BAC Round 1 2017
Editorial

February 27, 2017

Square

Just generate all permutations of $[1, 2, 3, 4, 5, 6, 7, 8, 9]$ and for each of the check that the sums are all 15. If you used the internet to solve it, shame on you!

Sum

This is a classic subset-sum problem and is indeed NP-complete. It can be solved with Dynamic Programming in $O(n \cdot t)$ where n is the number of integers in the set and t is the target sum. However the constraints don't allow such a solution.

However the constraint

$$s_i > s_1 + s_2 + \ldots + s_{i-1}$$

makes the problem much easier.
Suppose that $s_i \le t$ and $s_{i+1} > t$. If you don't choose s_i then you will choose a subset of $\{s_1, s_2, \ldots, s_{i-1}\}$ but this subset must have a sum at most $s_1 + s_2 + \ldots + s_{i-1}$. But

$$s_1 + s_2 + \ldots + s_{i-1} < s_i \le t$$

So you will never be able to reach t. Hence you always have to take such s_i. You subtract it from t and continue. If at the end $t = 0$ the answer is yes.

Bins

Apparently this is just a greedy problem. But I did not see it when I set the problem. It was supposed to be harder than that... Anyway. Here is the indented solution:
Let b_i be the number of tickets initially on the i-th bin and x_i be the number of tickets we put on bin i.

First observe that if some $b_i = 0$ then we win a prize with probability 1 as long as we have at least one ticket. If we have 0 tickets then the probability of winning at least one prize is 0.

We can treat these special cases apart and assume from now one that $b_i > 0$ for all i. Then the probability that we *fail* to gain the i-th prize is:

$$\frac{b_i}{b_i + x_i}.$$

Therefore, the probability of *failing* to gain all the prizes is

$$\prod_{i=1}^{n} \frac{b_i}{b_i + x_i}.$$

Hence the probability of *gaining* at least one prize is:

$$1 - \prod_{i=1}^{n} \frac{b_i}{b_i + x_i}.$$

Define

$$dp[i][j] = \text{to be the minimum probability of failing to gain a prize if we consider only bins } 0, 1, \ldots, i \text{ and having } j \text{ tickets to spend.}$$

Clearly $dp[i][0] = 1$ because we have no tickets to spend. Also, $dp[0][j] = \frac{b_0}{b_0+j}$ since, as there is only one bin, we just put all our tickets into it. Now, in general, we can write

$$dp[i][j] = \min_{x=0,1,\ldots,j} \quad dp[i-1][j-x] \cdot \frac{b_i}{b_i + x}.$$

Computing this in $O(n \cdot k^2)$ is straightforward.
We can accelerate it to $O(\log(n) \cdot k^2)$ by sorting the bins so that $b_0 \leq b_1 \leq \ldots \leq b_{n-1}$.

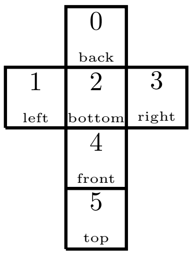
Observe that in this case a solution is such that $x_0 \leq x_1 \leq \ldots \leq x_{n-1}$.
In particular, if there are i bins and we have j tickets we will put at most j/i tickets on the last bin.
Hence we can write:

$$dp[i][j] = \min_{x=0,1,\ldots,\mathbf{i/j}} \quad dp[i-1][j-x] \cdot \frac{b_i}{b_i + x}.$$

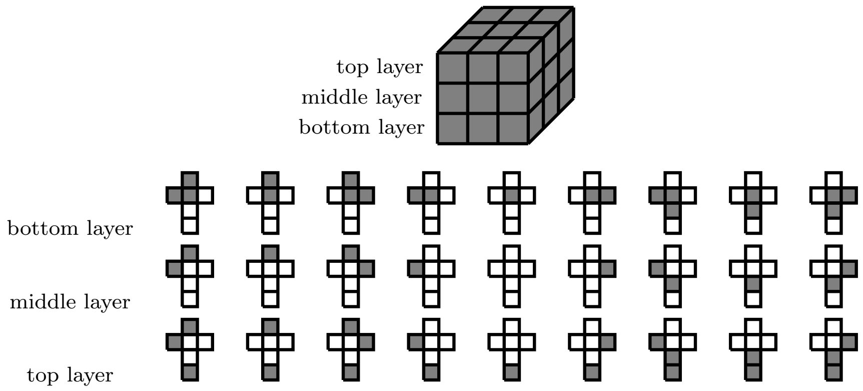
Computing this is $O(n \cdot k + \log(n) \cdot k^2)$ because $1/1 + 1/2 + \ldots + 1/n = O(\log(n))$.
There are other optimizations you can do. You can just consider the best k bins and this gets you a solution that is $O(k^3)$.

Cubes

We can flatten the cube and index the sides as in the picture:

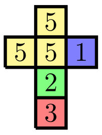


Consider a $3 \times 3 \times 3$ cube. If we cut it into 27 $1 \times 1 \times 1$ cubes, there are most three sides with their color preserved depending on the one that is selected. The following picture describes, for each of the 27 cubes, which side have their color preserved (the gray faces are preserved). The cubes are show from the bottom layer to the top layer, back to front and left to right.

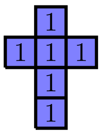


To solve this problem, think about what happens after the k-th cut. Let the colors in the given cube be $c_1 < c_2 < \ldots < c_n$. The faces with a color smaller that c_k need to be preserved (needs to be gray on the pattern). The face with color c_k needs to be changed (need to be white on the pattern). We don't care about what happens to the other faces because the will be replaced later by another color later. Hence we need to check that at each step there is one of the 27 patterns that does this.

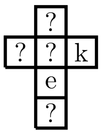
For example, suppose the cube we are given is $[5, 5, 5, 1, 2, 3]$ (note that there is not face of color 4 but that does not matter):



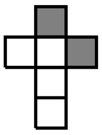
For the first color there is nothing to check because we start by painting all cells in blue so it is obviously possible to have the right cell in blue at the end. So we start with:



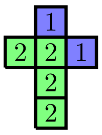
Now we look at color 2, green. We need a pattern that erases (e) the blue color from the front cell but keeps (k) the blue color from the right cell. We don't care about what it does to the others (?):



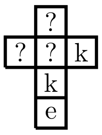
So any pattern that is gray on the right cell and white on the front one will do. For instance:



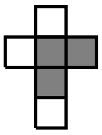
After doing this cut we obtain:



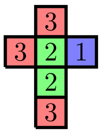
We now look at color 2. For this color we need a pattern that preserves the right cell and the front cell but changes the top one. So a patten of the form:



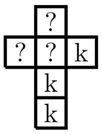
One possibility is the following cut:



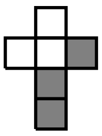
This results in:



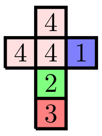
Finally, for the next cut we need to preserve the right, front and top cells. We don't care about what happens to the others because there is no color 4 in the final cube.



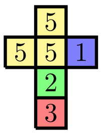
There is only one cut that does this:



And this gives us the desired cube:



Repeating this exact cut changes all the 4 into 5 as we get our target cube.



Graph

This sequence representation of a graph is described by a path on the graph that visits every edge at least once by giving the sequence of nodes to visit. We want to make the sequence as small as possible so, in other words, we want a path that visits each edge at least once of minimum length.

If the graph is *Eulerian* then there exists a representation on length $|E| + 1$ which is obviously optimal. But what can we do if the graph is not Eulerian? This is exactly the Chinese postman problem (CPP).

By the *handshaking lemma* there must be an even number of nodes with odd degree. Hence we can turn the graph into an Eulerian graph by adding paths between those nodes. Since we want to add a minimum number of edges we can find which paths to build by computing a minimum cost matching on the complete graph consisting of the odd degree nodes and having edge costs equal to the shortest path length between them. The paths can be computed using a BFS from each odd degree node.

Since the number of odd degree nodes is at most 20 we can compute the matching using dynamic programming. Notice that we do not need an Euler tour, just and Euler path. Hence we can leave two nodes unmatched.

Check the CP3 book to see how to compute the matching.