

Task POT	Author: Nikola Dmitrović
-----------------	---------------------------------

First we need to observe how we can determine the value of an expression $base^{pot}$. We can determine this by using a predefined function in a programming language of choice (i.e. function `pow(base, pot)` in Python) or by consecutively multiplying the number *base* with itself *pot* times. Another problem is, given *P*, to determine the value of *number* (the result of *P* modulo 10) and the value of *pot* (the result of integer division of *P* by 10).

Solution (written in Python 3.x):

```
N = int(input())

total = 0

for i in range(N):
    P = int(input())

    pot = P % 10
    base = P // 10

    power = 1
    for i in range(pot):
        power *= baza

    total += power

print(total)
```

Necessary skills: decision statements, repeat statements, algorithm for determining digits

Category: ad-hoc

Task ESEJ	Author: Adrian Satja Kurdija
------------------	-------------------------------------

In this task, we needed to find a large enough number of distinct words. How can we, for instance, generate 50 000 different words? Fortunately, there are many ways to do this.

One solution is to notice that it is very easy to output a sufficient number of different *numbers*. For example, we can output them sequentially: 1, 2, 3, ..., and it is possible to transform these numbers to *words* by, for example, transforming digits to letters using the key 0 -> 'a', 1 -> 'b', 2 -> 'c', and so on. For instance, the number 451 is transformed to the word "efb". If we use this way to transform every integer from 1 to **B**, all given words are distinct because the initial numbers are also distinct.

If we hadn't thought of this solution, we could have output **B** words by randomly choosing letters per word (using a function such as *rand()* in C/C++ or the module *random* in Python). This approach will be valid because all given words will be distinct with a huge probability.

Necessary skills: strings

Category: ad-hoc

Task MOLEKULE	Author: Mislav Balunović
----------------------	---------------------------------

The key observation in this task is to notice that we can always construct the solution so that the longest path an impulse has to travel is 1. Since the graph of molecules is a tree, we can color the molecules in two colors (i.e. black and white) using the dfs algorithm so that there are no two molecules of the same color that are connected with a covalent bond. Now we must direct each covalent bond so that it points from the white molecule to the black one. It is clear that a path longer than 1 does not exist in the newly created graph.

Notice that the claim can be generalized to any bipartite graph.

Necessary skills: dfs

Category: graph theory

Task SLON	Author: Dominik Gleich
------------------	-------------------------------

The condition from the task is that the expression is a first degree polynomial of the form $\mathbf{ax} + \mathbf{b}$. Let $f(\mathbf{x}) = \mathbf{ax} + \mathbf{b}$. If we calculate the expression in point 0, we will see that $f(0) = \mathbf{b}$. If we calculate it in point 1, we will see that $f(1) = \mathbf{a} + \mathbf{b}$. This gives us values \mathbf{a} and \mathbf{b} . We are left to figure out how to calculate the minimal value of \mathbf{x} such that $\mathbf{ax} + \mathbf{b} = \mathbf{p} \pmod{\mathbf{m}}$. Given that $\mathbf{m} \leq 10^6$, we can iterate over all values of \mathbf{x} until we find the minimal value that satisfies the upper equation. We are left to figure out how to calculate the expression in point 0 and point 1. One of the ways we can do this is using the command 'eval' in Python. Another way is to parse the expression from infix to postfix notation using the "shunting-yard" algorithm.

The official solution implements another approach where the expression is converted to postfix notation and then the addition, multiplication and subtraction is performed over polynomials with maximal degree of 1 and, finally, the same equation of the form $\mathbf{ax} + \mathbf{b} = \mathbf{p} \pmod{\mathbf{m}}$ is solved.

Necessary skills: stack, queue

Category: ad-hoc, postfix, infix

Task NEKAMELEONI	Author: Mislav Balunović
-------------------------	---------------------------------

We will describe a “divide and conquer” algorithm that, given an array **A** of length **N**, finds the shortest subsequence that contains all numbers from 1 to **K**.

```
solve(T) :  
    L = left half of array T  
    R = right half of array T  
    X = solve(L)  
    Y = solve(R)  
    Z = the shortest subsequence that contains all numbers from 1 to  
    K, and is comprised of the suffix of array L and the prefix of array  
    R  
    return min(X, Y, Z)
```

If we can calculate the value **Z** in the complexity $O(\text{length of array } \mathbf{T})$, then the algorithm’s complexity is $O(\mathbf{N} \lg \mathbf{N})$.

How to determine the value **Z**? For each suffix of the array **L**, we will calculate a bitmask that describes a set of numbers that is in that suffix ($\mathbf{K} \leq 50$ so that bitmask will be a single long long). We will do the same thing for each prefix of array **R**. Now we want to find two bitmasks whose binary or is equal to $2^{\mathbf{K}} - 1$, and the sum of the lengths of the corresponding suffix and prefix is minimal. Notice that there are at most **K** interesting bitmasks from the left and from the right side, so we can do this in $O(\mathbf{K}^2)$ by trying out all pairs of bitmasks. We can use the “monotonous” property of the bitmasks so we can achieve the same result in $O(\mathbf{K})$.

Nevertheless, this is not quick enough because we have **M** queries that need to be answered.

We will construct a tournament tree where each node stores “interesting” sets of bitmasks for prefixes and suffixes of its interval and the shortest subsequence that contains all numbers from 1 to **K** in its interval. Then the merging of nodes in the tree actually corresponds to the step of the aforementioned algorithm. When we modify a number in the array, we need to recalculate the values in $O(\lg \mathbf{N})$ nodes, so the complexity of a single change is $O(\mathbf{K} \lg \mathbf{N})$, and the total complexity is $O(\mathbf{KN} \lg \mathbf{N} + \mathbf{KM} \lg \mathbf{N})$.

For implementation details, consult the official source code.

Necessary skills: divide and conquer algorithms, tournament tree

Category: data structures

Task DOMINO	Authors: Adrian Satja Kurdija, Mislav Balunović
--------------------	--

It is clear that minimizing the sum of visible fields is equivalent to maximizing the sum of covered fields. A greedy algorithm ("take the largest domino, delete it, repeat") already fails on the second sample test. Nevertheless, in order to solve this problem, it is sufficient to only observe a certain number of dominoes that are the best or, in other words, largest considering the sum of their fields.

What exactly is that number of dominoes? Let us notice that each domino overlaps with at most 6 other dominoes. That means that after taking $K-1$ dominoes, we've "crossed out" $7(K-1)$ dominoes in total, so for the last domino it clearly pays off to take one of the best $7(K-1) + 1$ dominoes. Since the order of choosing dominoes is not important, each chosen domino can be the last one, which means that each domino from the optimal choice is one of the best $7(K-1) + 1$ dominoes.

For $K \leq 5$ that number is less than thirty, so trying out every possible combination (from that set of the best dominoes) is quick enough. For $K \leq 8$, the number of dominoes to observe reaches 50. Since the number of all possible choices is too large in this case, we will use the *meet in the middle* approach: we will split the observed set into two parts and, for each of them, observe the possible choices inside it.

Let us first construct a graph where the nodes are dominoes, and two dominoes are connected with an edge if they don't overlap or, in other words, if they can be taken at the same time. Therefore, we need to find a *clique* (a complete subgraph, i.e. a set of nodes where each two are connected) of size K where the sum of all the nodes (the dominoes' values) is maximal. Each subgraph is observed as a bitmask, a binary number where 1s represent the chosen nodes.

Let us split the graph into two parts so that the first one contains A nodes and the other B nodes (sizes A and B will be determined later). In all possible ways, let's assume that we will choose D dominoes of the required clique from the first part and $K - D$ dominoes from the second part of the graph.

For a fixed D we do the following:

- In the first part of the graph, for each subgraph *mask* we calculate $dp[mask]$ that gives us the maximal sum of nodes in its subclique of size D .
 - How can we do this? If the subgraph is of size D , we check whether it's a clique, and if it is, we sum up the nodes. If the subgraph consists of more than D nodes, then

$$dp[mask] = \max\{ dp[submask] \}$$

for all subgraphs *submask* obtained by removing one node (binary one) from the subgraph *mask*.

- In the second part of the graph, for each clique of size $K - D$, we search for all the nodes in the first part that are connected to all the nodes from the chosen clique -- these nodes from the first part comprise a subgraph *mask* and we want to know its best subclique of size D , which is stored in $dp[mask]$.
- To summarize: for this D , we maximize $sum(clique) + dp[mask]$ for all cliques in the second part of size $K - D$ and its corresponding "friendly" subgraphs *mask* from the first part.

The final solution is, naturally, the maximum for all $D = 0, 1, \dots, K$. Finally, how big should the parts A and B be? Since in the first part we are observing all subgraphs, and in the second only those of size $K - D$, the first one should be smaller: in the official solution, $A = 20$, $B = 30$. The analysis of this algorithm's complexity is left as an exercise to the reader.

Necessary skills: meet in the middle, dynamic programming, graphs

Category: graph theory