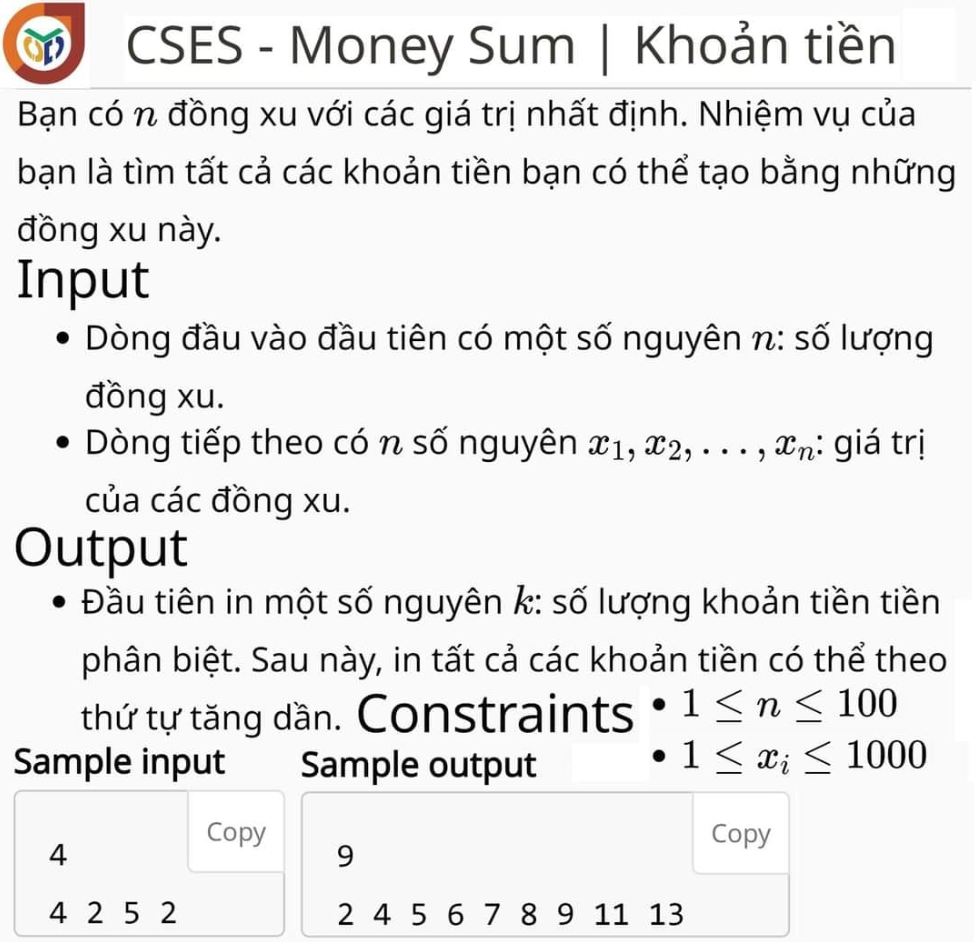
****

Bài này mình được một bạn học viên hỏi nên sẵn tiện mình viết bài giải chi tiết để nhiều bạn được tham khảo luôn. Hy vọng mang đến được giá trị cho các bạn 😇. Bạn nào muốn thử nộp bài để hệ thống chấm xem có đúng hay không thì link bài đây nhé, các bạn nào chưa có tài khoản thì tạo tài khoản rồi đăng nhập nộp bài thử hen:  
[https://lqdoj.edu.vn/problem/cses1745](https://lqdoj.edu.vn/problem/cses1745?fbclid=IwZXh0bgNhZW0CMTAAAR3tjSLkhdDCgjOwwWo8lKWrNuJ4QqjFVjCttgvIwieMnle2nco-C3sImNc_aem_zRbWEz-0S6x7cMgCrToYqA)  
---------------------------------------------------

1/ Tóm tắt

Cho bạn n đồng xu với các giá trị là a[i] (1 <= i <= n). Nhiệm vụ của bạn là tìm tất cả các khoản tiền bạn có thể tạo bằng những đồng xu này (mỗi đồng xu chỉ được dùng 1 lần).

2/ Đầu vào, giới hạn, đầu ra:  
  
Dòng đầu tiên chứa 1 số nguyên dương n: số lượng đồng xu. (1 <= n <= 100).  
  
Dòng thứ hai chứa n số nguyên dương: a[1], a[2], …, a[n] là giá trị của các đồng xu. (1 <= i <= n; 1 <= a[i] <= 1000).  
  
Ta cần in ra số khoản tiền phân biệt ta có thể tạo ra. Sau đó in tất cả các khoản tiền phân biệt có thể tạo ra theo thứ tự tăng dần.  
  
3/ Ví dụ:

n = 4; a = {4, 2, 5, 2}.  
Ouput:  
9  
2 4 5 6 7 8 9 11 13.  
----------------------------------------------  
**CÁCH VÉT CẠN:**

Cách làm đơn giản nhất của bài này chính là vét cạn. Tức ta thử hết mọi trường hợp có thể đó. Ta thấy số tiền tối đa ta tạo ra được là tổng của tất cả n đồng xu. Ở ví dụ trên thì nó chính = 4 + 2 + 5 + 2 = 13. Đó là điều hiển nhiên rồi. Và số tiền nhỏ nhất ta có thể tạo ra chính là đồng xu có giá trị nhỏ nhất trong n đồng xu. Ở ví dụ trên thì nó chính là 2. Vậy số tiền tạo tạo ra được chỉ có thể nằm trong khoảng từ giá trị của đồng xu nhỏ nhất đến tổng của tất cả n đồng xu. Giờ ta cần tạo một hàm sẽ cho ta biết khi ta sử dụng n đồng xu đầu thì có thể tạo ra số tiền là val không? Từ đó ta sẽ duyệt một vòng for() từ giá trị nhỏ nhất trong n đồng xu đến tổng của n đồng xu. Và ta sẽ dùng hàm để check xem với n đồng xu thì ta có thể tạo ra số tiền đó hay không. Ta sẽ thực hiện (xây dựng hàm đệ quy) như sau:  
  
- Gọi hàm bool Try(n, val) sẽ cho ta biết với n đồng xu đầu tiên thì ta có thể tạo ra số tiền là val được hay không. Nếu tạo được ta sẽ trả về là true; ngược lại là false.  
  
- Điều kiện dừng:  
  
1/ Ta sẽ nhận thấy; khi số tiền ta cần tạo ra (val) mà = 0 thì ta sẽ luôn luôn tạo được. Vì ta chỉ cần không lấy đồng xu nào thì ta sẽ có số tiền là 0. Vậy nếu val = 0 thì ta sẽ trả về là true. Ta thực hiện:  
if(val == 0) {  
return true;  
}  
  
2/ Tiếp tới ta thấy rằng; khi số tiền ta cần tạo ra (val) mà > 0. Nhưng ta lại không còn đồng xu nào để chọn thì chứng tỏ ta không thể tạo ra số tiền là val. Vì ta không còn xét đồng xu nào thì đâu thể tạo ra được số tiền > 0 đâu. Vậy tại đây nếu n = 0 thì ta trả về là false. Ta thực hiện:  
if(n == 0) {  
return false;  
}  
- Xử lý chạy: Tại đây ta giả sử ta đang xét đồng xu thứ n. Ta khởi tạo biến res (lưu trữ kết quả của hàm Try(n, val). Ta sẽ có 2 trường hợp là:  
  
1/ Không chọn đồng xu thứ n: Lúc này thì ta chỉ còn xét n – 1 đồng xu đầu tiên; và số tiền ta cần tạo ra vẫn là val. Để biết ta có thể tạo ra được số tiền val ở trường hợp ta chỉ còn xét

n – 1 đồng xu đầu tiên thì ta sẽ gọi hàm Try(n – 1, val). Vậy kết quả của trường hợp này là: Try(n – 1, val). Vì trường hợp này không có thêm điều kiện gì nên ta sẽ gán luôn biến

res = trường hợp này. Tức res = Try(n – 1, val).  
  
2/ Có chọn đồng xu thứ n: Để chọn được đồng xu thứ n thì ta cần 2 điều kiện là; biến res vẫn = false và val phải >= giá trị đồng xu thứ n (a[n]). Vì ở bên trên ta đã khởi tạo res = Try(n – 1, val). Nên nếu res = true (tức ta có thể tạo ra được số tiền là val rồi) thì việc gì ta phải thử thêm một trường hợp nữa; nếu làm nữa sẽ khiến ta tốn thêm thời gian chạy; vả lại nhỡ trường hợp có chọn đồng xu thứ n làm ta không thể tạo ra số tiền là val thì sao; lúc đó res đang = true rồi mà lại bị gán lại = false thì sẽ khiến kết quả của hàm bị chạy sai mất. Từ đó ta phải check res vẫn phải = false. Tiếp tới là val phải >= a[n]. Dĩ nhiên là số tiền ta cần tạo ra phải lớn hơn hoặc bằng đồng xu ta định chọn thì ta mới có thể lấy đồng xu đó rồi; ta không thể chọn đồng xu có giá trị = 4 khi ta đang cần tạo ra số tiền = 2 được.  
  
Khi đã đều thoả mãn 2 điều kiện trên; tức là việc không chọn đồng xu thứ n (Try(n – 1, val)) thì không thể tạo ra số tiền là val; và số tiền cần tạo ra >= đồng xu ta đang định chọn. Thì ta thấy để tạo ra số tiền là val từ n đồng xu đầu tiên tại trường hợp ta có lấy đồng xu thứ n thì ta cần biết: với n – 1 đồng xu đầu tiên; ta có thể tạo ra số tiền là val – a[n] hay không. Vì với n – 1 đồng xu đầu tiên mà ta tạo được số tiền có giá trị là val – a[n] thì ta chỉ việc thêm đồng xu thứ n (có giá trị là a[n]) vào thì ta sẽ có được số tiền là val rồi. Ngược lại nếu không thể tạo ra thì cũng đồng nghĩa với việc ta không thể tạo ra số tiền là val rồi. Và để biết: với n – 1 đồng xu đầu tiên; ta có thể tạo ra số tiền là val – a[n] không thì ta sẽ gọi hàm Try(n – 1, val – a[n]). Vì đó chính là công dụng của hàm Try() mà ta đang tạo ra. Vậy tại trường hợp này thì res = Try(n – 1, val – a[n]). Ta sẽ thực hiện như sau:  
bool res = Try(n - 1, val);  
if(!res && val >= a[n]) {  
res = Try(n - 1, val - a[n]);  
}  
  
Giải thích: if(!res) thì chính là if(res == false) đó. Trong c++; khi ta viết không thế này: if(res) thì nó sẽ hiểu là if(res != 0) hoặc chính là (res == true). Khi ta thêm dấu ! (có nghĩa là phủ định) vào thì nó sẽ bị đảo ngược lại. Tức là if(res == 0) hoặc chính là (res == false). Cuối cùng ta chỉ cần trả về biến res (kết quả của hàm Try(n, val)) là được rồi.  
  
**Sau đây là Source Code của mình về ý tưởng trên cho bạn nào cần tham khảo nha:**  
  
[**#include**](https://www.facebook.com/hashtag/include?__eep__=6&__cft__%5b0%5d=AZViWUdTJrghX2ZGzBh91WbbNqTmGAjHCRgF_ZKAR35-56KfuAnvyuH4MWtd5zdlJmavLhqvW-5nQqLEfVrZwLVPsbh8UtEe-I3hQ1T2AJeqQXi-AJ_l3Wq71VsPUBFBnIxNVOCMJfsmUbK0avsOBA0jxmvoYb0Cqr3eatql_KJBV-82qmhiRH0HNawqWAP_iYNGlxvYl0lC5g01IN8gm5kWN6oElRf0SjAVKuO3TLflNA&__tn__=*NK*F)<bits/stdc++.h>  
[**#define**](https://www.facebook.com/hashtag/define?__eep__=6&__cft__%5b0%5d=AZViWUdTJrghX2ZGzBh91WbbNqTmGAjHCRgF_ZKAR35-56KfuAnvyuH4MWtd5zdlJmavLhqvW-5nQqLEfVrZwLVPsbh8UtEe-I3hQ1T2AJeqQXi-AJ_l3Wq71VsPUBFBnIxNVOCMJfsmUbK0avsOBA0jxmvoYb0Cqr3eatql_KJBV-82qmhiRH0HNawqWAP_iYNGlxvYl0lC5g01IN8gm5kWN6oElRf0SjAVKuO3TLflNA&__tn__=*NK*F) endl '\n'  
  
using namespace std;  
typedef long long ll;  
  
int a[105];  
  
bool Try(int n, int val) {  
if(val == 0) {  
return true;  
}  
if(n == 0) {  
return false;  
}  
bool res = Try(n - 1, val);  
if(!res && val >= a[n]) {  
res = Try(n - 1, val - a[n]);  
}  
return res;  
}  
  
int main() {  
ios::sync\_with\_stdio(0);  
cin.tie(0);  
  
int n;  
cin >> n;  
int sum = 0;  
for(int i = 1; i <= n; ++i) {  
cin >> a[i];  
sum += a[i];  
}  
vector<int> res;  
for(int i = \*min\_element(a + 1, a + n + 1); i <= sum; ++i) {  
if(Try(n, i)) {  
res.push\_back(i);  
}  
}  
cout << (int)res.size() << endl;  
for(int x : res) {  
cout << x << " ";  
}  
return 0;  
}  
  
**Giải thích 1 số chỗ trong code ở trên:**  
  
1/ Biến sum mình dùng để lưu tổng giá trị của n đồng xu. Ta thấy có tối đa 100 đồng xu; và mỗi đồng xu có giá trị tối đa = 1000. Từ đó tổng giá trị tối đa của n đồng xu là 100 \* 1000 = 10^5. Vậy ta khai báo biến sum ở kiểu dữ liệu int (có thể chứa giá trị tối đa là 2^31 – 1).  
  
2/ vector<int> res; Ta sẽ dùng vector để lưu kết quả nha. Nó sẽ tiện cho ta việc biết có bao nhiêu phần tử (bao nhiêu khoản tiền phân biệt) có thể tạo ra. Và ta có thể in ra các khoản tiền được lưu trong vector nữa.  
  
3/ int i = \*min\_element(a + 1, a + n + 1); Mình dùng hàm min\_element(start, end) sẽ trả về con trỏ trỏ đến giá trị nhỏ nhất trong đoạn từ start đến trước end. Từ đó \*min\_element(a + 1, a + n + 1) chính là giá trị nhỏ nhất trong đoạn từ a[1] tới a[n]. Nếu bạn nào không biết tới hàm này thì cũng có thể tạo 1 biến minA khởi tạo = INT\_MAX (bằng một giá trị rất lớn; cụ thể là 2^31 – 1). Để khi ta đọc đầu vào thì ta sẽ tiện tìm luôn đồng xu nào có giá trị nhỏ nhất cũng được. Độ phức tạp của cả hai cách trên đều là O(n) nha.  
  
4/ cout << (int)res.size() << endl; Đây chính là ta in ra số khoản tiền phân biệt ta có thể tạo ra được.  
  
5/ for(int x : res); Tại đây mình sử dùng vòng lặp for each trong c++. Vòng lặp này sẽ gán lần lượt x = res[i] (0 <= i < res.size()). Từ đó mình chỉ việc in ra x là được. Nếu anh em nào mà không biết thì ta hãy duyệt vector bằng 1 vòng for như bình thường cũng được. Độ phức tạp thời gian của nó là như nhau nha.  
  
Với cách trên chắc chắn ta sẽ bị tle (time limit exceed – quá giới hạn thời gian); cụ thể ta ăn được 6 / 12 test nha. Tuy nhiên hãy vẫn nên cài đặt nó để sau này ta lấy nó làm công cụ kiểm tra tính chính xác cho cách làm tối ưu hoặc biết đâu ta có thể cải tiến nó thành quy hoạch động topdown. Bạn nào còn chưa biết cách lấy nó làm công cụ kiểm tra tính chính xác thì có thể xem qua bài viết này của mình nha:  
[**https://www.facebook.com/100003824621962/posts/2303526743118124/?d=n**](https://www.facebook.com/nvnamson/posts/pfbid02gmPYQjnnUvqgeu1TJZN6s7tX5hrYECEqvxVaccAantYJqaEcQiXwF5Np4vLEiJjfl?__cft__%5b0%5d=AZViWUdTJrghX2ZGzBh91WbbNqTmGAjHCRgF_ZKAR35-56KfuAnvyuH4MWtd5zdlJmavLhqvW-5nQqLEfVrZwLVPsbh8UtEe-I3hQ1T2AJeqQXi-AJ_l3Wq71VsPUBFBnIxNVOCMJfsmUbK0avsOBA0jxmvoYb0Cqr3eatql_KJBV-82qmhiRH0HNawqWAP_iYNGlxvYl0lC5g01IN8gm5kWN6oElRf0SjAVKuO3TLflNA&__tn__=-UK*F)  
----------------------------------------------  
**CÁCH QUY HOẠCH ĐỘNG TOPDOWN:**

Với code trên ta hoàn toàn có thể cải tiến thành quy hoạch động topdown – hay còn gọi là đệ quy có nhớ. Lý do code phần trên của ta chạy lâu chính là vì nếu ta đã tính 1 vấn đề rồi mà lần sau có gặp lại thì ta vẫn phải đi tính lại nó; việc tính đi tính lại này khiến ta tốn rất nhiều thời gian. Từ đó tư tưởng của thuật toán là dùng 1 kiểu dữ liệu (có thể là mảng, vector, map, …) để lưu trữ những kết quả ta đã tính toán; và khi ta gặp những vấn đề mà ta đã tính thì ta chỉ việc trả về kết quả mà thôi (không phải tính lại nữa). Ta sẽ làm như sau:  
  
1/ Xác định mỗi lần gọi đệ quy thì có những tham số nào có thể thay đổi:  
  
Ở hàm Try(n, val) có hai tham số đầu vào là n và val. Và cả hai tham số n và val đều có thể thay đổi ở mỗi lần ta gọi đệ quy. Vậy ta sẽ tạo một mảng 2 chiều F; với phần tử F[n][val] sẽ lưu trữ kết quả của hàm Try(n, val).  
  
2/ Xác định giới hạn của các tham số có thể thay đổi:  
  
Ta thấy n có thể nên tới tối đa là = 100; val thì có thể lên tới tối đa là = sum ( = 10^5). Vậy ta sẽ tạo một mảng 2 chiều là F[105][10^5 + 5]; mình thường cộng thêm 5 vào để thừa ra vài phần tử (tránh ta truy cập vào phần tử không có trong mảng; từ đó bị lỗi chương trình).  
  
3/ Xác định giá trị không thể trả về của hàm:  
Ta thấy hàm của ta mang kiểu dữ liệu bool; vậy chỉ có thể trả về là true và flase. Tức 0 và 1. Vậy hàm không thể trả về giá trị -1; ta sẽ tạo mảng F là một mảng theo kiểu dữ liệu int để lúc đầu ta lấp đầy mảng = giá trị -1. Sau đó; ở sau mục điều kiện dừng ta sẽ check nếu F[n][val] != -1 thì chứng tỏ ta đã tính Try(n, val) rồi và lưu kết quả ở F[n][val]; lúc đó ta sẽ trả về là F[n][val] luôn. Ngược lại; tức F[n][val] = -1; có nghĩa ta chưa tính thì ta sẽ đi tính như bình thường thôi. Đến lúc cuối ta chỉ cần gán F[n][val] = kết quả của hàm rồi trả về là được.  
  
**Sau đây là Source Code của mình về ý tưởng trên cho bạn nào cần tham khảo nha:**  
  
[**#include**](https://www.facebook.com/hashtag/include?__eep__=6&__cft__%5b0%5d=AZViWUdTJrghX2ZGzBh91WbbNqTmGAjHCRgF_ZKAR35-56KfuAnvyuH4MWtd5zdlJmavLhqvW-5nQqLEfVrZwLVPsbh8UtEe-I3hQ1T2AJeqQXi-AJ_l3Wq71VsPUBFBnIxNVOCMJfsmUbK0avsOBA0jxmvoYb0Cqr3eatql_KJBV-82qmhiRH0HNawqWAP_iYNGlxvYl0lC5g01IN8gm5kWN6oElRf0SjAVKuO3TLflNA&__tn__=*NK*F)<bits/stdc++.h>  
[**#define**](https://www.facebook.com/hashtag/define?__eep__=6&__cft__%5b0%5d=AZViWUdTJrghX2ZGzBh91WbbNqTmGAjHCRgF_ZKAR35-56KfuAnvyuH4MWtd5zdlJmavLhqvW-5nQqLEfVrZwLVPsbh8UtEe-I3hQ1T2AJeqQXi-AJ_l3Wq71VsPUBFBnIxNVOCMJfsmUbK0avsOBA0jxmvoYb0Cqr3eatql_KJBV-82qmhiRH0HNawqWAP_iYNGlxvYl0lC5g01IN8gm5kWN6oElRf0SjAVKuO3TLflNA&__tn__=*NK*F) endl '\n'  
  
using namespace std;  
typedef long long ll;  
  
int a[105];  
int f[105][(int)1e5 + 5];  
  
bool Try(int n, int val) {  
if(val == 0) {  
return true;  
}  
if(n == 0) {  
return false;  
}  
if(f[n][val] != -1){  
return f[n][val];  
}  
bool res = Try(n - 1, val);  
if(!res && val >= a[n]) {  
res = Try(n - 1, val - a[n]);  
}  
return f[n][val] = res;  
}  
  
int main() {  
ios::sync\_with\_stdio(0);  
cin.tie(0);  
  
int n;  
cin >> n;  
int sum = 0;  
for(int i = 1; i <= n; ++i) {  
cin >> a[i];  
sum += a[i];  
}  
fill(&f[1][1], &f[n][sum + 1], -1);  
vector<int> res;  
for(int i = \*min\_element(a + 1, a + n + 1); i <= sum; ++i) {  
if(Try(n, i)) {  
res.push\_back(i);  
}  
}  
cout << (int)res.size() << endl;  
for(int x : res) {  
cout << x << " ";  
}  
return 0;  
}  
  
**Giải thích 1 số chỗ trong code ở trên:**  
  
1/ return f[n][val] = res; Đây chính là ta gán f[n][val] = res rồi ta return f[n][val].  
  
2/ fill(&f[1][1], &f[n][sum + 1], -1); Đây chính là thao tác ta lấp đầy mảng F = -1. Bạn nào mà không biết tới hàm fill thì có thể dùng 2 vòng for() lồng nhau cũng được nha; độ phức tạp thời gian của nó là như nhau mà thôi.  
  
Đánh giá độ phức tạp của cách làm trên:  
  
- Độ phức tạp không gian: O(n \* sum) với sum là tổng n đồng xu trong mảng. Thực chất chính là O(n \* sum + n) vì ta dùng một mảng f và một mảng a. Nhưng theo quy tắc bigO thì nó sẽ là O(n \* sum).  
  
- Độ phức tạp thời gian: O(n \* sum). Vì trong trường hợp tệ nhất thì ta sẽ phải lấp đầy toàn bộ mảng F. Từ đó độ phức tạp thời gian của ta là O(n \* sum).  
  
Với cách làm trên là ta đã ac bài này rồi. Tuy nhiên thời gian chạy của ta vẫn còn rất lâu do phải tốn thời gian và bộ nhớ cho việc gọi đệ quy; và bộ nhớ của ta cũng ăn rất nhiều.  
--------------------------------  
**CÁCH QUY HOẠCH ĐỘNG BOTTOM UP**:

Ta sẽ cải tiến hơn bằng cách sử dụng quy hoạch động bottom up. Chính là anh em sẽ tạo ra công thức truy hồi đó:  
  
Cho ae nào đã biết cơ bản về quy hoạch động rồi thì ta có 4 bước như sau:  
  
Bước 1: Xác định bài toán con:  
  
Gọi F[i][j] sẽ cho ta biết với i đồng xu đầu tiên thì ta có thể tạo ra khoản tiền là j được hay không. Nếu F[i][j] = true tức ta có thể tạo ra khoản tiền là j từ i đồng xu đầu tiên; ngược lại nếu F[i][j] = false thì ta không thể tạo ra khoản tiền là j từ i đồng xu đầu tiên.  
  
Bước 2: Xác định bài toán cơ sở:  
  
Đầu tiên là ta thấy khi số tiền ta cần là = 0 (tức j = 0) thì ta luôn tạo được khoản tiền đó. Vì chỉ cần không chọn đồng xu nào là ta đã có được số tiền = 0. Vậy F[i][0] = true với (0 <= i <= n).  
Tiếp theo là khi ta không xét đồng xu nào (i = 0) mà khoản tiền ta cần lại > 0 (j > 0) thì dĩ nhiên là ta không thể tạo ra được khoản tiền đó. Vậy F[0][j] = false với (1 <= j <= sum; sum chính là tổng giá trị của n đồng xu).  
  
Để cho tiện thì ta chỉ việc khai báo mảng F là mảng toàn cục là các phần tử trong F sẽ được khởi tạo sẵn = 0 (tức là = false) đó. Sau đó ta chỉ cần làm một vòng for để gán F[i][0] = true là được.  
  
Bước 3: Xác định đáp án của bài toán:  
  
Ta cần biết rằng với n đồng xu thì ta sẽ tạo ra được bao nhiêu khoản tiền. Ta sẽ xét F[n][j] với (1 <= j <= sum); nếu F[n][j] = true thì chứng tỏ ta có thể tạo ra khoản tiền là j với n đồng xu. Lúc đó ta lưu khoản tiền j vào 1 vector kết quả. Ngược lại F[n][j] = false thì ta không thể tạo ra khoản tiền là j với n đồng xu. Vậy để có được đáp án của bài toán thì ta sẽ cần duyệt một vòng for() cho j chạy từ 1 tới sum để xem ta có thể tạo ra số tiền là j với n đồng xu hay không.  
  
Bước 4: Xác định công thức truy hồi:  
  
Tại đây ta sẽ có 2 trường hợp là:  
  
1/ Không chọn đồng xu thứ i: Lúc này để tạo ra khoản tiền là j thì ta chỉ được dùng i – 1 đồng xu đầu tiên. Vậy ta cần biết với i – 1 đồng xu đầu tiên thì ta có thể tạo ra khoản tiền là j được không? Và nó chính là F[i – 1][j]. Suy ra tại trường hợp này thì F[i][j] = F[i – 1][j].  
  
2/ Có chọn đồng xu thứ i: Đến trường hợp 2 này ta cũng phải thoả mãn 2 điều kiện; đầu tiên là F[i][j] phải = false. Vì nếu tại trường hợp 1; F[i][j] đã bằng true rồi thì ta không cần xét đến trường hợp này nữa. Tiếp theo là j >= a[i]; tức số tiền ta cần phải >= giá trị của đồng xu ta lấy. Ta không thể lấy đồng xu có giá trị = 5 khi số tiền ta cần là 4 được. Vậy nếu thoả mãn 2 điều kiện trên thì ta sẽ cần biết với i – 1 đồng xu đầu tiên thì ta có thể tạo ra khoản tiền là j – a[i] không? Và nó chính là F[i – 1][j – a[i]]. Vì nếu với i – 1 đồng xu đầu mà ta tạo ra khoản tiền là j – a[i] thì ta chỉ cần thêm đồng xu thứ i (có giá trị là a[i]) vào là ta có thể tạo được khoản tiền là j với i đồng xu đầu tiên rồi. Suy ra tại trường hợp này thì F[i][j] = F[i – 1][j – a[i]].  
  
Kết luận lại ta có công thức truy hồi là:  
F[i][j] = F[i – 1][j].  
Nếu F[i][j] = false và j >= a[i]: F[i][j] = F[i – 1][j – a[i]].  
  
**Sau đây là Source Code của mình về ý tưởng trên cho bạn nào cần tham khảo nha:**  
  
[**#include**](https://www.facebook.com/hashtag/include?__eep__=6&__cft__%5b0%5d=AZViWUdTJrghX2ZGzBh91WbbNqTmGAjHCRgF_ZKAR35-56KfuAnvyuH4MWtd5zdlJmavLhqvW-5nQqLEfVrZwLVPsbh8UtEe-I3hQ1T2AJeqQXi-AJ_l3Wq71VsPUBFBnIxNVOCMJfsmUbK0avsOBA0jxmvoYb0Cqr3eatql_KJBV-82qmhiRH0HNawqWAP_iYNGlxvYl0lC5g01IN8gm5kWN6oElRf0SjAVKuO3TLflNA&__tn__=*NK*F)<bits/stdc++.h>  
[**#define**](https://www.facebook.com/hashtag/define?__eep__=6&__cft__%5b0%5d=AZViWUdTJrghX2ZGzBh91WbbNqTmGAjHCRgF_ZKAR35-56KfuAnvyuH4MWtd5zdlJmavLhqvW-5nQqLEfVrZwLVPsbh8UtEe-I3hQ1T2AJeqQXi-AJ_l3Wq71VsPUBFBnIxNVOCMJfsmUbK0avsOBA0jxmvoYb0Cqr3eatql_KJBV-82qmhiRH0HNawqWAP_iYNGlxvYl0lC5g01IN8gm5kWN6oElRf0SjAVKuO3TLflNA&__tn__=*NK*F) endl '\n'  
  
using namespace std;  
typedef long long ll;  
  
int a[105];  
bool f[105][(int)1e5 + 5];  
  
int main() {  
ios::sync\_with\_stdio(0);  
cin.tie(0);  
  
int n;  
cin >> n;  
int sum = 0;  
for(int i = 1; i <= n; ++i) {  
cin >> a[i];  
sum += a[i];  
}  
for(int i = 0; i <= n; ++i) {  
f[i][0] = true;  
}  
for(int i = 1; i <= n; ++i) {  
for(int j = 1; j <= sum; ++j) {  
f[i][j] = f[i - 1][j];  
if(!f[i][j] && j >= a[i]) {  
f[i][j] = f[i - 1][j - a[i]];  
}  
}  
}  
vector<int> res;  
for(int j = 1; j <= sum; ++j) {  
if(f[n][j]) {  
res.push\_back(j);  
}  
}  
cout << (int)res.size() << endl;  
for(int x : res) {  
cout << x << " ";  
}  
return 0;  
}  
  
Đánh giá độ phức tạp của cách làm trên:  
  
- Độ phức tạp không gian (Space Complexity): O(n \* sum) với sum là tổng giá trị của n đồng xu.  
  
- Độ phức tạp thời gian (Time Complexity): O(n \* sum). Vì ta phải lấp đầy mảng F.  
  
Dù độ phức tạp thời gian và không gian của ta là giống hệ với cách quy hoạch động topdown nhưng code của ta đã chạy nhanh hơn và ăn ít bộ nhớ hơn rất nhiều so với cách cũ vì ta đã không phải tốn thời gian và bộ nhớ cho việc gọi đệ quy nữa.  
----------------------------------------------  
**SỬ DỤNG MẢNG:**  
  
Đến với cách này thì ta có thể cải tiến hơn nữa bằng việc thay vì sử dụng mảng 2 chiều thì ta chỉ sử dụng mảng 1 chiều mà thôi. Ở cách này này thì độ phức tạp không gian của ta sẽ được giảm xuống rất nhiều; còn độ phức tạp thời gian thì vẫn sẽ giữ nguyên. Tuy nhiên thời gian chạy vẫn sẽ nhanh hơn vì việc truy xuất phần tử trong mảng 1 chiều sẽ nhanh hơn so với mảng 2 chiều.

1/ Ở công thức cũ ta có:  
F[i][j] = F[i – 1][j].  
Nếu F[i][j] = false và j >= a[i]: F[i][j] = F[i – 1][j – a[i]].  
  
Ta thấy F[i][j] sẽ đều phục thuộc vào [i – 1]. Vậy ta bỏ [i] đi để thành F[j] mà thôi. Lúc này ta chỉ cần một mảng 1 chiều bool F[sum] mà thôi. (sum là tổng giá trị của n đồng xu).  
  
2/ Trường hợp cơ sở của ta là:  
  
F[0][j] (1 <= j <= sum) = false. Vậy giờ ta sửa thành F[j] = false (1 <= j <= sum). Từ đó ta chỉ cần khai báo mảng F là biến toàn cục là các phần tử đã được khởi tạo sẵn = false.  
F[i][0] = true. Ta đã bỏ [i] nên nó sẽ thành là F[0] = true.  
  
3/ Đáp án của bài toán:  
  
Lúc trước ta sẽ xét toàn bộ F[n][j] (1 <= j <= sum). Vậy giờ sau khi chạy xong phần xử lý (áp dụng công thức truy hồi). Thì ta chỉ cần làm một vòng for(j) và kiểm tra F[j] mà thôi. Lúc này F[j] sẽ cho ta biết với n đồng xu ta có thể tạo ra khoản tiền là j hay không.  
  
4/ Công thức truy hồi:  
  
Để dễ hình dung phần này thì các bạn có thể hiểu là cái mà ta đang xét và chưa xét thì chính là của quá khứ (của [i – 1]). Còn cái mà ta đã xét thì chính là của hiện tại (của [i]).  
  
Ta đã bỏ đi [i] nên ta sẽ biến đổi công thức truy hồi như sau:  
F[i][j] = F[i – 1][j] sẽ thành F[j] = F[j].  
  
Vậy ta sẽ không cần áp dụng công thức này nữa vì F[j] mà ta đang xét bản chất chính là F[i – 1][j] luôn rồi. (cái đang xét chính là của quá khứ).  
  
F[i][j] = F[i – 1][j – a[i]] với điều kiện F[i][j] = false và j >= a[i].  
  
Công thức trên sẽ thành: F[j] = F[j – a[i]] với điều kiện F[j] = false và j >= a[i].  
  
Và để ứng dụng công thức này thì ta cần quan tâm đến chiều duyệt của j. Nếu mà j ta duyệt tăng dần. Thì F[j – a[i]] thì chính là cái ta đã xét; mà cái ta đã xét bản chất chính là của hiện tại (chính là F[i][j – a[i]]). Nhưng cái ta cần phải là F[i – 1][j – a[i]] cơ. Để giải quyết vấn đề trên thì ta sẽ duyệt ngược lại; tức j sẽ giảm dần. Từ đó F[j – a[i]] sẽ chính là cái mà ta chưa xét; cái chưa xét thì chính là của quá khứ (bản chất chính là F[i – 1][j – a[i]]); và nó chính là cái ta cần. Vậy để áp dụng được công thức trên thì vòng for(j) của ta phải đảo ngược; kết hợp với điều kiện j >= a[i] thì ta sẽ có vòng for(j) chạy ngược từ sum chạy về a[i].  
  
**Sau đây là Source Code của mình về ý tưởng trên cho bạn nào cần tham khảo nha:**

[**#include**](https://www.facebook.com/hashtag/include?__eep__=6&__cft__%5b0%5d=AZViWUdTJrghX2ZGzBh91WbbNqTmGAjHCRgF_ZKAR35-56KfuAnvyuH4MWtd5zdlJmavLhqvW-5nQqLEfVrZwLVPsbh8UtEe-I3hQ1T2AJeqQXi-AJ_l3Wq71VsPUBFBnIxNVOCMJfsmUbK0avsOBA0jxmvoYb0Cqr3eatql_KJBV-82qmhiRH0HNawqWAP_iYNGlxvYl0lC5g01IN8gm5kWN6oElRf0SjAVKuO3TLflNA&__tn__=*NK*F)<bits/stdc++.h>  
[**#define**](https://www.facebook.com/hashtag/define?__eep__=6&__cft__%5b0%5d=AZViWUdTJrghX2ZGzBh91WbbNqTmGAjHCRgF_ZKAR35-56KfuAnvyuH4MWtd5zdlJmavLhqvW-5nQqLEfVrZwLVPsbh8UtEe-I3hQ1T2AJeqQXi-AJ_l3Wq71VsPUBFBnIxNVOCMJfsmUbK0avsOBA0jxmvoYb0Cqr3eatql_KJBV-82qmhiRH0HNawqWAP_iYNGlxvYl0lC5g01IN8gm5kWN6oElRf0SjAVKuO3TLflNA&__tn__=*NK*F) endl '\n'  
  
using namespace std;  
typedef long long ll;  
  
int a[105];  
bool f[(int)1e5 + 5];  
  
int main() {  
ios::sync\_with\_stdio(0);  
cin.tie(0);  
  
int n;  
cin >> n;  
int sum = 0;  
for(int i = 1; i <= n; ++i) {  
cin >> a[i];  
sum += a[i];  
}  
f[0] = true;  
for(int i = 1; i <= n; ++i) {  
for(int j = sum; j >= a[i]; --j) {  
if(!f[j]) {  
f[j] = f[j - a[i]];  
}  
}  
}  
vector<int> res;  
for(int j = 1; j <= sum; ++j) {  
if(f[j]) {  
res.push\_back(j);  
}  
}  
cout << (int)res.size() << endl;  
for(int x : res) {  
cout << x << " ";  
}  
return 0;  
}  
  
**Giải thích 1 số chỗ trong code ở trên:**  
  
1/ for(int j = sum; j >= a[i]; --j) if(!f[j]). Nếu bạn nào thấy đoạn này hơi khó hiểu thì bản chất chính là ta đảo ngược vòng for(j) cho chạy từ sum về đến 1. Ở bên trong thì ta check if(!f[j] && j >= a[i]). Và vì giờ ta không còn phải áp dụng công thức f[i][j] = f[i – 1][j] nữa (vì ta đã bỏ đi [i]) nên ta cho luôn điều kiện j >= a[i] vào trong vòng for(). Từ đó vòng for(j) của ta thay vì chạy từ sum về 1 thì chỉ cần chạy từ sum về a[i] mà thôi; bên trong thì chỉ còn phải check if(!f[j]).  
  
Đánh giá độ phức tạp của cách làm trên:  
  
- Độ phức tạp không gian: O(sum). Thực chất chính là O(sum + n) vì ta dùng một mảng f và một mảng sum. Nhưng theo quy tắc bigO thì nó sẽ là O(sum).  
  
- Độ phức tạp thời gian: O(n \* sum). Tuy độ phức tạp thời gian của ta vẫn giống cách cũ nhưng thực tế nó sẽ chạy nhanh hơn đó nha; các bạn chỉ cần nhìn ở phần thời gian code chạy là sẽ thấy rõ😁.  
  
Rất vui khi anh em nào đã đọc đến những dòng cuối cùng này. Anh em đọc chắc cũng mỏi mắt luôn chứ 😅 vậy anh em nghĩ người viết ra hết đống ở trên còn thế nào nữa 🤪 . Vậy nên đừng tiếc gì cho mình một like nha để động viên mình nè. Chúc anh em học tốt và gặt hái nhiều thành công 😇.