Demux Summer Hackathon-5
https://www.hackerrank.com/demux-summers-contest-5

1. RECURSIVE DIGIT SUM

We define *super-digit* function as following recursive function:
super-digit(n) =n,n<10

= super-digit(sum-of-digit(n)), otherwise

where,

sum-of-digit(m) =m,m=0

=(m mod 10) + sum-of-digit(m/10)

Note that since initially **n** can be very large, use string representation to represent it for the first time.

```
import Data.List
import Data.Char

main :: IO ()
main = getContents >>= print. (\[n, k] -> superDigit ((read k) * (getSumStr n))). words

superDigit n
  | n < 10 = n
  | otherwise = superDigit (getSum n)

getSum 0 = 0
getSum n = n`rem`10 + getSum (n`div`10)

getSumStr [] = 0
getSumStr (x:xs) = (ord x - ord '0') + getSumStr xs
```

C ++

```
#include <iostream>

using namespace std;

int super_digit(const string &s) {
    int rem = 0;
    for(char ch : s) {
        rem = (rem + (ch - '0')) % 9;
    }
    return rem;
}


int main() {
```

```
    ios_base::sync_with_stdio(false);

    string s;
    cin >> s;

    int k;
    cin >> k;

    k %= 9;

    if(k == 0) {
        cout << "9" << endl;
    } else {
        int remainder = super_digit(s);
        remainder = (remainder * k) % 9;
        cout << ((remainder != 0) ? to_string(remainder) : "9") << endl;
    }

    return 0;
}
```

2.Arithmetic Expressions

Let's denote the input array as @ = [a1, a2,.-., an].

Insights

With 7 elements, there are 3^{n-1} possibilities for the set of operations to use, which is too large to enumerate.

Thankfully, a few things make the task easier:

1. First, we only care about whether the final result is divisible by **101** or not. Thus, while computing the intermediate values,we can just reduce it **modulo 101** all the time, drastically shrinking the range of values we're considering. This works sincethe allowed operations (addition, subtraction, and multiplication) play well with modulo.

2. The specified left-to-right order of evaluation makes the whole thing much easier to analyze. For example, consider some big expression. It will look something like **(big expression) op number**. Now, there may be many possibilities for what big expression may be, but we can always assume that its result is in the range [0, 100] (from the previous point). Also, **op** can only be one of three operations, and **number** is predetermined by the problem. Hence, all in all, this actually only gives

us **101 *3 = 303** cases to analyze.

A brute-force solution

These two insights give us a natural way to generate an answer via brute-force:

```
def solve(expression_so_far, result_so_far, i):
```

```
if i > n:

if result_so_far == 0: # this means that the expression is divisible by 101

print expression_so_far

return True

else:

# try three operations

if solve(expression_so_far + "+" + a[i], (result_so_far + a[i]) % 101, i):

return True

if solve(expression_so_far + "-" + a[i], (result_so_far - a[i]) % 101, i):

return True

if solve(expression_so_far + "*" + a[i], (result_so_far * a[i]) % 101, i):

return True


return False
```

(Here, x % 101 is understood to only return values in the range , [unlike in some languages](#).)

To solve the problem, you should call solve(toString(a[1]), a[1], 2).

Of course, this obvious solution is also obviously slow. Consider the following test cases:

Case 1:

51

51 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1

Case 2:

51

50 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1

Case 3:

51

50 1 100

There's a unique answer in each of these cases, but the solution above (and perhaps many other types of brute-force solutions) will have trouble solving some of these.

A fast solution

Let's look back at our brute-force solution. Let's forget for a moment that we're trying to construct an expression, and suppose we're just trying to verify that there is indeed an answer, i.e., that there's a selection of operations such that the result is divisible by **101**.

As we've seen before, the left-to-right operation order makes this easier to decompose into subproblems. First, let's define **p(i,v)** to be true if it's possible to get the result **v** (modulo **101)** from the first **i** operands, and false otherwise. Thus, what we want to do is to check if **p(n,0)** is true.

In the base case, it should be easy to see that **p(1,a1)** is true, and **p(1,v)** is false if $\mathbf{v \neq a_1}$

Now, suppose we know that **p(i - 1,v)** is true. Then this implies three things:

p(i,(v + a_i) mod 101) must be true.

p(i,(v - a_i) mod 101) must be true.

p(i,(v . a_i) mod 101) must be true.

Thus, after computing **p(i-1,0),p(i-1,1),...,p(i - 1,100)**, we can now work out the values of **p(i,0),p(i,1),...,p(i,100)** using the implications above.

Here's one implementation:

```
p[1..n][0..100] # all initially false

p[1][a[1]] = true

for i from 2 to n:
    for v from 0 to 100:
        if p[i-1][v]:
            p[i][(v + a[i]) % 101] = true
            p[i][(v - a[i]) % 101] = true
            p[i][(v * a[i]) % 101] = true
```

After this computation, we can now verify that p[n][0] is indeed true, and this computation serves as the "proof".

Also, note that this "proof" is correct, i.e., p[n][0] becomes true at the end if and only if there's an answer to the problem. It should be clear why. The more surprising thing is that the computation is fast, i.e., it only takes iterations instead of . Why is this so much faster than brute force? Well, this is because we're only considering each intermediate value at most once and only if it can be formed at all, in other words, once we know that some can be formed, we don't need to know/care about how it can be formed.

Even more surprisingly, it turns out that it's possible to convert this "proof" into a concrete solution! The key is to remember how we were able to "prove" that p[n][0] is true in the first place:

For p[n][0] to have been true, there must have been some v such that p[n-1][v] is true and one of the following three things are true:

(v + a[n]) % 101 == 0

(v - a[n]) % 101 == 0

(v * a[n]) % 101 == 0.

Now, for p[n-1][v] to have been true, there must have been some w such that p[n-2][w] is true and one of the following three things are true:

(w + a[n-1]) % 101 == v

(w - a[n-1]) % 101 == v

(w * a[n-1]) % 101 == v.

Now, for p[n-2][w] to have been true, there must have been some x such that p[n-3][w] is true and one of the following three things are true:

(x + a[n-2]) % 101 == w

(x - a[n-2]) % 101 == w

(x * a[n-2]) % 101 == w.

etc.

By continuing this argument, we will end up at p[1][a[1]], and along the way, we have reconstructed the required expression!

The following is an implementation of this idea:

```
def reconstruct(i, v):
    if i != 1:
        for w from 0 to 100:
            if p[i-1][w]:
                if (w + a[i]) % 101 == v:
                    reconstruct(i-1, w)
                    print "+"
                    break
                if (w - a[i]) % 101 == v:
                    reconstruct(i-1, w)
                    print "-"
                    break
                if (w * a[i]) % 101 == v:
                    reconstruct(i-1, w)
                    print "*"
                    break
    print a[i]
```

(Here, print is understood to print only the given object and without additional whitespaces/newlines.)

Calling reconstruct(n, 0) will then print the required expression!

Code:

```
n = int(input())
a = [int(x) for x in input().strip().split()]
valid = [[False]*101 for i in range(n)]
valid[0][a[0]] = True
for i in range(1, n):
    for v in range(101):
        if valid[i-1][v]:
            valid[i][(v + a[i]) % 101] = True
            valid[i][(v - a[i]) % 101] = True
            valid[i][(v * a[i]) % 101] = True


v = 0
for i in range(n-1,0,-1):
    for w in range(101):
        if valid[i-1][w]:
            if (w + a[i]) % 101 == v:
                a[i] = '+' + str(a[i])
                v = w
                break
            if (w - a[i]) % 101 == v:
                a[i] = '-' + str(a[i])
                v = w
                break
            if (w * a[i]) % 101 == v:
                a[i] = '*' + str(a[i])
                v = w
                break
```

print(*a, sep='')101) from the first operands, and false otherwise. Thus, what we want to do is to check if is true.

- In the base case, it should be easy to see that is true, and is false if .
- Now, suppose we know that is true. Then this implies three things:
    - must be true.
    - must be true.
    - must be true.
- Thus, after computing , we can now work out the values of using the implications above.

Here's one implementation:

```
p[1..n][0..100] # all initially false


p[1][a[1]] = true


for i from 2 to n:
    for v from 0 to 100:
```

```
if p[i-1][v]:
    p[i][(v + a[i]) % 101] = true
    p[i][(v - a[i]) % 101] = true
    p[i][(v * a[i]) % 101] = true
```

After this computation, we can now verify that p[n][0] is indeed true, and this computation serves as the "proof".

Also, note that this "proof" is correct, i.e., p[n][0] becomes true at the end if and only if there's an answer to the problem. It should be clear why. The more surprising thing is that the computation is fast, i.e., it only takes **101n** iterations instead of $\mathbf{3^{n-1}}$**.** Why is this so much faster than brute force? Well, this is because we're only considering each intermediate value **v** at most once and only if it can be formed at all, in other words, once we know that some can be formed, we don't need to know/care about how it can be formed.

Even more surprisingly, it turns out that it's possible to convert this "proof" into a concrete solution! The key is to remember how we were able to "prove" that p[n][0] is true in the first place:

- For p[n][0] to have been true, there must have been some v such that p[n-1][v] is true and one of the following three things are true:
    - (v + a[n]) % 101 == 0
    - (v - a[n]) % 101 == 0
    - (v * a[n]) % 101 == 0.
- Now, for p[n-1][v] to have been true, there must have been some w such that p[n-2][w] is true and one of the following three things are true:
    - (w + a[n-1]) % 101 == v
    - (w - a[n-1]) % 101 == v
    - (w * a[n-1]) % 101 == v.
- Now, for p[n-2][w] to have been true, there must have been some x such that p[n-3][w] is true and one of the following three things are true:
    - (x + a[n-2]) % 101 == w
    - (x - a[n-2]) % 101 == w
    - (x * a[n-2]) % 101 == w.
- etc.

By continuing this argument, we will end up at p[1][a[1]], and along the way, we have reconstructed the required expression!

The following is an implementation of this idea:

```
def reconstruct(i, v):
  if i != 1:
    for w from 0 to 100:
      if p[i-1][w]:
        if (w + a[i]) % 101 == v:
          reconstruct(i-1, w)
          print "+"
          break
        if (w - a[i]) % 101 == v:
          reconstruct(i-1, w)
          print "-"
          break
        if (w * a[i]) % 101 == v:
```

```
                reconstruct(i-1, w)
                print "*"
                break
    print a[i]
```

(Here, print is understood to print only the given object and without additional whitespaces/newlines.)

Calling reconstruct(n, 0) will then print the required expression!

Here's the editorialist's implementation:

```
n = int(input())
a = [int(x) for x in input().strip().split()]
valid = [[False]*101 for i in range(n)]
valid[0][a[0]] = True
for i in range(1, n):
    for v in range(101):
        if valid[i-1][v]:
            valid[i][(v + a[i]) % 101] = True
            valid[i][(v - a[i]) % 101] = True
            valid[i][(v * a[i]) % 101] = True

v = 0
for i in range(n-1,0,-1):
    for w in range(101):
        if valid[i-1][w]:
            if (w + a[i]) % 101 == v:
                a[i] = '+' + str(a[i])
                v = w
                break
            if (w - a[i]) % 101 == v:
                a[i] = '-' + str(a[i])
                v = w
                break
            if (w * a[i]) % 101 == v:
                a[i] = '*' + str(a[i])
                v = w
                break

print(*a, sep='')
```

3. CUT THE STICKS

You are initially given the height of N sticks. At each step, find the height of the smallest stick, say k , and cut a portion of length k from each stick. All those sticks with height k will disappear. Then print the number of the remaining sticks.
Repeat the above step until all the sticks disappear.

Python :

```
N = input()
num = map(int, raw_input().split())
val = [0] * 1001
for i in num:
    val[i] += 1
counter = 0
val = val[::-1]
ans = []
for i in val:
    if i > 0:
        counter += i
        ans.append(counter)
ans = ans[::-1]
for i in ans:
    print i
```

C++:

```
#include <bits/stdc++.h>
using namespace std;

int main() {
    int N;
    cin >> N;
    vector <int> height(N);

    for(int i = 0; i < N; ++i) {
        cin >> height[i];
    }

    int mx = *max_element(height.begin(), height.end());
    while(mx > 0) {
        int mn = mx;
        int cuts = 0;

        for(int i = 0; i < (int)N; ++i) {
            if(height[i] > 0)
                mn = min(mn, height[i]);
        }
        for(int i = 0; i < (int)N; ++i) {
```

```
            if(height[i] > 0) {
                cuts++;
                height[i] -= mn;
            }
        }

        cout << cuts << "\n";
        mx = *max_element(height.begin(), height.end());
    }

    return 0;
}
```

Haskell:

```
main :: IO ()
main = getContents >>= mapM_ print. solve. map read. tail. words

solve :: [Int] -> [Int]
solve [] = []
solve xs =  length xs: (solve. filter (>mn)$ xs)
  where mn = minimum xs
```

4.MINIMUM DISTANCES

Use two nested loops to choose two indices and check for matching elements, then find the distance between the two matching elements. Finally, select the minimum of the distances and print it on a new line.

Featured Solutions

Ruby

```
#!/bin/ruby

n = gets.strip.to_i
a = gets.strip
a = a.split(' ').map(&:to_i)

last ={}
min = -1
n.times do |i|
    x = a[i]
    l = last[x]
    if l and (min < 0 or i - l < min)
        min = i - l
    end
    last[x] = i
end

puts min
```

Problem Setter's code:

Python 2

```
n = int(raw_input())
A = map(int, raw_input().split())

ans = 9999999999
for i in range(n):
    for j in range(n):
        if A[i] == A[j] and i != j:
            ans = min(ans, abs(i - j))

if ans == 9999999999:
    ans = -1
print ans
```

C++

```
#include <bits/stdc++.h>
using namespace std;

int n;
int a[1000+1];
#define inf 1000000000
int main() {
        cin>>n;
        for(int i = 0; i < n; i++) cin>>a[i];

        int res = inf;
        for(int i = 0; i < n; i++)
                for(int j = 0; j < i; j++)
                        if(a[i] == a[j]) res = min(res, i - j);

        if(res == inf) res = -1;
        cout << res << endl;

        return 0;

}
```

5. STRANGE COUNTER

Let's define a series,**S = {1,4,10,22,...}** , where each element S_i is the initial time at the beginning of a cycle. At each time **t=** S_i, the counter displays the value S_i **+ 2**. We then loop through series **S** and use it to find the value displayed by the counter:

1. If input time **t € S** , then the answer is **t + 2**
2. If S_i **< t <** S_i **+1**
   We know at time **t =** S_i , the value displayed by the counter is S_i **+ 2** . Because time **t** occurs later within that cycle, the value displayed at time **t** will be decreased by **t -** S_i . This means the value displayed by the counter at time t is S_i **+ 2 - (t -** S_i **)**.

Python :

```
time = input()
t = 1
value = 3
delta = 3
while (t + delta <= time):
    t += delta
    delta *= 2
    value *= 2
print value - time + t
```

Code:

```
t=int(raw_input())
S = 1
init = 3
ans = None

while True:
    if S == t:
        ans = S+2
        break
    elif S + init > t:
        ans = S+2-(t-S)
        break
    S = S + init
    init = init * 2

print ans
```

6. **K Factorization**

```
import java.io.*;
import java.util.*;

public class Solution {


    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
```

```
        int n = sc.nextInt();
        int k = sc.nextInt();
        int[] a = new int[k];
        for(int i = 0; i < k; i++) {
            a[i] = sc.nextInt();
        }

        LinkedList<Integer> sol = new LinkedList<Integer>();
        Arrays.sort(a);

        if(backtrack(n,a,a.length-1,sol)) {
            int curr = 1;
            for(int i = 0; i < sol.size()-1; i++) {
                curr *= sol.get(i);
                System.out.print(curr + " ");
            }
            curr *= sol.get(sol.size()-1);
            System.out.println(curr);
        } else {
            System.out.println(-1);
        }



    }

    public static boolean backtrack(int n, int[] a, int index,LinkedList<Integer> sol) {
        if(n == 1) {
            sol.add(1);
            return true;
        }

        for(int i = index; i >= 0; i--) {
            if(n % a[i] == 0) {
                if(backtrack(n/a[i],a,i,sol)) {
                    sol.add(a[i]);
                    return true;
                }
            }
        }

        return false;
    }

}
```