BT TÍNH ĐIỂM CỘNG

Bài 1.

Cho dãy số nguyên có n số $a_1, a_2, \ldots, a_n$ mà giá trị mỗi số thuộc tập hợp $\{0,1,2\}$. Cần đếm xem có tất cả mấy cách chọn ra cặp chỉ số (L,R) với $1 \le L < R \le n$ sao cho trong dãy con của a xét từ vị trí L đến vị trí R thì có một số nào đó xuất hiện từ 3 lần trở lên. Hãy mô tả ít nhất ba cách giải cho bài này với ba độ phức tạp khác nhau.

Ví dụ:

$a = [0,1,2,2,2]$ thì đáp số là 3, ứng với 3 cách chọn cặp chỉ số $(L,R) = (3,5),\ (2,5),\ (1,5)$.

Cách 1. Ta duyệt vét cạn 3 vòng for → O(n^3):

Duyệt theo L, duyệt theo R, kiểm tra khoảng từ L → R thì có thoả mãn không?

Cách 2. Ta gọi b(i), c(i), d(i) là số các số 0, 1, 2 tích luỹ đến vị trí i. Từ đây, ta thấy có thể xây dựng được mảng này bằng 1 vòng lặp:

If a[i] = 0 → b(i) = b(i-1)+1, c(i) = c(i-1) và d(i) = d(i-1).

Tương tự nếu a[i] = 1 hoặc a[i] = 2.

Để kiểm tra khoảng từ L → R có thoả mãn không, ta chỉ tốn O(1). Cụ thể: để đếm số số 0 từ L → R, ta tính b(R) – b(L-1). Tương tự với số số 1 và số số 2.

Ta chỉ tốn 2 vòng for → O(n^2).

Cách 3. Ý tưởng: nếu dãy con có độ dài >= 7 thì theo nguyên lý Dirichlet / chuồng thỏ / chuồng bồ câu thì chắc chắn sẽ có một ký tự xuất hiện >= 3 lần.

Vì thế, ta chỉ cần duyệt theo L → các chỉ số R kể từ L+6 trở đều thoả mãn → res += (n+1-L-6).

Ta kiểm tra thêm các đoạn [L;L+2], ...; [L;L+5] xem có thêm mấy cái thoả.

Từ đó, ta tốn 1 vòng for → O(n).

//Ghi chú: bài này có 1 phiên bản quen thuộc hơn là: đếm số dãy con mà trong đó có đủ 3 loại ký tự. Ta xây dựng 3 mảng đếm b(i), c(i), d(i) như trên. Sau đó, ta cũng duyệt theo L rồi tìm kiếm nhị phân trên miền b(i+1...n) xem chỉ số j đầu tiên nào mà b(j) – b(i-1) >= 1, nghĩa là có chứa số 0. Tương tự tìm chỉ số k để có c(k) – c(i-1) >= 1 và chỉ số g để d(g)–d(i-1)>= 1. Khi đó, vị trí đầu tiên thoả mãn sẽ là: max(j, k, g) → res += (n+1 – max).

Bài 2. Cho trước các số n, m, k với $mk \le n$. Từ một mảng có n phần tử không âm và các số nguyên dương, hãy xây dựng công thức quy hoạch động để xác định tổng lớn nhất trong k mảng con rời nhau, mỗi mảng có đúng m phần tử liên tiếp.

Ví dụ:

- n = 5 và 1,2,3,4,5 và k = m = 2; lấy ra mảng [2;3] và [4;5] → 2+3+4+5 = 14.
- n = 5 và 5,2,6,3,4 và k = m = 2 → [5,2] và [6,3] → 5+2+6+3=16.
- n = 5 và 5,2,6,3,4 và k = 1, m = 3 → [5,2,6] hoặc [6,3,4] → tổng max = 13.
- n = 5 và 5,2,6,3,4 và k = 3, m = 1 → [5], [6], [4] → tổng max = 15.

Ta định nghĩa công thức quy hoạch động:

dp[i][j] = tổng lớn nhất của j mảng con rời nhau tính từ 1 → i. Ta có 2 chiều để mô tả 2 trạng thái, bao gồm: vị trí đang xét và số mảng đã lấy được.

+ Nếu không chọn số a[i] → dp[i][j] = dp[i-1][j].

+ Nếu chọn số a[i] → khi đó, ta đã chọn tất cả các số a[i], a[i-1], …, a[i-m+1]. Do đó:

dp[i][j] = dp[i-m][j-1] + sum(a[i], …, a[i-m+1]).

Giữa hai giá trị dp ở vế phải, ta tính max để có vế trái.

Chú ý tổng sum có thể tính bằng mảng cộng dồn (prefix sum).

Đến đây, ta duyệt 2 vòng for theo i và theo j → kq = dp[n][k].

Nhắc lại: mảng prefix sum có dạng s[i] = a[1]+a[2]+…+a[i] → tính mảng này độc lập trong O(n). Khi đó, để tính query sum trong miền [L;R] → s[R] – s[L-1] với L > 1 (quy ước s[0] = 0) và nếu L=1 thì s[R].

Độ phức tạp chung là O(n^2).

BT tương tự: Cho dãy n số gồm 0, 1, 2 tương tự trên, cho n <= 10^5. Đếm số cặp chỉ số (L, R) sao cho 1 <= L < R <= n và:

số lượng số 0 * số lượng số 1 * số lượng số 2 = luỹ thừa của 2.

Cách 1 & 2 tương tự trên.

//kiểm tra số n là luỹ thừa của 2 → if (n > 0) và (n & (n-1) == 0).

Cách 3 → suy nghĩ thử cách O(n).

Nhận xét: giả sử số lượng số 0, 1, 2 là a, b, c → khi đó: a.b.c là luỹ thừa của 2 nên bản thân mỗi số đều cũng phải là luỹ thừa của 2. Do đó có dạng 2^x, 2^y, 2^z. Khi đó, độ dài của đoạn [L;R] sẽ có dạng: 2^x + 2^y + 2^z → tổng của 3 luỹ thừa của 2 (*)

Từ 1 đến 10^5 số lượng số k có thể biểu diễn được thành (*) không quá nhiều → khoảng vài trăm số, ta có thể tìm được hết để đưa vào vector size, kích thước d với d << n.

Khi đó, duyệt theo chỉ số L và với mỗi L, ta kiểm tra L + size[0], L + size[1], …, L + side[d-1] xem có thoả mãn điều kiện tích = luỹ thừa của 2 không?

VD: 7 = 4+2+1, 6 = 2+2+2, 10 = 8+1+1 đều là các số có dạng (*). Do đó với L cho trước → ta kiểm tra [L; L+6], [L; L+5], [L; L+9] xem có thoả mãn không?

//Chú ý: số lượng d có một biểu thức nào đó theo n, chẳng hạn số luỹ thừa 2 không vượt quá n sẽ là k = log(n). Chọn ra 3 trong đó thì số lượng <= k^3. Vì thế d <= (logn)^3.

Vì thế, độ phức tạp sẽ bị chặn trên bởi n.log(n)^3.

MỘT SỐ VẤN ĐỀ VỀ QUY HOẠCH ĐỘNG (tiếp)

dynamic programming → dp: mô tả trạng thái của lời giải của các BT con.

Có thể 1 chiều, 2 chiều, ... → trong trường hợp 1 chiều: công thức theo dạng truy hồi.

Đôi khi, ta không cần phải rút gọn công thức để tìm dạng tổng quát mà thay vào đó, ta tìm cách tính bằng các vòng lặp (max, min), ...

Các bài kinh điển về quy hoạch động (có thể tìm được ở nhiều nguồn): phân hoạch số nguyên, knapsack, bài toán cái túi, độ dài chuỗi con tăng dài nhất, tìm một chuỗi con (phần tử có thể không liên tiếp) mà tăng và dài nhất có thể có.

Bài toán cái túi (knapsack) và các phiên bản.

Có một nhà thám hiểm có 1 cái túi với sức chứa V và đột nhập vào một kho báu: có n món đồ cổ với thông tin (ai, bi) trong đó ai = giá trị còn bi = khối lượng. Hỏi nhà thám hiểm cần lấy những món đồ nào để không vượt quá sức chứa mà tổng giá trị là lớn nhất?

//Chú ý: ở đây ta xét các số ai, bi (1 <= i <= n) và V đều nguyên dương. Bài toán trong phiên bản số thực là bài khó, không thể giải bằng dp được, chỉ giải bằng cách heuristic.

Ngoài ra, ngay cả phiên bản số nguyên này, nếu giải không tốt, dùng vét cạn thì chi phí O(2^n).

Gợi ý: gọi dp[i][j] = trong đó i là chỉ số của đồ vật hiện tại đang xét đến còn j là sức chứa đang có.

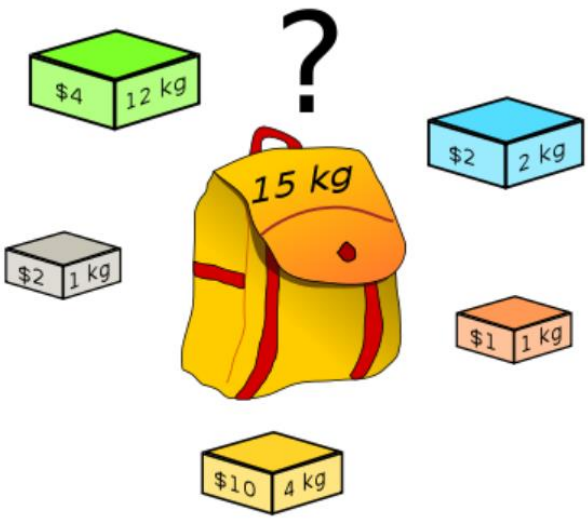
Kết quả cần tính là dp[n][V].

+ Nếu lấy đồ vật thứ i thì sao (chỉ xảy ra khi j >= b[i]) → dp[i-1][j-b[i]] + a[i].

+ Không lấy thì sao hoặc j < b[i] → dp[i-1][j].

Chú ý lấy max giữa 2 kq.

Do đó, ta cũng thực hiện 2 vòng for theo i = 1,2,…,n và j = 0,1,2,…,V.



VD1. Cho k đồng xu có mệnh giá là a1, a2, …, ak. Hãy mô tả cách đổi n đồng sang các mệnh giá sao cho số tiền thừa trả lại là ít nhất.

//Cách làm = tham lam có nguy cơ bị sai trong vài trường hợp đặc biệt. Ý tưởng tham lam: đổi theo đồng lớn nhất đến khi đổi được thì giảm xuống.

VD: {3,4} đổi n=10 → nếu tham lam thì kq = 4+4 = 8, nếu chuẩn thì kq = 3+3+3 = 9.

Gọi dp[i][j] = {0,1}, trong đó:

dp[i][j] = 0 nghĩa là không tạo được tổng = i nếu dùng các đồng từ a1, a2, …, aj.

dp[i][j] = 1 nếu tạo được.

dp[0][j]= true với mọi j = 1,2,…,k vì tổng = 0 thì ta luôn tạo được.

dp[i][j] = dp[i][j-1] or dp[i-a[j]][j] , với j>=a[i] (tức là lấy hoặc không lấy a[j]).

ans = dp[i][j] == true với i lớn nhất.

//ở đây, ta điều chỉnh lại công thức so với phiên bản gốc vì một đồng có thể được dùng nhiều lần.

VD2. Hãy biểu diễn số nguyên dương thành tổng của các số Fibonacci nhỏ hơn nó sao cho số lượng số hạng là ít nhất. Câu hỏi tương tự nếu thay bởi số nguyên tố, số chính phương, … ?

1, 2, 3, 5, 8, …

dp[i] = số lượng số hạng ít nhất các số Fibonacci cần để biểu diễn i.

Gọi F1, F2, …, Fk là các số Fibonacci <= i. Ta có:

dp[i] = min{ dp[i – F1] + 1, dp[i – F2] + 1, …, dp[i – Fk] + 1 }.

Chú ý: dp[0] = 0.

Tương tự, nếu đổi Fibonacci thành số nguyên tố → đổi công thức ở trên tương ứng, thay vì ta sinh trước các số Fibonacci thì sinh trước các SNT.

Chú ý: đôi khi, các bài toán biểu diễn này còn liên quan đến tính chất số học.

+ Giả thuyết về số nguyên tố Goldbach-Euler: mọi số chẵn >= 6 thì đều biểu diễn được thành tổng của 2 số nguyên tố lẻ; mọi số nguyên >= 6 thì đều có thể biểu diễn được thành tổng của 3 SNT bất kỳ. Nếu ta đã biết định lý này thì kết quả của bài toán đổi số thành ít SNT nhất sẽ trở nên dễ:

Nếu n là SNT → kq = 1; nếu n chẵn → kq = 2; nếu n lẻ → nếu n-2 là SNT thì kq = 2 còn nếu không thì kq = 3.

+ Nếu biểu diễn thành tổng các số chính phương (scp): định lý Lagrange nói rằng mọi số nguyên dương thì có thể biểu diễn được thành <= 4 scp và chỉ khi n có dạng 4^k.(8m+7) với m, k >= 0 thì mới cần đến 4 scp. Kết quả chỉ có 1, 2, 3, 4.

- ⇨ Khi đặt vấn đề trong Tin học, ta có thể né tránh các tính chất đặc thù như trên để tập trung vào công thức dp.

VD3. (codeforces.com)

Để làm bánh Trung thu, có n <= 10^3 kg bột và m <= 10 loại nhân với khối lượng a[1], a[2], ..., a[m]. Để làm bánh với nhân thứ i thì cần: mỗi bánh cần b[i] kg nhân, c[i] kg bột và bán được d[i] tiền. Ngoài ra, có thể làm bánh chay, không nhân tốn c[0] kg bột và bán được d[0] tiền (các khối lượng được cho đều là các số nguyên <= 100). Tính tổng tiền max.

Ta gọi dp[i][j] với i là số lượng bột, j là thứ tự nhân

k là số lượng bánh của mỗi loại nhân khi xét tới

dp[i][j] = max(dp[i][j-1], dp[i-c[j]*k][j-1] + d[j]*k) với k là số tự nhiên lớn nhất để đồng thời có c[j]*k <= i và k*b[j] <= a[j].

Với j=0 → dp[i][0] = dp[i-c[0]*k][0] + d[0] * k với k là số max để c[0] * k <= i.

Giải thích:

Không làm bánh thứ j → dp[i][j-1].

Nếu làm bánh thứ j → làm nhiều cái trong điều kiện bột và nhân cho phép (làm k cái bánh); ngoài ra, còn thừa lại n-i ký bột, ta dùng hết để làm bánh chay.

Chú ý: ở đây, ta không fixed giá trị k mà cho thêm một biến t chạy từ 0 đến k để kiểm tra vì bánh thứ i có thể không nhất thiết làm hết k cái mà chỉ làm vài cái.

Đáp số bài toán = dp[n][m].

```
for(int i = 0; i <= n; i++) dp[0][i] = d[0]*(i/c[0]);
   for(int i = 1; i <= m; i++){
      for(int j = 0; j <= n; j++){
         for(int k = 0; k*b[i] <= a[i] && k * c[i] <= j; k++)
         dp[i][j] = max(dp[i][j], dp[i-1][j-k*c[i]]+k*d[i]);
      }
   }
   cout<<dp[m][n]<<endl;
```

Quy hoạch động là một kỹ thuật mạnh, giải được nhiều dạng bài. Người ta cũng quan tâm tìm cách cải tiến việc tính toán → bao lồi, bit mask / trạng thái, …

Link nộp các BT đã thảo luận ở lớp:

https://codeforces.com/contest/106/problem/C

https://codeforces.com/contest/467/problem/C

https://codeforces.com/problemset/problem/474/D

BT về nhà.

Bài 1.

Cho mảng a có n phần tử, mỗi phần tử là 0 hoặc 1. Cho phép chọn 1 dãy con liên tiếp và đổi trạng thái của chúng (0 sang 1 và 1 sang 0), làm <= 1 lần. Tìm tổng các số trong dãy mới thu được là lớn nhất có thể bằng các cách với độ phức tạp khác nhau.

https://codeforces.com/problemset/problem/327/A

VD: n = 5 và 1,0,0,1,0; nếu đổi vị trí từ 2 đến 3 → 1,1,1,1,0 được tổng là 4;

hoặc đổi từ 2 đến 5 → 1,1,1,0,1 vẫn tổng là 4.

Ý tưởng 1: xét tất cả các dãy con → dùng 2 vòng for cho vị trí left và vị trí right của dãy con; ứng với mỗi dãy như thế thì ta sẽ tính số số 1 thu được là bao nhiêu → so sánh trong tất cả nC2 cách thì cách nào có đáp số lớn nhất → O(n^3).

Ý tưởng 2: nhận xét → nếu trong 1 dãy con đang thao tác, có a số 0 và b số 1 → tổng hiện tại là b; sau thao tác → tổng mới sẽ là a. Chênh lệch sẽ là a-b.

```
for(int i = 1; i <= n; i++) for(int j = i; j <= n; j++)
        // tính số số 1,0 trong dãy con từ i đến j sau thao tác là bao nhiêu (*)
```

// so sánh với các kết quả khác: res = max(res, current).

Để cải tiến tốc độ, ta có thể tính trước số số 0 và số số 1 (pre-compute).

ones[i] và zeros[i] → cho biết số số 1, 0 tương ứng từ vị trí 1 đến vị trí i.

1,0,0,1,0 → (ones[1]=1,zeros[1]=0), (ones[2]=1,zeros[2]=1), (ones[3]=1,zeros[3]=2), ...

Tính được trong 1 vòng lặp:

```
ones[0] = zeros[0] = 0;
for(int i = 1; i <= n; i++){
    if(a[i] == 0) ones[i]=ones[i-1], zeros[i] = zeros[i-1]+1;
    else ones[i]=ones[i-1]+1, zeros[i] = zeros[i-1];
}
```

	zeros[i] = số số 0 từ 1 đến i			ones[i] = số số 1 từ 1 đến i			
index	1	2	3	4	5		
a =	1	0	0	1	0	tổng tích lũy	accumulate sum
	one[1] = 1	one[2] = 1	one[3] = 1	one[4] = 2	one[5] = 2		
	zero[1] = 0	zero[2] = 1	zero[3] = 2	zero[4] = 2	zero[5] = 3		
	Nhận xét: one[i] + zero[i] = i						
	Dãy one[i] và zero[i] là dãy không giảm						
	Nhờ mảng này --> ta tính được: số số 1 từ index L đến index R là:					one[R] - one[L-1]	
	Tương tự với số số 0 từ L đến R là: zero[R] - zero[L-1]						

Ở (*), ta có thể tính được số số 0 và 1 trong dãy con từ i → j trong thời gian là O(1), bù lại cần có mảng để lưu trữ → trade-off.

VD: **000100010000** → ý tưởng lấy khoảng dài nhất các số 0 là ý tưởng tham lam: dễ code, dễ hiểu.

Ý tưởng 3: nhận xét: ban đầu ta coi tổng các số là S. Ta quan tâm khi tác động lên 1 số a[i] thì giá trị S biến đổi thế nào:

a[i] = 0 → S tăng 1.

a[i] = 1 → S giảm 1.

Viết mảng a đã cho lại theo giá trị mà nó sẽ ảnh hưởng đến tổng S.

VD: 1,0,0,1,0 → -1, 1, 1, -1, 1. Ban đầu, tổng S = 2.

Ta cần chọn một khoảng liên tiếp trong dãy mới này mà sao cho tổng chênh lệch là lớn nhất.

Đến đây, ta đưa về bài toán: maximum contiguous subsequences (chuỗi con tổng lớn nhất).

Ta có thuật toán chỉ cần 1 vòng lặp là giải được bài này → O(n).

```
int maxSubArraySum(int a[], int size)
{
   int max_so_far = a[0];
   int curr_max = a[0];

   for (int i = 1; i < size; i++)
   {
        curr_max = max(a[i], curr_max+a[i]);
        max_so_far = max(max_so_far, curr_max);
   }
   return max_so_far;
}
```

Lời giải hiện tại đang xét đến vị trí i – 1, khi đi đến vị trí i có 2 khả năng xảy:

- Tổng hiện tại đang tìm được < a[i] → bỏ hết phần đã tìm được và bắt đầu quá trình tính toán mới từ vị trí a[i].
- Nếu a[i] > 0 → thêm a[i] vào tổng hiện tại thì giá trị của đáp số sẽ tăng lên.

VD: index 0-base: -1, **1, 1, -1, 1, 1.**

result = a[0] = -1 và cur = a[0] = -1;

xét i = 1: so sánh a[1] = 1 và cur+a[1] = 0 → cur = 1 (bỏ hẳn phần trước, lấy từ index = 1)

cập nhật lại result = max(-1,cur) = 1.

xét i = 2: so sánh a[2] = 1 và cur+a[2] = 2 (không bỏ mà làm tiếp)

cập nhật lại result = max(1,2) = 2

xét i = 3: so sánh a[3] = -1 và cur+a[3] = 1 → cur = 1.

so sánh với result → không cập nhật.

xét i = 4: so sánh a[4] = 1 và cur+a[4] = 2 → cur = 2.

so sánh với result → không cập nhật.

xét i = 5: so sánh a[5] = 1 và cur+a[5] = 3 → cur = 3.

cập nhật lại result = 3.

Bài 2. Đếm số cách chia mảng độ dài n <= 5.10^5 thành 3 phần khác rỗng, gồm các số a[i] mà |a[i]| <= 10^9 và có tổng bằng nhau.

https://codeforces.com/contest/466/problem/C

VD: n = 5 và 1,2,3,0,3 → [1,2],[3],[0,3] hoặc [1,2],[3,0],[3] → 2 cách.

n = 4 và 0,0,0,0 → [0],[0],[0,0]; [0,0],[0],[0]; [0],[0,0],[0]; => 3 cách.

n = 3 và 1,2,3 → không thể chia được → 0 cách.

//trước khi giải bài trên, ta thử làm bài tương tự (dễ hơn) → có mấy cách chia mảng thành 2 phần mà tổng các số trong mỗi phần bằng nhau.

Gọi S = tổng các số ban đầu → ta vẫn pre-compute mảng tổng s[i] của các số từ a[1] đến a[i] (trong 1 vòng for); sau đó, ta chạy tiếp vòng for nữa và đếm các số s[i] mà s[i] = S/2 (vì nếu i số đầu có tổng là S/2 thì chắc chắn n-i số sau cũng phải là S/2).

VD: 1,2,3,0,0,6 → tổng là 12 và ta thấy s[3] = s[4] = s[5] = 6 nên có 3 cách.

Cách làm: nhận xét để chia được thành 3 mảng con thỏa mãn đề thì tổng các số trong mỗi mảng phải là S/3 (với S = tổng ban đầu → nếu tổng S không chia hết cho 3 thì thôi).

for(int i = 1; i <= n; i++){

if(s[i] == S/3) → ta cần tính xem trong khoảng từ i+1 đến n thì có bao nhiêu cách chia nó thành 2 phần có tổng bằng nhau; nhưng nếu ta thêm vòng for nữa vào đây → O(n^2) (*) }

(*) ta cần đếm xem trong các tổng a[n], a[n]+a[n-1], a[n]+a[n-1]+a[n-2], ..., a[n]+a[n-1]+...+a[i+1] thì có bao nhiêu tổng là S/3, mà:

a[n] = s[n] – s[n-1];

a[n]+a[n-1] = s[n] – s[n-2];

a[n]+a[n-1]+a[n-2] = s[n] – s[n-3]; ...

a[n]+a[n-1]+...+a[i+1] = s[n] – s[i].

Ta cần đếm xem trong các số ở trên thì có mấy số bằng S/3.

Thực ra thông tin này có thể pre-compute được chứ không cần tính trong vòng for.

Gọi r[k] = cho biết trong các số từ s[n], s[n] – s[n-1], ..., s[n] – s[k] thì có mấy số bằng s/3.

```
r[n] = 0;
for(int i = n-1; i >= 1; i--){
    if(s[n]-s[i] % 3 == 0) r[i] = r[i+1] + 1;
    else r[i] = r[i+1];
}
```

VD: n = 5 và 1,2,3,0,3 thì s[1]=1,s[2]=3,s[3]=6,s[4]=6,s[5]=9 và S = s[5] = 9.

i = 4: s[5]-s[4]=3 thỏa mãn r[4] = 1 (ý nghĩa là có 1 index thỏa mãn).

i = 3; s[5]-s[3]=3 thỏa mãn tiếp nên r[3] = r[4] + 1 (nghĩa là có 2 index thỏa).

i = 2; s[5]-s[2]=6 không thỏa; tương tự với i = 1.

https://codeforces.com/contest/255/problem/C (*1500)

https://codeforces.com/contest/431/problem/C (*1600)

https://codeforces.com/contest/789/problem/C (*1600)