Bài 1: Tìm một số trong mảng đã sắp xếp và dịch phải - easy level

https://leetcode.com/problems/search-in-rotated-sorted-array-ii/

Cho một mảng sắp xếp tăng dần đã bị dịch phải n bước, tìm xem liệu một số cho trước có nằm trong mảng này không

Ví dụ: cho mảng [1 2 3 4 5 6]

Dịch 1 bước ta có: [6 1 2 3 4 5]

Dịch 4 bước ta có: [3 4 5 6 1 2]

Dịch 6 bước ta có mảng ban đầu

Điều kiện:

Độ dài mảng: 1 <= n <= 5000

Phần tử của mảng: -10^4 <= A[i] <= 10 ^ 4

Phân tích:

Một cấu hình có dạng một số nguyên x. Việc kiểm tra là so sánh cấu hình x với số cần tìm, nếu x giống với số cần tìm thì dừng.

Tổng số cấu hình có trong bài là 5000 cấu hình.

=> Các sinh là dùng một vòng lặp từ phần tử đầu mảng đến phần tử cuối cùng của mảng, tại mỗi bước trả ra cấu hình bằng cách lấy giá trị ở vị trí hiện tại.

Độ phức tạp tính toán là O(n).

Độ phức tạp không gian là 1 (thêm một biến chỉ số).

Data structure: không. Cấu hình là một số nguyên, là kiểu cơ bản.

```
bool search(vector<int>& nums, int target) {
    int length = nums.size();
    for(int i = 0; i < length; i++) { //Sinh cấu hình
       if(nums[i] == target) { // kiểm tra cấu hình.
          return true;
       }
    }
    return false;
 }
```

Bài 2: Tìm chuỗi con palindrome dài nhất - easy level

https://leetcode.com/problems/longest-palindromic-substring/

Chuỗi palindrome là chuỗi mà viết xuôi hay ngược đều giống nhau, ví dụ: “aba” , “tnt”

Cho một chuỗi kí tự s, tìm chuỗi con dài nhất của s sao cho chuỗi này là palindrome

Độ dài của chuỗi s nằm trong khoảng từ 1 đến 1000

Phân tích:

Một chuỗi con của chuỗi (S1, S2, S3,…,Sn) là (Si,Si+1, …, Sj) trong đó 1 ≤ i ≤ j ≤ n.

=> Cách sinh cấu hình là dùng hai vòng lặp để tạo ra hai số i và j thỏa mãn 1 ≤ i ≤ j ≤ n.

=> Độ phức tạp của việc sinh ra các cấu hình là O(n^2).

Để kiểm tra một chuỗi con ***sub*** bắt đầu từ index ***a***, có độ dài len :

Ban đầu ta sẽ sử dụng ***a*** và ***len*** để tìm index của phần tử cuối thuộc chuỗi con ***sub*** là ***b***

Rõ ràng nếu s[a] != s[b] ta có thể kết luận ngay ***sub*** không là palindrome

Nếu s[a] == s[b], ***sub*** sẽ là palindrome nếu chuỗi con từ a+1 đến b-1 là palindrome, vậy ta sẽ tăng ***a*** lên 1, giảm ***b*** đi 1 và lại tiếp tục việc so sánh như trên cho đến khi có trường hợp s[a] != s[b] hoặc…

Vấn đề còn lại sẽ là khi nào dừng việc kiểm tra để đưa đến kết luận chắc chắn chuỗi con ***sub*** là palindrome. Dễ thấy rằng, trong quá trình tăng a, giảm b, ban đầu a luôn < b (giả sử a == b, chuỗi chắc chắn là palidrome do có độ dài 1), đến 1 thời điểm, ta sẽ có a == b + 1 hoặc a == b, hay a > b hoặc a==b

Có thể thấy rằng nếu a == b, ta có thể kết luận chuỗi ***sub*** là palindrome, còn với trường hợp a > b, ta đã xét hết tất cả các cặp kí tự của chuỗi ***sub*** và kết quả đều là trùng khớp. Bởi vậy, khi a == b hoặc a > b, ta có thể đưa ra kết luận ***sub*** là một palindrome, kết hợp 2 trường hợp này lại , ta có điều kiện chỉ tiếp tục so sánh các cặp kí tự khi mà a < b, ví dụ cho thuật toán như sau:

```
bool isPalindrome(string &s, int start, int end) {
    while(start < end) {
      if(s[start] != s[end]) return false;
      ++start; --end;
    }
    return true;
  }
```

=> Độ phức tạp tính toán của việc kiểm tra là O(n).

=> Tổng độ phức tạp là O(n^3). Với n = 1000 thì O(n^3) = 10^9 => Chấp nhận được.

=> Độ phức tạp không gian => 1 (dùng một vài biến số nguyên để giữ giá trị)

=> Cấu trúc dữ liệu => kiểu số nguyên (nguyên thủy).

Vậy ta có thể dùng vét cạn để giải quyết bài toán như sau:

```
string longestPalindrome(string s) {
    int max = 0;
        for (int i = 0; i < s.length(); i ++) {
                for (int j = i; j < s.length(); j ++)
                        if (isPalindrome(s, i, j) && (j - i + 1 > max))
                                max = j - i + 1;
        }
    return max;
}
```

Tăng tốc độ:

Có hai cách để tăng tốc độ vét cạn. Là giảm không gian tìm kiếm hoặc sắp xếp lại không gian tìm kiếm.

- Giảm không gian tìm kiếm:
    - Có một biến giữ giá trị độ dài chuỗi con dài nhất hiện tại. Tại mỗi cấu hình ta kiểm tra xem độ dài cấu hình hiện tại có độ dài tốt hơn cấu hình tốt nhất đang có thì mới tiến hành kiểm tra.
    - Dùng binary search.
- Sắp xếp lại không gian tìm kiếm: Ta có thể bắt đầu từ chuỗi có độ dài lớn nhất rồi giảm dần cho đến khi tìm được chuỗi palindrome. Trường hợp xấu nhất vẫn là O(n^3).

Bài 3: REPROAD - easy level

https://www.spoj.com/problems/REPROAD/

Tóm tắt đề bài:
Cho một con đường dài N (5 <= N <= 10000) gồm các đoạn 0 (bị hỏng, không di chuyển được) và 1 (di chuyển được); và số K (0<=K<=N).
Bài toán cho phép sửa một đoạn đường bị hỏng (sửa 0 thành 1). Có tối đa K lần sửa.
In ra chiều dài lớn nhất của quãng đường di chuyển được (số các số 1 liên tiếp) với K lần sửa.

Input

Tổng số lượng phép thử là T (1 <= T <= 20) được cho trên dòng đầu tiên.
Mỗi phép thử được cho trên 2 dòng, dòng đầu tiên của mỗi phép thử là chiều dài N (5 <= N <= 10000) của con đường và số lần sửa tối đa K (0 <= K <= N), cách nhau bởi 1 dấu cách trắng.
Dòng tiếp theo mô tả trạng thái của từng đoạn đường (0 hoặc 1), 2 số cạnh nhau được ngăn cách bởi 1 dấu cách trắng.

Output
Hãy in đáp án của mỗi phép thử trên 1 dòng.

Phân tích:

Giống bài số 2, một cấu hình sẽ là vị trí đầu và cuối (i, j) trong đó 1 ≤ i ≤ j ≤ n.

=> Các sinh giống bài 2. Độ phức tạp sinh là O(n^2).

Cách kiểm tra cấu hình là tìm số lượng số 0 ở trong dãy từ i đến j. Nếu số lượng số 0 nhỏ hơn K -> cấu hình hợp lệ.

=> Độ phức tạp kiểm tra là O(n).

*=> Độ phức tạp của vét cạn là O(n^3). Với n = 10^4 -> độ phức tạp là O(10^12) => **không** chấp nhận được.*

=> Độ phức tạp không gian -> 1.

=> Cấu trúc dữ liệu -> kiểu số nguyên.

Tăng tốc độ:

Giảm không gian:

- Dùng binary search -> độ phức tạp sinh là O(nlogn), độ phức tạp kiểm tra O(n) -> tổng O(n^2logn) = 10^9 => tạp chấp nhận được.
  => độ phức tạp là O(n^2logn). Độ phức tạp không gian ->1. Cấu trúc dữ liệu -> kiểu cơ bản.
- Dùng tham lam: Nhận xét thấy rằng việc khác nhau giữa hai cấu hình hợp lệ (i1,j1) và (i2, j2) trong đó i1 <i2, j1 < j2 phải thỏa mãn: Số số 0 nằm trong khoảng i1 đến i2 nhỏ hơn hoặc bằng số số 0 nằm trong j1 đến j2.
  Do đó, việc tính toán cấu hình tiếp theo từ cấu hình hiện có (i1, j1) bằng cách tăng i1 lên thành i2 với số số 0 nằm trong i1 và i2 là 1 số 0. Tương tự với j1 và j2.
  Cấu hình đầu tiên hợp lệ là cấu hình đã sửa K đoạn hỏng đầu tiên.
  => độ phức tạp là O(n^2). Độ phức tạp không gian ->1. Cấu trúc dữ liệu -> kiểu cơ bản.
- Bằng cách stack lại các vị trí của i, ta có thể loại bỏ việc tìm i2 -> giảm độ phức tạp xuống còn O(n).
  => độ phức tạp là O(n). Độ phức tạp không gian ->1. Cấu trúc dữ liệu -> stack.

```
int solve(){
        int ans = -1;
        list<int> q;
        int len = 0;
        int cost = 0;
        for(int i = 0; i<N;i++){
                if(A[i] == 0){
                        if(cost < K){
                                cost++;
                                len++;
```

```
                        } else {
                                len = i - q.front();
                                q.pop_front();
                        }
                        q.push_back(i);
                } else {
                        len++;
                }
                ans = max(ans,len);
        }
        return ans;
}
```

Bài 4: SOLIT - hard level

https://www.spoj.com/problems/SOLIT/

Tóm tắt đề bài:
Cho một bàn cờ 8x8 ô, đánh số từ 1 đến 8 (từ trên xuống dưới, từ trái sang phải).
Trên bàn cờ có 4 quân cờ khác nhau. Mỗi quân cờ có thể di chuyển:
- Di chuyển theo hướng Trên/Dưới/Trái/Phải sang ô còn trống ngay bên cạnh.
- Nếu ô bên cạnh đã có một quân cờ, nhảy qua ô này đến ô còn trống tiếp theo (trên cùng hướng di chuyển) nếu ô đó còn trống.

Bàn cờ A có 4 quân cờ.
Bàn cờ B có 4 quân cờ.
Xác định xem với tối đa 8 lần di chuyển các quân cờ trên bàn cờ A, có thể làm bàn cờ A và bàn cờ B có các quân cờ ở vị trí giống nhau hay không?

Input

Tổng số lượng phép thử là T được cho trên dòng đầu tiên.
Mỗi phép thử được cho trên 2 dòng, dòng đầu tiên của mỗi phép thử là tọa độ các quân cờ (a[2j-1] và a[2j] (1 <= j <= 4) lần lượt là hàng và cột của quân cờ thứ j) của bàn cờ A, cách nhau bởi dấu cách trắng.
Dòng tiếp theo là tọa độ các quân cờ của bàn cờ B, cách nhau bởi dấu cách trắng.

Output
Hãy in đáp án của mỗi phép thử trên 1 dòng dưới dạng:

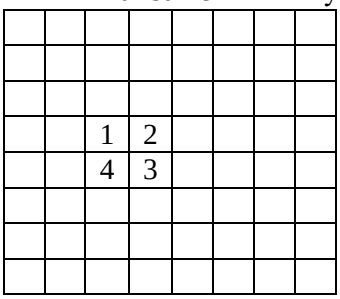
Case #i: K

Nhận xét chung

- Ta có thể sử dụng một mảng 4 phần tử để lưu tọa độ x của 4 quân cờ, và một mảng 4 phần tử khác để lưu tọa độ y của 4 quân cờ; hoặc sử dụng một mảng 4 phần tử với kiểu struct để lưu đồng thời x và y của mỗi quân cờ.
- Trong phần bài giải này, ta chọn phương án thứ 2 với struct như sau

```
typedef struct _Node{
        int r; //Hàng của quân cờ
        int c; //Cột của quân cờ
} Node;
Node A[4],B[4];
```

- Hai bàn cờ dưới đây có cách biểu diễn trong mảng là khác nhau nhưng có trạng thái giống nhau.



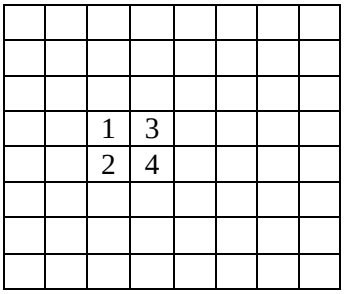
A[0] = {4,3}

A[1]= {4,4}

A[2]={5,4}

A[3]={5,3}

⇨ A[] = {4, 3, 4, 4, 5, 4, 5, 3}



B[0] = {4,3}

B[1]={5,3}

B[2]={4,4}

B[3]={5,4}

⇨ B[] = {4, 3, 5, 3, 4, 4, 5, 4}

⇨ Mặc dù thứ tự quân cờ là khác nhau nhưng 2 bàn cờ trên vẫn giống nhau.

⇨ Để so sánh 2 bàn cờ, ta cần sắp xếp các quân cờ trong mảng theo thứ tự tăng dần. Sau đó, ta mới có thể kiểm tra xem các quân cờ trong mảng có giống nhau không.

- Sắp xếp – bubble sort do chỉ có 4 quân cờ:

```
for(int i=0;i<4;i++){
        for(int j=i+1;j<4;j++){
                if((tmp[j].r < tmp[i].r || (tmp[j].r == tmp[i].r && tmp[j].c < tmp[i].c))){
                        swap(tmp[i],tmp[j]);
                }
        }
}
```

- Sau khi đã sắp xếp, 2 bàn cờ trên trở thành:

A[] = {4 3 4 4 5 3 5 4}

B[] = {4 3 4 4 5 3 5 4}

⇨ Để so sánh hai bàn cờ, ta duyệt qua từng phần tử của A và B và kiểm tra xem chúng có giống nhau không.

- So sánh:

```
for(int i = 0; i < 4; i++){
        if(a[i].r != b[i].r || a[i].c != b[i].c){
                return false;
        }
}
return true;
```

Bruteforce-01

1.1. Nhận xét

- Mỗi quân cờ có 4 cách di chuyển (trên/dưới/trái/phải).
- Nước đi đầu tiên có thể chọn 1 trong 4 quân cờ => Có 4x4 trạng thái cuối.
- Nước đi thứ 2 phải bắt đầu từ 1 trong các trạng thái cuối của nước đi thứ nhất (4x4 trạng thái), mỗi trạng thái có thể chọn 1 trong 4 quân cờ, mỗi quân cờ có 4 cách di chuyển => Có 4x4x4x4 trạng thái cuối.
- Tương tự, sau 8 nước đi, có tổng cộng 16^8 = 2^32 ~ 4*1e9 trạng thái.
- Liệt kê tất cả các trạng thái và kiểm tra xem có trạng thái nào trùng với trạng thái cuối (bàn cờ B) hay không.

1.2. Liệt kê các trạng thái đồng thời kiểm tra tính đúng đắn

```
bool backtrack(Node a[4],Node b[4], int level){
    if(equal(a,b)){
        return true;
    }
    if(level >=8){
        return false;
    }
    for(int p = 0; p < 4; p++){
        for(int dir = 0; dir < 4; dir++){
            for(int step = 1; step<=2; step++){
                if(movable(a[p].r+dr[dir]*step,a[p].c+dc[dir]*step,a)){
                    a[p].r += dr[dir]*step;
                    a[p].c += dc[dir]*step;
                    if(backtrack(a,b,level+1)){
                        return true;
                    }
                    a[p].r -= dr[dir]*step;
                    a[p].c -= dc[dir]*step;
                    break;
                }
            }
        }
```

```
        }
        return false;
    }
```

Bruteforce-02

1.3. Nhận xét

- Thay vì liệt kê 8 nước đi từ A tới B, nếu ta liệt kê các trạng thái sau 4 nước đi từ A (S1) và các trạng thái sau 4 nước đi từ B (S2). Bài toán được thỏa mãn nếu có trạng thái chung giữa S1 và S2.
- Không gian tìm kiếm lúc này chỉ là 2*16^4 = 2^17~1e5.

⇨ Sắp xếp lại không gian tìm kiếm giúp giảm không gian tìm kiếm.

- Trong trường hợp này, ta cần lưu lại các trạng thái của S1 vào map, sau đó khi liệt kê các trạng thái của S2, ta kiểm tra xem trạng thái này có nằm trong S1 hay không.

⇨ Bộ nhớ sử dụng: 16^4*4*sizeof(Node) = 2^16*4*8 = 2^21 ~ 2MB.

- Ta có thể giảm bộ nhớ và khiến việc cập nhật/kiểm tra map dễ dàng hơn bằng việc biến mỗi trạng thái từ mảng 4 phần tử thành số integer như sau:
    - Bàn cờ có 8x8 ô => Tọa độ hàng và cột được biểu diễn từ 0 đến 7 => Cần 3 bit => Mỗi quân cờ có thể được biểu diễn bằng 6 bit.
    - Với 4 quân cờ, ta cần 24 bit để biểu diễn được trạng thái của bàn cờ.
    - Việc so sánh 2 bàn cờ lúc này trở thành so sánh 2 số integer.
- Đổi mảng 4 phần tử thành số integer:

```
ul encode(Node a[4]){
        ul s=0;
        Node tmp[4];
        memcpy(tmp,a,sizeof(Node)*4);          //Copy data để việc sắp xếp không ảnh hưởng tới mảng ban đầu
        for(int i=0;i<4;i++){      //Sắp xếp lại các quân cờ theo thứ tự tăng dần
            for(int j=i+1;j<4;j++){
                if((tmp[j].r < tmp[i].r || (tmp[j].r == tmp[i].r && tmp[j].c < tmp[i].c))){
                    swap(tmp[i],tmp[j]);
                }
            }
        }
        for(int i=3; i>=0; i--)      //Chuyển đổi mảng 4 phần tử thành số integer
      s = (s << 6) | ((tmp[i].r << 3) | tmp[i].c);
        return s;
}
```

- Với mỗi trạng thái integer đã được lưu, ta cần đổi ngược về mảng 4 quân cờ ban đầu:

```
void decode(ul s, Node a[4]){
        for(int i=0; i<4; i++) {
      a[i].r = (s >> 3) & 7;
      a[i].c = s & 7;
      s = s >> 6;
   }
}
```

1.4. Liệt kê các trạng thái đồng thời kiểm tra tính đúng đắn

```
bool backtrack(ul src,ul dst, int level,int cnt){
        if(src == dst){
                return true;
        }
        if(cnt == 1 && m[0][src]){
                return true;
        }
        m[cnt][src] = true;
        if(level >=4){
                return false;
        }
        Node a[4];
        decode(src,a);
        for(int p = 0; p < 4; p++){
                for(int dir = 0; dir < 4; dir++){
                        for(int step = 1; step<=2; step++){
                                if(movable(a[p].r+dr[dir]*step,a[p].c+dc[dir]*step,a)){
                                        a[p].r += dr[dir]*step;
                                        a[p].c += dc[dir]*step;
                                        if(backtrack(encode(a),dst,level+1,cnt)){
                                                return true;
                                        }
                                        a[p].r -= dr[dir]*step;
                                        a[p].c -= dc[dir]*step;
                                        break;
                                }
                        }
                }
        }
```

```
        return false;
    }
```

Bài 5: BURGLARY - medium level

https://www.spoj.com/problems/BURGLARY/

Tóm tắt đề bài:
Cho 1 mảng N (1<=N<=30) các số tự nhiên (mỗi số nằm trong khoảng từ 0 đến 1e9) và một số D (0<=D<=3*1e10).
Chọn K số bất kì trong mảng N sao cho tổng của các số này bằng D.
- Nếu chỉ có 1 cách chọn duy nhất, in ra số k
- Nếu có nhiều cách chọn khác nhau (nhiều số k khác nhau thỏa mãn đề bài), in "AMBIGIOUS"
- Nếu không có cách chọn nào, in "IMPOSSIBLE"

Input

Tổng số lượng phép thử là T (1 <= T <= 20) được cho trên dòng đầu tiên.
Mỗi phép thử được cho trên 2 dòng, dòng đầu tiên của mỗi phép thử là N và D, cách nhau bởi 1 dấu cách trắng.
Dòng tiếp theo mô tả các giá trị trong mảng, 2 số cạnh nhau được ngăn cách bởi 1 dấu cách trắng.

Output
Hãy in đáp án của mỗi phép thử trên 1 dòng dưới dạng:

Case #i: K

Bruteforce-01

1.1. Nhận xét
- Với mỗi số trong mảng, có 2 lựa chọn: chọn/không chọn -> O(2^30) ~ 1e9
- Với mỗi lần thử, ta tính tổng các số đã được chọn và kiểm tra xem nó có bằng D không. Nếu bằng D, lưu lại số các số đã chọn (K)
- Ta thử với tất cả các trường hợp và kiểm tra xem liệu có thể chọn được nhiều số K hay không.

1.2. Liệt kê các hoán vị
Liệt kê các hoán vị theo phương pháp liệt kê các dãy nhị phân.

```
ul s[1<<N];
int ans = -1;       //Không chọn được số K
for (i = 0; i < (1<<N); i++){
        ul sum = 0;
        for(int j = 0; j < N; j++){
                if(i&1<<j){       // j được chọn
                        sum += v[j];      //số j được chọn
```

```
                }
            }
            if(sum == D){
                int cnt = __builtin_popcount(i);  //lấy ra số bit được set của i = số các số cần chọn
                if(ans != -1 && ans != cnt){
                    return -2;          //Chọn được nhiều số K
                }
                ans = cnt;
            }
        }
    return ans;
```

Bruteforce-02

1.3. Nhận xét

- Nếu ta chia mảng đầu vào thành 2 nửa (nửa A và nửa B) sau đó lưu tổng của mỗi hoán vị của mảng A và B tương ứng vào bảng S1 và S2. Lúc này, tổng của các số trong mảng ban đầu bằng D nếu tồn tại một số trong S1 và một số trong S2 sao cho tổng của 2 số này bằng D.
- Không gian tìm kiếm: O(2^15) ~ 32768 ~ 3*1e5

⇨ Giảm không gian tìm kiếm

1.4. Liệt kê các hoán vị

```
map<ul,int> generate(vector<int> v){
        map<ul,int> m;
        int sz = v.size();
        vector<ul> s(1<<sz,0ULL);
        for(int i = 0; i < 1<<sz; i++){
                for(int j = 0; j< sz; j++){
                        if(i&(1<<j)){
                                s[i] = s[i^(1<<j)] + v[j];
                                int cnt = __builtin_popcount(i);
                                if(m[s[i]] != 0 && m[s[i]]!= cnt){
                                        m[s[i]] = -2;
                                } else {
                                        m[s[i]] = cnt;
                                }
                                break;
                        }
                }
        }
        if(m.count(0)){
                m[0] = -2;
        } else{
                m[0] = 0;
        }
        return m;
}
```

1.5. Kiểm tra tính đúng đắn

```
int ans = -1;
for(auto it:m1){
        ul n = D - it.first;
        if(m2.find(n)!=m2.end()){
                if(m2[n] != -2 && it.second != -2){
                        if(ans != -1 && ans != it.second + m2[n]){
                                ans = -2;           // Chọn được nhiều số K
                                break;
                        }
                        ans = it.second + m2[n];
```

```
                } else {
                        ans = -2;           // Chọn được nhiều số K
                        break;
                }
        }
}
```