Candies (Spoiler)

1 Initial observations

First of all, we have to note that instead of altering the number of candies in one package, we can say that Kristian first discards one package and then adds a new one. We start with the observation that by adding a package with sufficiently many candies Kristian can always double the number of options he can give to a customer. Hence, we divide the solution into two stages:

1. Remove one package, such that the number of distinct orders decreases as little as possible.
2. Add a package with the smallest possible size, which doubles the number of distinct orders.

2 The first stage

We can easily fill a boolean array $orders[x]$, such that $orders[x]$ is true iff we can serve order of size x. This takes $O(BN^2)$ time with DP, where B is the limit for the size of each package. In this task however, we need a slightly smarter data structure. We define $orders[x]$ as the number of subsets of packages, which have x candies in total. The array is filled with a DP algorithm that works almost the same way. Now, we can serve an order of size x iff $orders[x] > 0$. Below is some code that shows how to compute it:

```
for(int i=0; i<N; i++)
    for(int j=N*B; j>=packages[i]; j--)
        orders[j] += orders[j-packages[i]];
```

The important property of this array, is that after we build an array for a set of packages P, we can efficiently obtain an analogous array for each $N-1$–element subset of P in $O(NB)$ time. We have to look at the array orders as on a data structure. We iterate through all the packages and add them one by one to the structure. The important property of this process is that every step can be reversed and we can remove a package number p that has already been added by 'reversing' the DP:

```
for(int j=packages[p]; j<=N*B; j++)
    orders[j] -= orders[j-packages[p]];
```

In this way, we obtain the array $orders[x]$ for each $N-1$–element subset in $O(BN^2)$ total time.

Note that $orders[x]$ can overflow even a 64-bit integer, as it can be as big as 2^{100}. To deal with that, we can make all calculations modulo a random big prime number — the chance that the algorithm would fail, ie. $orders[x] \mod P = 0$ and $orders[x] > 0$ is very low. Alternatively, we can use simple big-integer arithmetics and instead of keeping $orders[x]$, just keep $orders[x] \mod P_1$ and $orders[x] \mod P_2$, where P_1 and P_2 are big prime numbers, such that $P_1 P_2 > 2^{100}$.

It has to be noted that in theory this algorithm runs in $O(BN^3)$ time, as every element of $orders[x]$ has to be a $O(N)$-bit number. In practice, the version with 64-bit integers is relatively slow, because of the costly modulo operation.

However, there is another approach to solve the first part of the task. Again, for a given set of packages $p_1, p_2, p_3, ..., p_N$, we have to build some data structures for all its $N-1$–element subsets. We already know how to construct a data structure, which for a given set of packages says how many different orders can be served with them. Adding a package to the structure takes $O(NB)$ time. Building a structure for each $N-1$–element subset can be achieved using a recursive approach. Function *generate*, whose pseudocode is shown below, takes two arguments. The first is a list of package sizes and the second is a data structure that represents some set of packages. It generates all data structures that can be obtained by extending the structure with a $m-1$–element subset of packages.

Algorithm 1 $generate((p_1, p_2, p_3, ..., p_m), S)$

if $m = 1$ **then**
  S is one of the desired structures
else
  $S' = add(S, (p_1, p_2, ..., p_{\lfloor \frac{m}{2} \rfloor}))$
  $S'' = add(S, (p_{\lfloor \frac{m}{2} \rfloor + 1}, ..., p_m))$
  $generate((p_{\lfloor \frac{m}{2} \rfloor + 1}, ..., p_m), S')$
  $generate((p_1, p_2, ..., p_{\lfloor \frac{m}{2} \rfloor}), S'')$
end if

The *add* function adds a set of packages to the data structure given by the first argument. It runs in $O(BNm)$ time, if the size of the added set is $O(m)$. It can be shown that the *generate* function can iterate through all structures for all $m-1$–element subsets in $O(BN^2 \log N)$ total time.

3 The second stage

Now we need to find a package of smallest possible size, which added to a set of packages $p_1, p_2, p_3, ..., p_n$, doubles the number of possible orders we can serve. If we can serve orders of size $x_1, x_2, x_3, ...$, then adding a package of size T doubles the number of possible distinct orders iff T is not equal to any of $x_i - x_j$. So we need to find the smallest positive T, which cannot be expressed as $x_i - x_j$, where x_i and x_j are possible orders. To do that, it is sufficient to generate all 'orders' we can serve with packages of size $p_1, -p_1, p_2, -p_2, ..., p_n, -p_n$ and find the smallest possible integer, which cannot be obtained from packages of this size.