Belgian Olympiad in Informatics

Carnival Training — Day 2

Dynamic Programming and Greedy Algorithms

Trainer:
Victor Lecomte

March 6, 2014

Contents

1 Dynamic programming

There are two main ways to increase the efficiency of an algorithm: "don't do anything twice" and "don't do anything stupid". While the definition of stupid is not obvious and "not being stupid" is quite difficult, "don't do anything twice" is quite clear and fixing your algorithm with it is quite easy. That's what DP is about.

1.1 Memorization

What makes DP special is that it can take some complexities from exponential time ($O(k^n)$) to polynomial time ($O(n^k)$). And it does that through memorization.

Sometimes it's obvious when thinking about the naive approach what you should memorize to speed it up. Usually it will take the shape of an array (possibly 2- or 3-dimensional) that you will gradually fill up when finding solutions to sub-problems, and then re-use it to find the solutions of the larger problems.

1.2 Fibonacci sequence

This is a simple example of how memorization can dramatically speed up calculations. We want to find the n-th Fibonacci number. If we just implement the definition of the Fibonacci sequence, we get:

```
int fibo(int n)
{
    if(n==0)
        return 0;
    else if(n==1)
        return 1;
    else
        return fibo(n-1) + fibo(n-2);
}
```

However, since `fibo(n)` calls both `fibo(n-1)` and `fibo(n-2)`, the running time of that function will about double every time n increases by 1. So the complexity is $O(2^n)$, which is quite crappy.

(Actually, it's more like $O(\phi^n)$, but it doesn't affect the crappiness.)

And that's because we compute the same thing *many* times. To avoid that, we just have to remember the solutions to the sub-problems, that is, the previous Fibonacci numbers, in an array. Then we can just fill up the array progressively, which is easier than putting the results in throughout the algorithm as we find them.

```
int fibo(int n)
{
    vector<int> f;
    f.push_back(0);
    f.push_back(1);
    for(int i=2; i<=n; i++)
        f.push_back(f[i-1] + f[i-2]);
    return f[n];
}
```

1.3 Longest increasing subsequence

But finding the sub-problems to solve is not always as easy, as the following example shows.

In the longest increasing subsequence problem, you have to find the longest increasing sequence of elements in an array of length n, not necessarily consecutive. For example, in the list $[4, 7, 5, 1, 3, 2, 6, 8]$, longest increasing subsequences would be $[4, 5, 6, 8]$ or $[1, 2, 6, 8]$. (The solution is not unique.)

Trying all subsequences and finding the longest increasing one would take $O(2^n)$ time, as every number in the array may or may not be in the subsequence.

A first intuition might be to start from the beginning and keep the longest increasing subsequence so far in memory, and add to it as we go. But the problem is that it might not find the best solution. In $[4, 7, 5, 1, 3, 2, 6, 8]$, for example, it will add 4, then 7, but then it will not add any element until 8, giving $[4, 7, 8]$, which is not optimal.

Actually, when we want to add some element $a[i]$ of the array a, instead of trying to add it to the longest increasing subsequence between 0 and $i - 1$, we should try to find the longest increasing subsequence that ends with an element smaller than $a[i]$. So we could keep a table of the longest increasing subsequences ending with all possible values. The implementation would look like that:

```
int lis(vector<int> a)
```

```
{
    int n = a.size()
    vector<int> ending_with;
    for(int i=0; i<n; i++)
        ending_with.push_back(0);

    for(int i=0; i<n; i++)
        for(int j=0; j < a[i]; j++)
            ending_with[a[i]] = max(ending_with[a[i]],
                                    ending_with[j]+1);
}
```

Beware: this assumes that the max value in array a is smaller than n (we can remap the values if it is not the case). The time complexity is $O(n^2)$, much better than $O(2^n)$, but we can still do better (and more convenient).

Instead of looking at the longest subsequence for some value, let us do the opposite and look at the smallest ending value for which there is a subsequence of some length. We would then have to find the biggest length such that the ending value is smaller than the current element.

Let's name this array `smallest_end_for[]`. It will contain the *index* of the smallest ending value. If you can achieve some length with an element, then you can achieve any shorter length, so `smallest_end_for` is non-decreasing. So you can just use binary search to find the longest subsequence there is ending with an element smaller than. Let's say its length is `prev_len`.

Then you just have to update `smallest_end_for[prev_len+1]` if the solution you've just found is better than the previous one. You don't have to update anything else, since the lengths 1 to `prev_len` can already end with smaller values.

In conclusion, here is an implementation:

```
vector<int> lis(vector<int> a)
{
    int L=0; // Longest so far
    vector<int> smallest_end_for;
    for(int i=0; i<a.size(); i++)
    {
        // Binary search for the best length before a[i]
        int lower=0, upper=L+1;
        while(lower+1 < upper)
        {
            int middle = (lower+upper)/2;
            if(a[smallest_end_for[middle]] < a[i])
                lower = middle;
            else
                upper = middle;
        }
        int prev_len = lower;
        // If the length is the best so far
        if(prev_len+1 > L)
        {
            smallest_end_for.push_back(i);
            L++;
        }
        // If this value is the new smallest for this length
        else if(a[i] < a[smallest_end_for[prev_len+1]])
        {
            smallest_end_for[prev_len+1] = i;
        }
    }
}
```

Since the outer loop is run n times and the inner loop is run at most $\log_2 n$ times, the total complexity is $O(n \log n)$.

Note: both algorithms can be adapted to output the generated subsequence as well, not just its length.

2 Greedy algorithms

The idea behind greedy algorithms is to always take the decision that sounds the best, and damn the consequences. It will always look for immediate profit. To put it another way, we build up large solutions from smaller ones. For example, assuming we have found the best solution for $N = 4$, we just build the solution for $N = 5$ from it, without considering any other solution.

The good thing is greedy algorithms are usually fast. The bast thing is they don't often work, and when they do, it's difficult to tell.

2.1 Box stacking

See complete statement in the notes for day 1.

We have to put n boxes in as few piles a possible with a constraint: some boxes have a maximum number of boxes that can be put on top of them.

The greedy algorithm that was proposed by a few students went like this: first we sort the boxes in increasing strength, then we always add them under existing stacks. If all piles are full, we create a new stack.

At first sight, it's not obvious *why*, but it seems to work on a few examples.

In practice that will be enough to begin implementing. Since this is no math olympiad, we only need to be *fairly sure* that the algorithm works in all cases, depending on the time it will take to implement.

To prove the correctness, we notice two things:

- There is always an optimal solution with increasing strength in all piles from top to bottom. If it is not the case, just swap boxes until it is. So we can reach an optimal solution by placing the boxes in order.
- When we begin placing the boxes of strength s with this algorithm, only two things matter: the number of existing piles and the number of boxes

already placed. If some lower-strength box is placed in one pile instead of another, it will not change the number of places the s-boxes can fill.

But both the number of existing piles and the number of boxes placed will not depend on the way the smaller boxes are placed. So all partial solutions with the best number of piles are equivalent, and since the beginning of the complete optimal solution is one of them, then we can be sure this algorithm will achieve an optimal solution as well.

2.2 Change-making problem

In some far away land, there are n types of coins, with values c_i. How can you add them up to an amount of money m using the least coins possible? You can assume one of the coins is 1.

The intuitive method for that problem is to take the biggest coin not bigger than the amount, then the biggest coin not bigger than the rest, etc. After all, it works quite well in real life.

But that approach is not always correct. For example, with coins $\{1, 5, 8, 10\}$ and $m = 13$, this algorithm would find $10 + 1 + 1 + 1$ while the best solution is $5 + 8$. It actually depends on the coin system that is used, and ours works fine ($\{1, 2, 5, 10, \ldots\}$).

Actually, since we can find the solution from a few of the solutions with lower m, we can use DP to compute the minimum number of coins to make all amounts from 1 to m. The complexity is $O(nm)$:

```
int best(vector<int> c, int m)
{
    vector<int> num_coins;
    for(int i=0; i<=m; i++)
    {
        num_coins.push_back(i);
        for(int j=0; j<c.size(); j++)
            if(i >= c[j])
                num_coins[i] = min(num_coins[i],
                                   num_coins[i-c[j]]+1);
    }
}
```

Again, the algorithm can be slightly tweaked to return the actual set of coins that is needed.

3 Problem set

3.1 Mixtures

Harry Potter has n mixtures in front of him, arranged in a row. Each mixture has one of 100 different colors (colors have numbers from 0 to 99).

He wants to mix all these mixtures together. At each step, he is going to take two mixtures that stand next to each other and mix them together, and put the resulting mixture in their place.

When mixing two mixtures of colors a and b, the resulting mixture will have the color $(a + b) \% 100$.

Also, there will be some smoke in the process. The amount of smoke generated when mixing two mixtures of colors a and b is $a \times b$.

Find out what is the minimum amount of smoke that Harry can get when mixing all the ixtures together.

Input

There will be a number of test cases in the input.

The first line of each test case will contain n, the number of mixtures, $1 \leq n \leq 100$.

The second line will contain n integers between 0 and 99 — the initial colors of the mixtures.

Output

For each test case, output the minimum amount of smoke.

Limits

- Time limit: 9 sec
- Memory limit: 256 MB

Example

```
Input:
2
18 19
3
40 60 20
Output:
342
2400
```

Notes

In the second test case, there are two possibilities:

- first mix 40 and 60 (smoke: 2400), getting 0, then mix 0 and 20 (smoke: 0); total amount of smoke is 2400
- first mix 60 and 20 (smoke: 1200), getting 80, then mix 40 and 80 (smoke: 3200); total amount of smoke is 4400

The first scenario is the correct approach since it minimizes the amount of smoke produced.

Source

`http://www.codechef.com/problem/MIXTURES`

3.2 Fox and Number Game

Fox Ciel is playing a game with numbers now.

Ciel has n positive integers: $x_1, x_2, \ldots, x_n$. She can do the following operation as many times as needed: select two different indexes i and j such that $x_i > x_j$ holds, and then apply assignment $x_i = x_i - x_j$. The goal is to make the sum of all numbers as small as possible.

Please help Ciel find this minimal sum.

Input

The first line contains an integer n ($2 \leq n \leq 100$). Then the second line contains n integers: $x_1, x_2, \ldots, x_n$ ($1 \leq x_i \leq 100$).

Output

Output a single integer — the required minimal sum.

Limits

- Time limit: 1 sec
- Memory limit: 256 MB

Examples

```
Input:
2
1 2
Output:
2

Input:
3
2 4 6
Output:
6

Input:
2
12 18
Output:
```

```
12

Input:
5
45 12 27 30 18
Output:
15
```

Notes

In the first example the optimal way is to do the assignment $x_2 = x_2 - x_1$.

In the second example the optimal sequence of operations is: $x_3 = x_3 - x_2$, $x_2 = x_2 - x_1$.

Source

`http://codeforces.com/problemset/problem/389/A`

3.3 George and Number

George is a cat, so he really likes to play. Most of all he likes to play with his array of positive integers b. During the game, George modifies the array by using special changes. Let's mark George's current array as $b_1, b_2, \ldots, b_{|b|}$ (record $|b|$ denotes the current length of the array). Then one change is a sequence of actions:

- Choose two distinct indexes i and j ($1 \leq i, j \leq |b|,\ i \neq j$), such that $b_i \geq b_j$.
- Get number $v = \text{concat}(b_i, b_j)$, where $\text{concat}(x, y)$ is a number obtained by adding number y to the end of the decimal record of number x. For example, $\text{concat}(500, 10) = 50010$, $\text{concat}(2, 2) = 22$.
- Add number v to the end of the array. The length of the array will increase by one.
- Remove from the array numbers with indexes i and j. The length of the array will decrease by two, and elements of the array will become re-numbered from 1 to current length of the array.

George played for a long time with his array b and received from array b an array consisting of exactly one number p. Now George wants to know: what is the maximum number of elements array b could contain originally? Help him find this number. Note that originally the array could contain only *positive* integers.

Input

The first line of the input contains a single integer p ($1 \leq p < 10^{100000}$). It is guaranteed that number p doesn't contain any leading zeroes.

Output

Print an integer — the maximum number of elements array b could contain originally.

Limits

- Time limit: 1 sec
- Memory limit: 256 MB

Examples

```
Input:
9555
Output:
4

Input:
10000000005
Output:
2

Input:
800101
Output:
3

Input:
45
Output:
1

Input:
1000000000000001223300003342220044555
Output:
17

Input:
19992000
Output:
1

Input:
310200
Output:
2
```

Notes

Let's consider the test examples:

- Originally array b can be equal to $\{5, 9, 5, 5\}$. The sequence of George's changes could have been: $\{5, 9, 5, 5\} \to \{5, 5, 95\} \to \{95, 55\} \to \{9555\}$.
- Originally array b could be equal to $\{1000000000, 5\}$. Please note that the array b cannot contain zeros.
- Originally array b could be equal to $\{800, 10, 1\}$.
- Originally array b could be equal to $\{45\}$. It cannot be equal to $\{4, 5\}$, because George can get only array $\{45\}$ from this array in one operation.

Note that the numbers can be very large.

Source

`http://codeforces.com/problemset/problem/387/C`

3.4 Bytelandian gold coins

In Byteland they have a very strange monetary system.

Each Bytelandian gold coin has an integer number written on it. A coin n can be exchanged in a bank into three coins: $n/2$, $n/3$ and $n/4$. But these numbers are all rounded down (the banks have to make a profit).

You can also sell Bytelandian coins for American dollars. The exchange rate is 1 : 1. But you can not buy Bytelandian coins.

You have one gold coin. What is the maximum amount of American dollars you can get for it?

Input

The input will contain several test cases (not more than 10). Each testcase is a single line with a number n, $0 \le n \le 1\,000\,000\,000$. It is the number written on your coin.

Output

For each test case output a single line, containing the maximum amount of American dollars you can make.

Limits

- Time limit: 9 sec
- Memory limit: 256 MB

Example

```
Input:
12
2
Output:
13
2
```

Notes

You can change 12 into 6, 4 and 3, and then change these into \$6+\$4+\$3 = \$13.

If you try changing the coin 2 into 3 smaller coins, you will get 1, 0 and 0, and later you can get no more than \$1 out of them. It is better just to change the 2 coin directly into \$2.

Source

`http://www.codechef.com/problems/COINS`