KỸ THUẬT NHÁNH CẬN

Định nghĩa: Trong một số bài toán, để tìm ra được đáp án, chúng ta phải sử dụng phương pháp liệt kê hết cấu hình để đánh giá và tìm ra cấu hình tốt nhất. Việc liệt kê cấu hình có thể sử dụng bằng việc sinh tuần tự hoặc quay lui. Tuy vậy, trong quá trình liệt kê, chúng ta có thể sẽ liệt kê rất nhiều cấu hình vô nghĩa, gây ra rất nhiều trường hợp khiến chương trình chạy chậm, thậm chí không thể ra được kết quả. Khi đó, một vấn đề đặt ra là trong quá trình liệt kê lời giải ta cần tận dụng những thông tin đã tìm được để loại bỏ sớm những phương án chắc chắn không phải tối ưu. Kỹ thuật đó gọi là kỹ thuật đánh giá nhánh cận trong tiến trình quay lui.

Mô hình nhánh cận:

```
<khởi tạo một cấu hình bất kỳ cho BESTCONFIG>
void Try(i) {
  for <mọi giá trị V có thể gán cho x[i]> {
    <thử x[i] := V>;
    if <việc thử trên vẫn còn hi vọng tìm ra cấu hình tốt hơn
BESTCONFIG> {
        if <x[i] là phần tử cuối cùng trong cấu hình> {
            <cập nhật BESTCONFIG>
        } else {
                <ghi nhận việc thử x[i] = V (nếu cần)>;
                Try(i + 1);
                <bỏ ghi nhận việc thử x[i] := V (nếu cần)>;
            }
    }
  }
}
```

Lưu ý : Kỹ thuật nhánh cận thêm vào cho thuật toán quay lui khả năng đánh giá theo từng bước, nếu tại bước thứ *i*, giá trị thử gán cho *x[i]* không có hi vọng tìm thấy cấu hình tốt hơn cấu hình *BESTCONFIG* thì thử giá trị khác ngay mà không cần phải gọi đệ quy tìm tiếp hay ghi nhận kết quả làm gì (đặt cận để bỏ nhánh, cận càng tốt thì sẽ bỏ được nhiều nhánh hơn). Nghiệm của bài toán sẽ được làm tốt dần, bởi khi tìm ra một cấu hình mới (tốt hơn

BESTCONFIG), ta không in kết quả ngay mà sẽ cập nhật *BESTCONFIG* bằng cấu hình mới vừa tìm được.

Ví dụ. Tên bài: **NHIPHANK.cpp**

Cho tệp **NHIPHANK.inp** gồm 2 số nguyên *n, k (1<=n<=20, 1<=k<=n)*. Hãy ghi ra tệp **NHIPHANK.out** gồm các số nhị phân có đúng *k* số 1. (mỗi số nằm trên mỗi hàng và theo thứ tự từ điển từ nhỏ đến lớn)

Ở ***ví dụ*** này ta chỉ đơn giản đệ quy sinh nhị phân, trong lúc sinh: nếu mà tổng các số 1 lớn hơn *k* thì ta bẻ nhánh đó (vì nếu có đi theo nhánh đó thì không bao giờ cho ra kết quả số nhị phân có đúng *k* số 1 được. Dùng thêm 1 biến *tong* để tính số lượng số 1 của dãy nhị phân. Nếu như ta đang xét đến phần tử *x[i]* mà *tong* > *k* thì ta bỏ nhánh đó đi (vì có đi tiếp thì *tong* này vẫn > *k*)

```
#include <bits/stdc++.h>
using namespace std;
int i, n, k;
int tong = 0;
int x[21];
void   vietcauhinh (){
   // Vì x[n] chưa cập nhật vào tong nên ta phải cộng với tong để so sánh
   if (tong + x[n]==k){
      for (int i = 1; i <= n; i++)   cout << x[i];
      cout << endl;
   }
}

void   nhiphank(int i){
   if ( tong >k) return; // Đặt cận để bẻ nhánh
   for (int j = 0; j <= 1; j++){
      x[i] = j;
      if (i == n ) vietcauhinh();
      else {
         tong = tong + x[i]; // Tính số lượng số 1
         nhiphank (i+1);
         tong = tong - x[i]; // Trả lại giá trị cho tong
      }
   }
}
```

```
int main() {
    ios_base::sync_with_stdio(false);
    cin.tie(0); cout.tie (0);
    freopen ("nhiphank.inp", "r", stdin);
    freopen ("nhiphank.out", "w", stdout);
    cin >> n >> k;
    nhiphank (1);
    return 0;
}
```

Ở bài toán này, ta có thể nghĩ thêm 1 cận nữa để cho chương trình chạy nhanh hơn: Khi xét tiếp đến phần tử $x[i]$, Nếu $tong + (n - i + 1) < k$ thì ta cũng bỏ nhánh đó đi, có đi tiếp nữa thì số lượng con 1 vẫn $< k$ (vì trường hợp tốt nhất là tất cả phía sau vị trí i đều bằng 1 mà tổng số con 1 vẫn $< k$)

```
void   nhiphank(int i){
    if ( tong >k) return; // Đặt cận để bẻ nhánh
    if (tong + n - i +1) // Đặt thêm cận để bẻ nhánh
    for (int j = 0; j <= 1; j++){
        x[i] = j;
        if (i == n ) vietcauhinh();
        else {
            tong = tong + x[i];
            nhiphank (i+1);
            tong = tong - x[i];
        }
    }
}
```