theo một thông số nào đó

Ví dụ:



Hình A: Đơn đồ thị, vô hướng

Hình B: Đơn đồ thị, có hướng

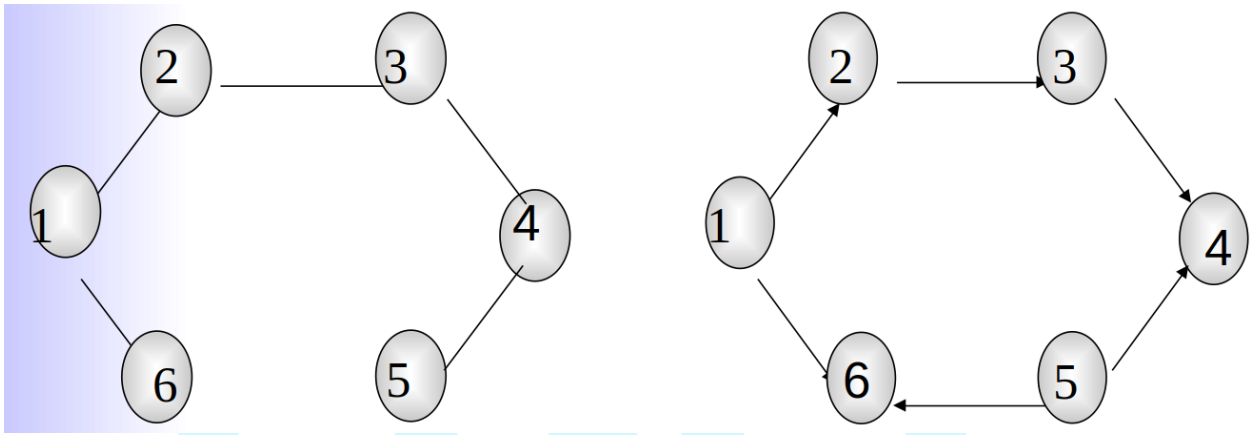
Hình C: Đa đồ thị, vô hướng

Hình D: Đa đồ thị, có hướng

2. Các khái niệm

Cạnh liên thuộc, đỉnh kề, bậc

- Nếu e = (u,v) thì u,v kề nhau (adjacent) và e là liên thuộc với u và v
- Nếu e là một cung thì ta nói u nối tới v và v nối từ u. Cung e đi ra khỏi đỉnh u (đỉnh đầu) và đi vào đỉnh v (đỉnh cuối).
- Bậc (degree) của v (deg(v)) là số cạnh liên thuộc với v.



Đường đi và chu trình

- Một đường đi P = ($v_1 \ldots v_p$) mà cạnh v_{i-1}, v_i thuộc E với mọi i (1<=i<=p).
- P gọi là đơn giản nếu tất cả các đỉnh trên đường đi đều phân biệt.
- Một đường đi con của P là một đoạn liên tục dọc theo P.
- P gọi là chu trình (circuit) nếu $v_1 = v_p$
- Một đồ thị vô hướng gọi là liên thông (connected) nếu với mọi cặp đỉnh (u,v) đều có u đến được v.

3. Các phép toán

- Đọc nhãn của đỉnh.
- Đọc trọng số của cạnh.
- Thêm một đỉnh vào đồ thị.
- Thêm một cạnh vào đồ thị.

- Xoá một đỉnh.
- Xoá một cạnh.
- Lần theo các cung trên đồ thị để đi từ đỉnh này sang đỉnh khác.
- FIRST(v): Lấy chỉ số của đỉnh đầu tiên kề với v. Nếu không thì trả về giá trị đặc biệt nào đó (ví dụ $).
- NEXT(v,i): lấy chỉ số của đỉnh nằm sau đỉnh có chỉ số i và kề với v. Nếu không có thì trả về $.
- VERTEX(i) trả về đỉnh có chỉ số i.

4. Biểu diễn đồ thị trong máy tính

Ma trận kề (Adjacency Matrix)

Giả sử $G=(V, E)$ là một đa đồ thị có số đỉnh là N. Coi rằng các đỉnh được đánh số từ 1 tới N. Khi đó, ta có thể biểu diễn đồ thị bằng một ma trận vuông *adj* kích thước $N \times N$, trong đó:

- $adj_{u,v} = 0$, nếu $(u,v) \notin E, u \neq v$.
- $adj_{u,v} = x$, nếu $(u,v) \in E$, $u \neq v$ và x là số lượng cạnh nối giữa u và v.
- Đối với $adj_{u,u}$ với $\forall u : 1 \leq u \leq N$, có thể đặt giá trị tùy theo mục đích, thông thường nên đặt bằng 0.

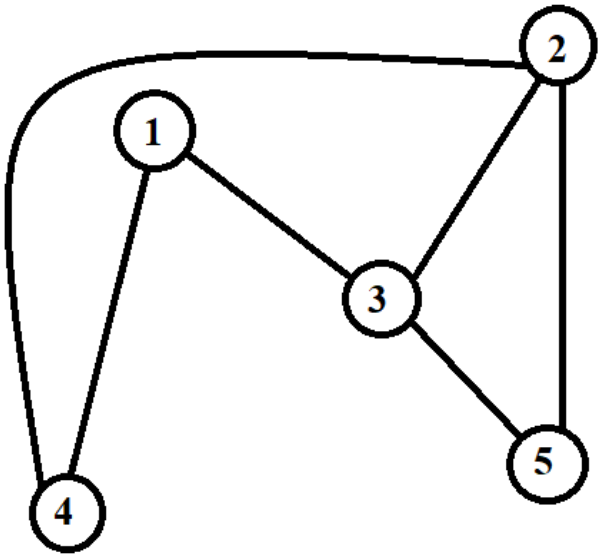
Cài đặt:

```
void enter_adjacency_matrix()
{
    cin >> N >> M; // Nhập số đỉnh và số cạnh của đồ thị.

    for (int i = 1; i <= M; ++i)
    {
        int u, v;
        cin >> u >> v;

        adj[u][v]++; // Tăng số cạnh giữa u và v.
        adj[v][u]++; // Nếu là đồ thị có hướng thì không có dòng này.
    }
}
```

Ví dụ: Đồ $G(V, E)$ dưới đây có 5 đỉnh, 6 cạnh:



Ma trận kề của nó sẽ có dạng như sau:

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

Ưu điểm của ma trận kề:

- Đơn giản, dễ cài đặt.
- Để kiểm tra hai đỉnh u và v có kề nhau hay không, chỉ việc kiểm tra trong $O(1)$ bằng phép so sánh $a_{u,v} \neq 0$.

Nhược điểm của ma trận kề:

- Luôn luôn tiêu tốn N^2 ô nhớ để lưu trữ ma trận kề, dù là trong trường hợp đồ thị ít cạnh hay nhiều cạnh.
- Để xét một đỉnh u kề với những đỉnh nào, buộc phải duyệt toàn bộ các đỉnh v và kiểm tra điều kiện $a_{u,v} \neq 0$. Như vậy kể cả đỉnh u không kề với đỉnh nào, chúng ta vẫn phải duyệt mất $O(N)$ để biết được điều đó.

Phù hợp khi nào: Trong các bài toán đồ thị có số lượng đỉnh ít (thường là không vượt quá 300).

Danh sách cạnh (Edge List)

Trong trường hợp biết trước đồ thị có N đỉnh, M cạnh, ta có thể biểu diễn đồ thị dưới dạng một danh sách lưu các cạnh (u,v) của đồ thị đó (nếu là đồ thị có hướng thì mỗi cặp (u,v) ứng với một cung $(u \rightarrow v)$. Vector hoặc mảng là một kiểu dữ liệu rất phù hợp để lưu trữ danh sách cạnh.

Cài đặt:

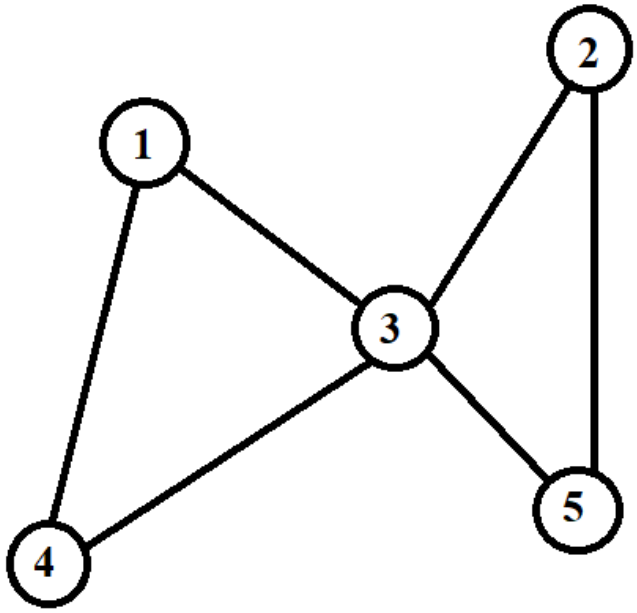
```
vector < pair < int, int > > edge_list; // Danh sách cạnh.

void enter_edge_list()
{
    cin >> N >> M;

    for (int i = 1; i <= M; ++i)
    {
        int u, v;
        cin >> u >> v;

        edge_list.push_back({u, v});
    }
}
```

Ví dụ: Đồ thị G(V,E) dưới đây 5 đỉnh, 6 cạnh theo thứ tự là: (1,3), (1,4), (3,4), (3,2), (5,3), (2,5)



Danh sách cạnh của nó được biểu diễn bằng một vector như sau:

edge_list[0]	{1, 3}
edge_list[1]	{1, 4}
edge_list[2]	{3, 4}
edge_list[3]	{3, 2}
edge_list[4]	{5, 3}
edge_list[5]	{2, 5}

Ưu điểm của danh sách cạnh:

- Trong trường hợp đồ thị ít cạnh, cách biểu diễn này sẽ giúp tiết kiệm không gian lưu trữ.
- Ở một số trường hợp đặc biệt, ta phải xét tất cả các cạnh trên đồ thị thì phương pháp cài đặt này giúp việc duyệt cạnh dễ dàng hơn trong $O(M)$ (ví dụ giải thuật tìm cây khung nhỏ nhất Kruskal).

Nhược điểm của danh sách cạnh: Trong trường hợp cần duyệt các đỉnh kề với một đỉnh *u*, bắt buộc phải duyệt qua mọi cạnh, lọc ra các cạnh có chứa đỉnh *u* và xét đỉnh còn lại. Điều này sẽ tốn thời gian nếu đồ thị có nhiều cạnh.

Phù hợp khi nào: Trong các bài toán cần duyệt toàn bộ cạnh, tiêu biểu như trong giải thuật Kruskal.

Danh sách kề (Adjacency List)

Để khắc phục nhược điểm của ma trận kề và danh sách cạnh, người ta sử dụng danh sách kề (cũng là cách thường xuyên sử dụng nhất trong các bài toán đồ thị). Trong cách biểu diễn này, với mỗi đỉnh u của đồ thị, ta sẽ tạo ra một dánh sách adj_u là các đỉnh kề với nó. Việc cài đặt adj_u có thể thực hiện dễ dàng với vector.

Cài đặt:

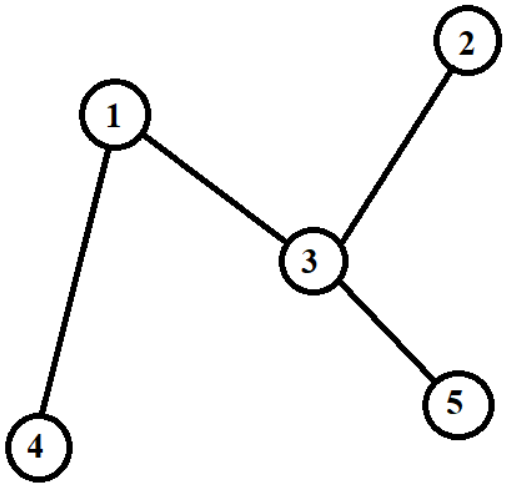
```
vector < int > adj[maxn + 1]; // Danh sách kề, maxn là số đỉnh tối đa của đồ

void enter_adjacency_list()
{
    cin >> N >> M; // Số đỉnh và số cạnh của đồ thị.

    for (int i = 1; i <= M; ++i)
    {
        int u, v;
        cin >> u >> v;

        adj[u].push_back(v); // Đưa v vào danh sách kề với u.
        adj[v].push_back(u); // Nếu là đồ thị có hướng thì không có dòng này.
    }
}
```

Ví dụ: Đồ thị $G(V, E)$ dưới đây gồm 5 đỉnh, 4 cạnh:



Danh sách kề của nó có thể biểu diễn bằng một mảng adj[6] gồm các vector (kích thước mảng là 6 do không có đỉnh số 0), mỗi vector adj[u] lưu danh sách kề của đỉnh u:

adj[1] = {4, 3}
adj[2] = {3}
adj[3] = {1, 5}
adj[4] = {1}
adj[5] = {3}

Ưu điểm của danh sách kề:

- Duyệt đỉnh kề và các cạnh của đồ thị rất nhanh.
- Tiết kiệm không gian lưu trữ, do vector là kiểu dữ liệu với bộ nhớ động, sẽ chỉ tạo ra các ô nhớ tương ứng với số lượng đỉnh kề.

Nhược điểm của danh sách kề: Khi cần kiểm tra (u,v) có phải là một cạnh của đồ thị hay không thì bắt buộc phải duyệt toàn bộ danh sách kề của u hoặc của v.

Phù hợp khi nào: Hầu hết trong mọi bài toán đồ thị đều nên sử dụng, chỉ trừ các bài toán cần duyệt toàn bộ cạnh của đồ thị.

II. Tìm kiếm theo chiều sâu (DFS)

Để giải các bài toán trên đồ thị, chúng ta cần một cơ chế duyệt đồ thị. Giống như thuật toán duyệt cây (Inorder, Preorder, Postorder và Level-Order traverse), các thuật toán tìm kiếm đồ thị bắt đầu từ một số đỉnh nguồn trong đồ thị và tìm kiếm đồ thị bằng cách đi qua các cạnh và đánh dấu các đỉnh.

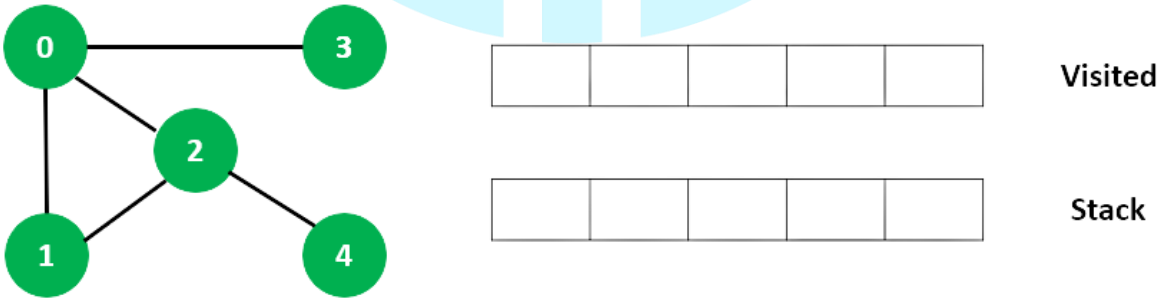
Tìm kiếm theo chiều sâu (*Depth First Search*) là một thuật toán để duyệt qua hoặc tìm kiếm cấu trúc dữ liệu dạng cây hoặc đồ thị. Thuật toán bắt đầu tại nút gốc (chọn một số nút tùy ý làm nút gốc trong trường hợp đồ thị) và kiểm tra từng nhánh càng xa càng tốt trước khi quay lui.

Kết quả của một DFS là một cây bao trùm (spanning tree). Cây khung (spanning tree) là một đồ thị không có vòng lặp. Để thực hiện duyệt theo DFS, chúng ta cần sử dụng cấu trúc dữ liệu ngăn xếp có kích thước tối đa bằng tổng số đỉnh trong biểu đồ.

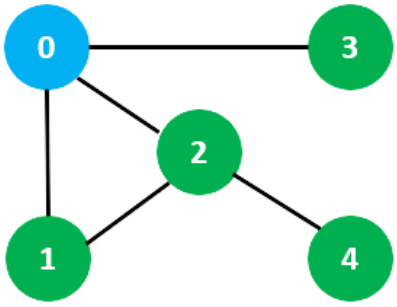
Để cài đặt DFS, ta cần thực hiện các bước sau:

- Lấy một đỉnh bất kỳ trong đồ thì đưa vào ngăn xếp.
- Lấy top value của ngăn xếp để duyệt và thêm vào visited list.
- Tạo một list bao gồm các đỉnh liền kề của đỉnh đang xét, thêm những đỉnh không có trong visited list vào ngăn xếp.
- Tiếp tục lặp lại bước 2 và bước 3 đến khi ngăn xếp rỗng.

Ví dụ: Hãy xem thuật toán Tìm kiếm theo chiều sâu hoạt động như thế nào với một ví dụ. Chúng ta dùng đồ thị vô hướng có 5 đỉnh.



Chúng ta bắt đầu từ đỉnh 0, thuật toán DFS bắt đầu bằng cách đứa nó vào danh sách Visited và đưa tất cả các cạnh liền kề đỉnh đang xét vào ngăn xếp.



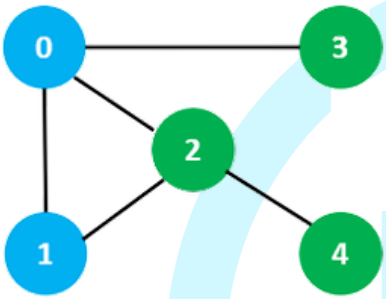
0				

Visited

1	2	3		

Stack

Tiếp theo, chúng ta truy cập phần tử ở đầu ngăn xếp tức là 1 và đi đến các nút liền kề của nó. Vì 0 đã được truy cập, nên 2 là số được xét.



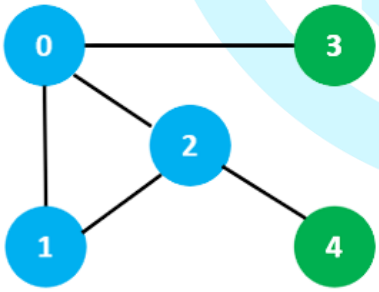
0	1			

Visited

2	3			

Stack

Đỉnh 2 có một đỉnh liền kề chưa được thăm là 4, vì vậy chúng ta thêm đỉnh đó vào vị trí đầu của ngăn xếp và duyệt nó.

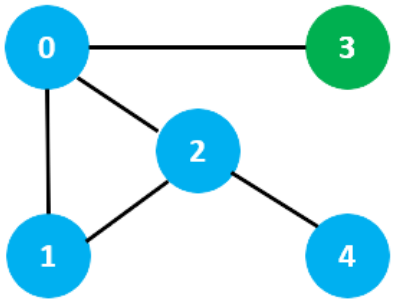


0	1	2		

Visited

4	3			

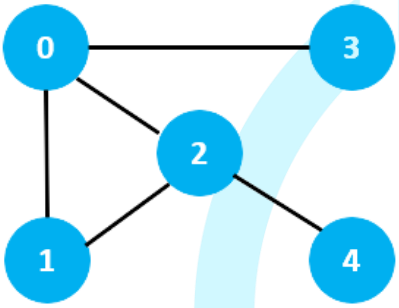
Stack



0	1	2	4		Visited

3					Stack

Sau khi chúng ta duyệt phần tử 3 cuối cùng, nó không có bất kỳ nút liền kề nào chưa được duyệt, vì vậy chúng tôi đã hoàn thành tìm kiếm theo chiều sâu trong đồ thị trên.



0	1	2	4	3	Visited

					Stack

Độ phức tạp thời gian của thuật toán DFS được biểu diễn dưới dạng **O(V + E)**, trong đó V là số nút và E là số cạnh. Độ phức tạp không gian của thuật toán là O(V).

Bài 1: Cho một cây gồm n đỉnh (1 <= n <= 10^5). Hãy in ra các đỉnh được xét lần lượt trong quá trình DFS. Coi đỉnh 1 là gốc.

Ví dụ: Với n = 8 và các cạnh (1,2), (2,3), (3,4), (2,5), (1,6), (6,7), (3,8) kết quả duyệt sẽ là 1 2 3 4 8 5 6 7.

- [Đầu vào] Dòng đầu tiên là n <= 10^5, n -1 dòng tiếp theo gồm u, v (1 <= u, v <= n, u != v) thể hiện 1 cạnh giữa 2 đỉnh u và v.
- [Đầu ra] array integer: mảng là thứ tự đỉnh được xét bằng DFS.

Hướng dẫn:

```cpp
void dfs(vector<vector<int>>& graph, int start) {
    int n = graph.size();
    vector<bool> visited(n, false);
    stack<int> s;
    s.push(start);

    while (!s.empty()) {
        int node = s.top();
        s.pop();

        if (!visited[node]) {
            cout << node << " ";
            visited[node] = true;
        }

        for (int i = graph[node].size() - 1; i >= 0; --i) {
            int neighbor = graph[node][i];
            if (!visited[neighbor]) {
                s.push(neighbor);
            }
        }
    }
}

int main() {
    int n;
    cin>> n;
    vector<vector<int>> graph(n + 1);
    for (int k = 0; k < n - 1; k++)
    {
        int u, v;
        cin >> u >> v;
        graph[u].push_back(v);
        graph[v].push_back(u);
    }
    dfs(graph, 1);
    cout << endl;
}
```

Tiến trình DFS được trình bày như sau:

- Khởi tạo một vector visited để đánh dấu các đỉnh đã được duyệt qua. Ban đầu, tất cả các đỉnh đều chưa được duyệt qua.
- Khởi tạo một stack s để lưu trữ các đỉnh cần duyệt.
- Thêm đỉnh bắt đầu start vào stack s.
- Trong vòng lặp, lấy một đỉnh từ đỉnh stack, gán cho biến node và loại bỏ đỉnh đó khỏi stack.

- Kiểm tra xem đỉnh node đã được thăm chưa. Nếu chưa, in đỉnh đó ra màn hình và đánh dấu là đã thăm.
- Duyệt qua tất cả các đỉnh kề của đỉnh node và đưa những đỉnh kề chưa được thăm vào stack s.

III. Tìm kiếm theo chiều rộng (BFS)

Tìm kiếm theo chiều rộng (*Breadth First Search*) là một thuật toán để duyệt đồ thị hoặc cây. BFS áp dụng cho cây và đồ thị gần như giống nhau. Sự khác biệt duy nhất là đồ thị có thể chứa các chu trình, vì vậy chúng ta có thể duyệt lại cùng một nút. Để tránh xử lý lại cùng một nút, chúng ta sử dụng mảng boolean đã truy cập, mảng này sẽ đánh dấu các đỉnh đã truy cập. BFS sử dụng cấu trúc dữ liệu hàng đợi (queue) để tìm đường đi ngắn nhất trong biểu đồ.

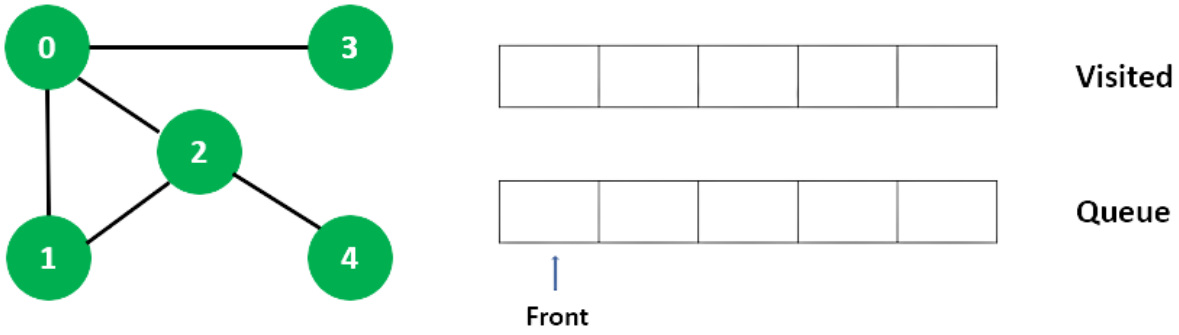
Triển khai BFS tiêu chuẩn sẽ đặt mỗi đỉnh của đồ thị vào một trong hai loại: visited, not visited. Mục đích của thuật toán là đánh dấu mỗi đỉnh là đã thăm để tránh các chu trình.

Cách thuật toán hoạt động như sau:

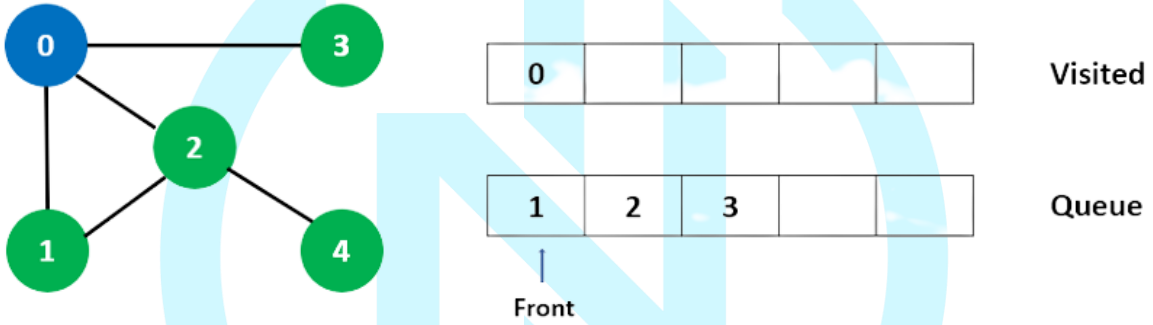
- Lấy một đỉnh bất kỳ trong đồ thị thêm vào cuối hàng đợi.
- Lấy phân tử đầu tiên của hàng đợi và thêm nó vào danh sách đã duyệt.
- Tạo một danh sách các đỉnh liền kề của đỉnh đang xét. Thêm những đỉnh không có trong danh sách đã duyệt vào cuối hàng đợi.
- Tiếp tục lặp lại bước 2 và 3 cho đến khi hàng đợi trống.

Lưu ý: Đồ thị có thể chứa hai thành phần không liên kết khác nhau, vì vậy để đảm bảo rằng mọi đỉnh đều đã được thăm, chúng ta cũng có thể chạy thuật toán BFS trên mọi đỉnh.

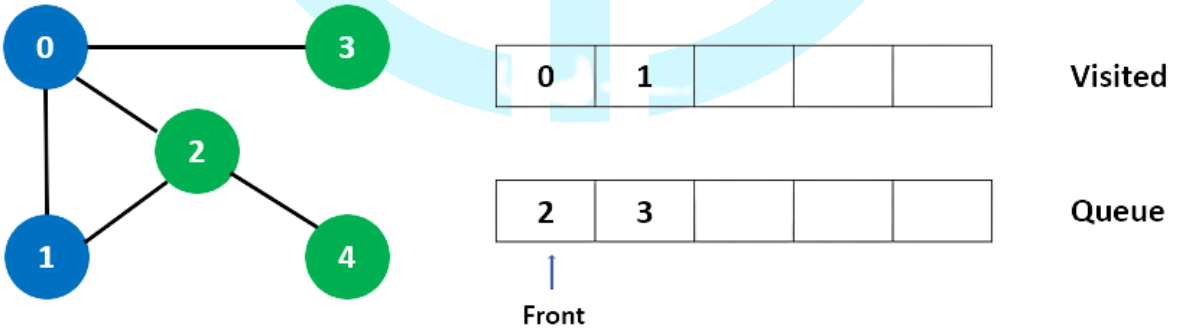
Ví dụ: Hãy xem cách hoạt động của thuật toán Tìm kiếm theo chiều rộng với một ví dụ. Ta dùng đồ thị vô hướng có 5 đỉnh.



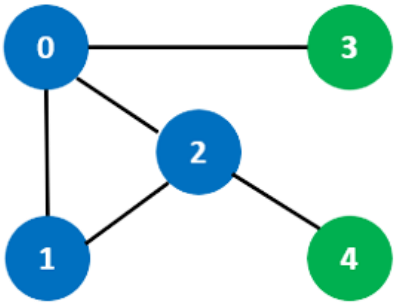
Chúng ta bắt đầu từ đỉnh 0, thuật toán BFS bắt đầu bằng cách đặt nó vào danh sách đã duyệt và đặt tất cả các đỉnh liền kề của nó vào hàng đợi.

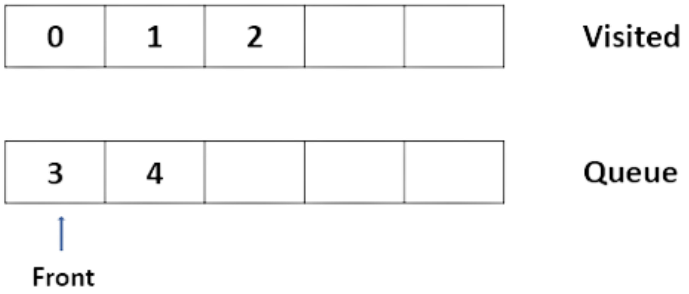


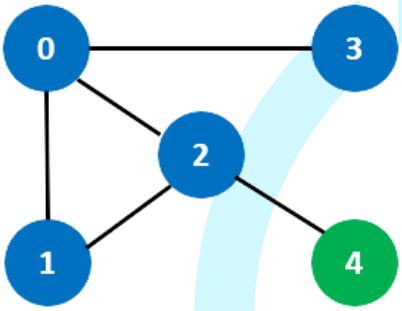
Tiếp theo, chúng tôi truy cập phần tử đầu của hàng đợi, tức là 1 và đi đến các nút liền kề của nó. Vì 0 đã được truy cập, thay vào đó, chúng tôi truy cập 2.

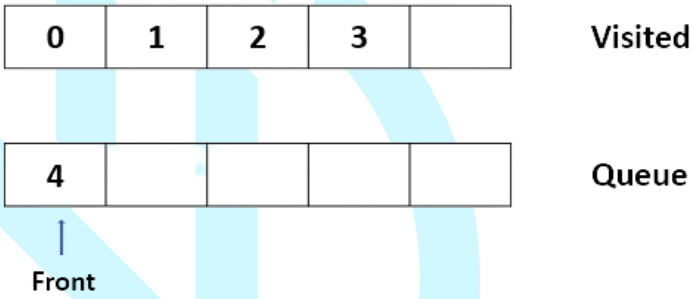


Đỉnh 2 có một đỉnh liền kề chưa được thăm là 4, vì vậy chúng tôi thêm đỉnh đó vào cuối hàng đợi và thăm 3, ở đầu hàng đợi.

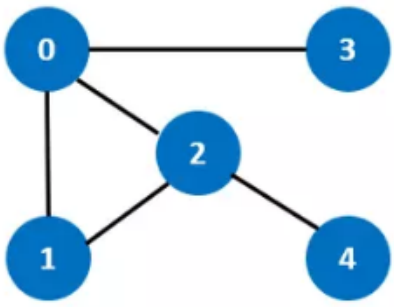


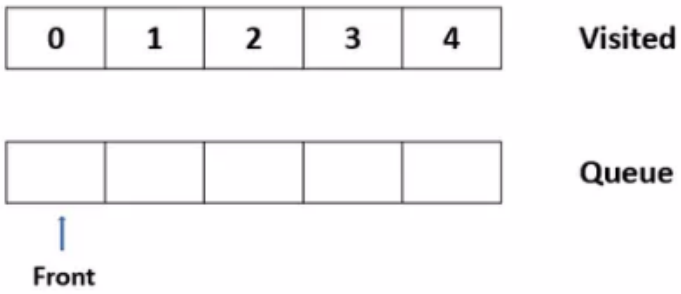






Chỉ còn lại 4 trong hàng đợi vì nút liền kề duy nhất của 3 tức là 0 đã được truy cập. Chúng ta đến thăm nó.





Độ phức tạp thời gian của thuật toán BFS được biểu diễn dưới dạng **O(V + E)**, trong đó V là số nút và E là số cạnh.

Độ phức tạp không gian của thuật toán là O(V).

Bài 1: Cho một cây gồm n đỉnh (1 <= n <= 105). Hãy in ra các đỉnh được xét lần lượt trong quá trình BFS. Coi đỉnh 1 là gốc.

Ví dụ: Với n = 8 và các cạnh (1,2), (2,3), (3,4), (2,5), (1,6), (6,7), (3,8) kết quả duyệt sẽ là 1 2 6 3 5 7 4 8.

- [Đầu vào] Dòng đầu tiên là n <= 10^5, n -1 dòng tiếp theo gồm u, v (1 <= u, v <= n, u != v) thể hiện 1 cạnh giữa 2 đỉnh u và v.
- [Đầu ra] array integer: mảng là thứ tự đỉnh được xét bằng BFS.

Hướng dẫn:

```cpp
void bfs(vector<vector<int>>& graph, int start) {
    int n = graph.size();
    vector<bool> visited(n, false);
    queue<int> q;
    q.push(start);
    visited[start] = true;

    while (!q.empty()) {
        int node = q.front();
        q.pop();
        cout << node << " ";

        for (int neighbor : graph[node]) {
            if (!visited[neighbor]) {
                q.push(neighbor);
                visited[neighbor] = true;
            }
        }
    }
}

int main() {
    int n;
    cin>> n;
    vector<vector<int>> graph(n + 1);
    for (int k = 0; k < n - 1; k++)
    {
        int u, v;
        cin >> u >> v;
        graph[u].push_back(v);
        graph[v].push_back(u);
    }
    bfs(graph, 1);
    cout << endl;

    return 0;
}
```

Tiến trình BFS được thực hiện như sau:

- Khởi tạo một vector visited để theo dõi các đỉnh đã được duyệt qua. Ban đầu, tất cả các đỉnh đều chưa được duyệt qua.
- Khởi tạo một queue q để lưu trữ các đỉnh cần duyệt.
- Thêm đỉnh bắt đầu start vào queue q và đánh dấu là đã thăm.
- Trong vòng lặp, lấy một đỉnh từ đầu queue q, gán cho biến node và loại bỏ đỉnh đó khỏi queue.
- In đỉnh node ra màn hình.
- Duyệt qua tất cả các đỉnh kề của đỉnh node và thêm những đỉnh kề chưa được thăm vào queue q, đồng thời đánh dấu chúng là đã thăm.

IV. Các cấu trúc dữ liệu cây khác

Để có thể viết một các đầy đủ tất cả lý thuyết và cách cài đặt của tất cả các cấu trúc cây – đồ thị khác vào tài liệu này quả là một điều khó khăn với người biên soạn. Vì vậy phần này chúng ta sẽ điểm qua một số cấu trúc đồ thị quan trọng (chắc chắn sẽ gặp trong các bài thi ICPC/ACM) khác kèm đường link tham khảo để các bạn có thể hiểu rõ hơn về CTDL ấy.

1. Cây nhị phân

Cây nhị phân là một trường hợp đặc biệt của cấu trúc cây và nó cũng phổ biến nhất. Đúng như tên gọi của nó, cây nhị phân có bậc là 2 và mỗi nút trong cây nhị phân đều có bậc không quá 2.

Có một số khái niệm khác về cây nhị phân các bạn cần nắm như sau:

- Cây nhị phân đúng: là cây nhị phân mà mỗi nút của nó đều có bậc 2. Ví dụ như hình trên, hoặc hình trên bỏ đi nút H và I cũng là cây nhị phân đúng.
- Cây nhị phân đầy đủ là cây nhị phân có mức của các nút lá đều bằng nhau. Ví dụ hình trên, tất cả các nút lá đều có mức 3.
- Cây nhị phân tìm kiếm (sẽ tìm hiểu bên dưới)
- Cây nhị phân cân bằng: số phần tử của cây con bên trái chênh lệch không quá 1 so với cây con bên phải.

Có 3 cách duyệt cây nhị phân:

- Duyệt tiền tự (NLR): duyệt nút gốc, duyệt tiền tự cây con trái, duyệt tiền tự cây con phải.

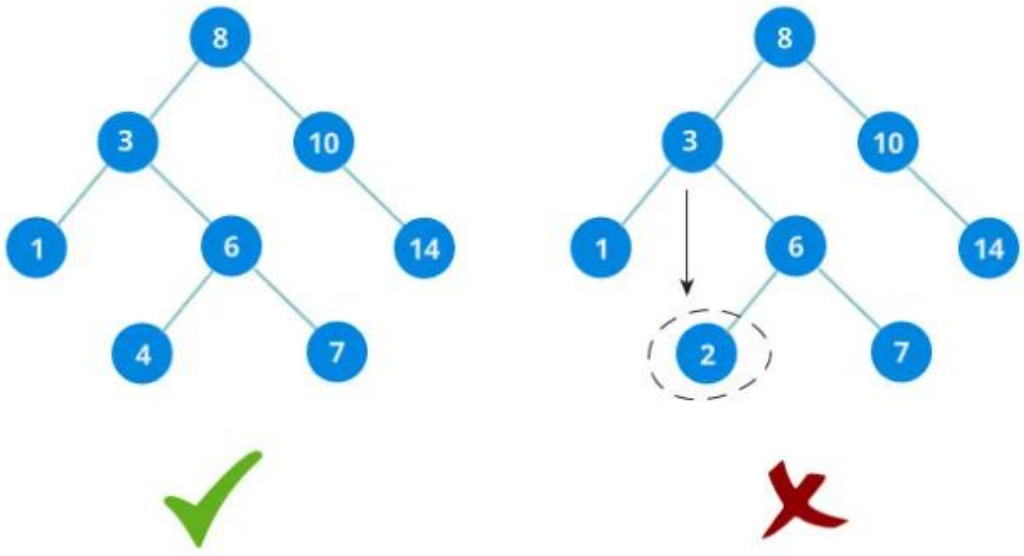
- Duyệt trung tự (LNR): duyệt trung tự cây con trái, duyệt nút gốc, duyệt trung tự cây con phải.
- Duyệt hậu tự (LRN): duyệt hậu tự cây con trái, duyệt hậu tự cây con phải, duyệt nút gốc.

Tham khảo*:* *https://gochocit.com/ky-thuat-lap-trinh/cac-thao-tac-co-ban-tren-cay-nhi-phan-binary-tree*

2. Cây nhị phân tìm kiếm

Cây nhị phân tìm kiếm là cây nhị phân mà trong đó, các phần tử của cây con bên trái đều nhỏ hơn phần tử hiện hành và các phần tử của cây con bên phải đều lớn hơn phần tử hiện hành. Do tính chất này, cây nhị phân tìm kiếm không được có phần tử cùng giá trị.

Nhờ vào tính chất đặc biệt này, cây nhị phân tìm kiếm được sử dụng để tìm kiếm phần tử nhanh hơn (tương tự với tìm kiếm nhị phân). Khi duyệt cây nhị phân theo cách duyệt trung tự, bạn sẽ thu được một mảng có thứ tự.



Tham khảo: *https://blog.luyencode.net/cay-tim-kiem-nhi-phan-binary-search-tree/*

2. Cây phân đoạn (Segment tree)

Cây phân đoạn là một cấu trúc dữ liệu hỗ trợ 2 toán tử: xử lý truy vấn đoạn và cập nhật giá trị mảng. Cây phân đoạn có thể hỗ trợ truy vấn tổng, tối tiểu, tối đại và nhiều các truy vấn khác mà tất cả các toán tử đều làm việc trong thời gian O(logn).

So sánh với cây chỉ mục nhị phân, lợi thế của cây phân đoạn ở chỗ nó là cấu trúc dữ liệu tổng quát hơn. Trong khi cây chỉ mục nhị phân chỉ hỗ trợ truy vấn tổng, thì cây phân đoạn còn hỗ trợ các truy vấn khác. Mặt khác, cây phân đoạn yêu cầu nhiều bộ nhớ hơn và nó cũng khó thực thi mã hơn 1 chút.

Tham khảo:

https://vnoi.info/wiki/algo/data-structures/segment-tree-basic.md

https://vnoi.info/wiki/algo/data-structures/segment-tree-extend.md

3. Cây chỉ số nhị phân (Fenwick tree)

Fenwick Tree, hay còn gọi là **cây chỉ số nhị phân** (*Binary Indexed Tree - BIT*), là một cấu trúc dữ liệu tối ưu cho việc cập nhật giá trị một phần tử và tìm tổng, min/max giữa 2 vị trí bất kì trong mảng. Độ phức tạp cho mỗi lần cập nhật, truy xuất là O(logN) với N là độ dài dãy cần quản lý. Ngoài thao tác tính tổng, tìm min/max thì BIT còn có thể sử dụng được cho nhiều thao thác khác nữa.

Tham khảo:

https://vnoi.info/wiki/algo/data-structures/fenwick.md

https://vietcodes.github.io/algo/fenwick

V. Bài tập với đồ thị

Bài 1: [*https://cses.fi/problemset/task/1192*] Bạn được đưa cho một bản đồ của một tòa nhà và nhiệm vụ của bạn là đếm số phòng của tòa nhà đó. Kích thước của bản đồ là n×m ô vuông và mỗi ô vuông là sàn hoặc tường. Bạn có thể đi sang trái, phải, lên và xuống qua các ô vuông trên sàn.

Ví dụ:

n = 5, m = 8, bản đồ tòa nhà như sau:

```
########
#..#...#
####.#.#
#..#...#
########
```

Kết quả sẽ là 3 phòng.

- [Đầu vào] 1 <= n, m <= 1000, n dòng tiếp theo mỗi dòng có m ký tự “.” (sàn) và “#” (tường)
- [Đầu ra] integer: in ra số phòng của tòa nhà

Hướng dẫn:

Áp dụng DFS để giải bài tập này.

```cpp
#include <bits/stdc++.h>
using namespace std;

int neighborX[4] = {0, 0, 1, -1};
int neighborY[4] = {1, -1, 0, 0};

int n, m, answer = 0;
int vis[1010][1010];
char grid[1010][1010];

bool isValid (int y, int x) {
  if (y < 0) return false;
  if (x < 0) return false;
  if (y >= n) return false;
  if (x >= m) return false;
  if (grid[y][x] == '#') return false;
  return true;
}

void DFS (int y, int x) {
  vis[y][x] = 1;
  for (int i = 0 ; i < 4 ; i++) {
    int newX = x + neighborX[i];
    int newY = y + neighborY[i];
    if (isValid(newY, newX)) {
      if (!vis[newY][newX]) {
        DFS(newY, newX);
      }
    }
  }
}
```

```cpp
int main() {
    cin >> n >> m;
    for (int i = 0 ; i < n ; i++) {
      for (int j = 0 ; j < m ; j++) {
        cin >> grid[i][j];
        vis[i][j] = 0;
      }
    }
    for (int i = 0 ; i < n ; i++) {
      for (int j = 0 ; j < m ; j++) {
        if (grid[i][j] == '.' && !vis[i][j]) {
          DFS(i, j);
          answer++;
        }
      }
    }
    cout << answer << endl;
    return 0;
}
```

Bài 2: [*https://vn.spoj.com/problems/MTWALK/*] Cho một bản đồ kích thước NxN (2 <= N <= 100), mỗi ô mang giá trị là độ cao của ô đó (0 <= độ cao <= 110). Bác John và bò Bessie đang ở ô trên trái (dòng 1, cột 1) và muốn đi đến cabin (dòng N, cột N). Họ có thể đi sang phải, trái, lên trên và xuống dưới nhưng không thể đi theo đường chéo. Hãy giúp bác John và bò Bessie tìm đường đi sao cho chênh lệch giữa điểm cao nhất và thấp nhất trên đường đi là nhỏ nhất.

Ví dụ:

N = 5, bản đồ như sau:

1 1 3 6 8

1 2 2 5 5

4 4 0 3 3

8 0 2 3 4

4 3 0 2 1

Ta được trả về = 2 là chênh lệch cao độ nhỏ nhất (2 và 0)

- [Đầu vào] N là kích thước bản đồ với 2 <= N <= 100, mảng a là mảng 2 chiều với 0 <= $a_{i,j}$ <= 110

- [Đầu ra] integer: 1 số là độ chênh lệch độ cao giữa ô cao nhất và ô thấp nhất của đáp án.

Gợi ý: Duyệt bản đồ theo BFS hoặc DFS để tìm ra đường đi tối ưu.

Bài 3: [*https://codeforces.com/problemset/problem/1829/E*] Bạn được cho một lưới n×m a gồm các số nguyên không âm. Giá trị $a_{i,j}$ biểu thị độ sâu của nước ở hàng thứ i và cột thứ j. Hồ là một tập hợp các ô sao cho: mỗi ô trong tập hợp có $a_{i,j} > 0$ và luôn tồn tại một đường đi giữa một cặp ô bất kỳ trong hồ bằng cách đi lên, xuống, sang trái hoặc sang phải mà không cần bước lên ô có $a_{i,j}=0$ (ô mặt đất). Thể tích của một hồ là tổng độ sâu của tất cả các ô trong hồ đó. Tìm thể tích lớn nhất của hồ trong lưới.

Ví dụ:

Với n x m = 3 x 3. Bản đồ hồ có dạng sau:

1 2 0

3 4 0

0 0 5

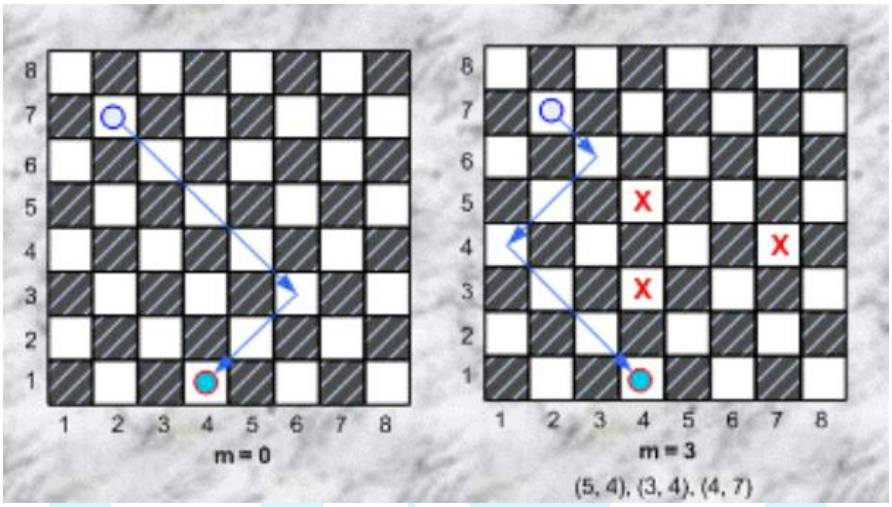
Thì kết quả là 10 (1 + 2 + 3 + 4)

- [Đầu vào] Dòng 1 gồm 1 số nguyên t ($1 <= t <= 10^4$) là số bộ test, mỗi bộ test gồm n, m ($1 \leq n,m \leq 1000$) là kích thước lưới, các giá trị $a_{i,j}$ trong lưới là độ sau tại vùng nước đó thỏa mãn $0 <= a_{i,j} <= 1000$.
- [Đầu ra] Gồm t dòng, mỗi dòng là kết quả của test case tương ứng.

Gợi ý: Áp dụng BFS hoặc DFS để tìm ra thể tích hồ lớn nhất. Với mỗi ô khác 0, ta duyệt tất cả các vùng nước gần kề với nó và so sánh các thể tích hồ tìm được với nhau. Bài toán này cũng hoàn toàn có thể dùng **thuật toán loang** (một giải thuật áp dụng trên ma trận, các bạn có thể tìm hiểu thêm).

Bài 4:[*https://oj.vnoi.info/problem/qbbishop*] Xét bàn cờ vuông kích thước n×n. Các dòng được đánh số từ 1 đến n, từ dưới lên trên. Các cột được đánh số từ 1 đến n từ trái qua phải.

Ô nằm trên giao của dòng i và cột j được gọi là ô (i,j). Trên bàn cờ có m ($0 \le m \le n$) quân cờ. Với m > 0, quân cờ thứ i ở ô (r_i, c_i), i = 1,2,..., m. Không có hai quân cờ nào ở trên cùng một ô. Trong số các ô còn lại của bàn cờ, tại ô (p, q) có một quân tượng. Mỗi một nước đi, từ vị trí đang đứng quân tượng chỉ có thể di chuyển đến được những ô trên cùng đường chéo với nó mà trên đường đi không phải qua các ô đã có quân.



Cần phải đưa quân tượng từ ô xuất phát (p, q) về ô đích (s,t). Giả thiết là ở ô đích không có quân cờ. Nếu ngoài quân tượng không có quân nào khác trên bàn cờ thì chỉ có 2 trường hợp: hoặc là không thể tới được ô đích, hoặc là tới được sau không quá 2 nước đi (hình trái). Khi trên bàn cờ còn có các quân cờ khác, vấn đề sẽ không còn đơn giản như vậy.

Yêu cầu: Cho kích thước bàn cờ n, số quân cờ hiện có trên bàn cờ m và vị trí của chúng, ô xuất phát và ô đích của quân tượng. Hãy xác định số nước đi ít nhất cần thực hiện để đưa quân tượng về ô đích hoặc đưa ra số -1 nếu điều này không thể thực hiện được.

Ví dụ:

Với n = 8, m = 3, p = 7, q = 2, s = 1, t = 4 và vị trí (r_i, c_j) của các quân cờ (tượng không di chuyển được vào các ô này):

5 4

3 4

4 7

Kết quả trả về = 3 là số nước đi ít nhất để quân tượng đến được ô đích.

- [Đầu vào] Dòng đầu tiên chứa 6 số nguyên n, m, p, q, s, t. Nếu m > 0 thì mỗi dòng thứ i trong m dòng tiếp theo chứa một cặp số nguyên ri , ci xác định vị trí quân thứ i. Hai số liên tiếp trên cùng một dòng được ghi cách nhau ít nhất một dấu cách.
- [Đầu ra] integer: số nước đi ít nhất để quân tượng đến được ô đích.

Gợi ý: Dùng BFS để duyệt 4 hướng đi của quân tượng cho đến khi đến được ô đích. Một điều nữa là đối với các bài toán trên ô bàn cờ nói riêng hay các bài toán ma trận nói chung nên sử dụng **struct** hoặc **pair** của STL C++ để xử lý dữ liệu trực quan hơn.

Bài 5:[*https://oj.vnoi.info/problem/olp_kc21_distance*] Nam định nghĩa khoảng cách giữa hai dãy số A = (a_1, a_2,… a_m) và B = (b_1, b_2,… b_n) là giá trị $|a_i – b_j|$ nhỏ nhất trong tất cả các cặp (a_i, b_j).

Ví dụ, khoảng cách giữa hai dãy (1, 5, 7) và (4, -1, 3, 9) là |5 – 4| = 1

Trên dãy số A = (a_1, a_2,… a_m), với cặp chỉ số (L, R), tạo ra dãy số C gồm các phần tử từ L đến R (1 <= L <= R <= m) cụ thể C = (a_L, a_{L+1},… a_R), Nam cần tính khoảng cách của hai dãy số C và B .

Yêu cầu: Cho hai dãy số nguyên A = (a_1, a_2,… a_m), B = (b_1, b_2,… b_n) và cặp chỉ số (L, R), với mỗi cặp chỉ số (L, R), hãy tạo dãy số C tương ứng và đưa ra khoảng cách của dãy số C với dãy số B.

Ví dụ:

Với A = (1, 5, 7), B = (4, -1, 3, 9):

- L = 1, R = 3 → C = (1, 5, 7). Kết quả = |5 – 4| = 1.
- L = 1, R = 2 → C = (1, 5). Kết quả vẫn là 1.
- L = 1, R = 1 → C = (1). Kết quả là |3 – 1| = 2 hoặc |1 – (-1)| = 2

- [Đầu vào] Dòng đầu tiên là 3 số m, n (kích thước của A và B) và k (số truy vấn với mỗi cặp L - R). 2 dòng tiếp theo là 2 mảng A và B ($|a_i|, |b_i| <= 10^9$). K dòng tiếp theo là các truy vấn L – R (1 <= L <= R <= m)
- [Đầu ra] Trên mỗi dòng, in kết quả của từng truy vấn.

Hướng dẫn:

Bài này được lấy từ đề OLP Sinh viên Toàn quốc Khối không chuyên năm 2021. Ta cần sử dụng cấu trúc **Cây phân đoạn** (*Segment tree*) để giải quyết bài toán này. Đây cũng là cấu trúc khá được ưa chuộng khi ra đề thi OLP tin học trong những năm trở lại đây.

```cpp
#include <bits/stdc++.h>      You, 10 months ago • upload
using namespace std;
typedef long long ll;
ll m, n, k;
vector<ll> arrA;
vector<ll> arrB;
vector<ll> tree(100000000);

void buildST(ll id, ll l, ll r)
{
    if (l == r)
    {
        long long left = 0, right = n - 1;
        long long pos;
        if (arrA[l] >= arrB[n - 1]) tree[id] = arrA[l] - arrB[n - 1];
        else if (arrA[l] <= arrB[0]) tree[id] = arrB[0] - arrA[l];
        else{
            while (left <= right)
            {
                long long mid = (left + right) / 2;
                if (arrB[mid] <= arrA[l] && arrA[l] < arrB[mid + 1]) {pos = mid; break;}
                if (arrA[l] >= arrB[mid]) {left = mid + 1;}
                else if (arrA[l] < arrB[mid]) {right = mid - 1;}
            }

            tree[id] = min(abs(arrA[l] - arrB[pos]), abs(arrA[l] - arrB[pos + 1]));
        }
    }

    else {

        ll mid = (l + r) / 2;
        buildST(id * 2 + 1, l, mid);
        buildST(id * 2 + 2, mid + 1, r);
        tree[id] = min(tree[id * 2 + 1], tree[id * 2 + 2]);

    }

}
```

```
ll getMin(ll l, ll r, ll ql, ll qr, ll id)
{
    if (ql > r || qr < l) return LONG_MAX;
    else if (ql <= l && r <= qr) return tree[id];
    ll mid = (l + r) / 2;

    return min(getMin(l, mid, ql, qr, id * 2 + 1), getMin(mid + 1, r, ql, qr, id * 2 + 2));
}

int main()
{
    ios_base::sync_with_stdio(0); cin.tie(NULL); cout.tie(NULL);
    cin>>m>>n>>k;

    for (int i = 0; i < m; i++)
    {
        ll tmp;
        cin>>tmp;
        arrA.push_back(tmp);
    }
    for (int i = 0; i < n; i++)
    {
        ll tmp;
        cin>>tmp;
        arrB.push_back(tmp);
    }
    sort(arrB.begin(), arrB.end());

    buildST(0, 0, m - 1);

    while (k--)
    {
        ll ql, qr;
        cin>>ql>>qr;
        cout<<getMin(0, m - 1, ql - 1, qr - 1, 0)<<'\n';
    }

}
```

Tài liệu tham khảo

Các chuyên mục về CTDL & GT tại:

- *https://viblo.asia/*
- *https://howkteam.vn/*
- *https://topdev.vn/*
- *https://vallicon.com/*
- *https://nguyenvanhieu.vn/*
- *https://vnoi.info/*
- *https://codelearn.io/*

Các bài tập thực hành được tham khảo tại:

- *https://codelearn.io/*
- *https://www.geeksforgeeks.org/*
- *https://codeforces.com/*
- *https://www.spoj.com/PTIT/*
- *https://oj.vnoi.info/*
- *https://leetcode.com/*
- *https://cses.fi/problemset/*
- *https://www.hackerearth.com/*