Lection 1: Dynamic Programming

09.03.2019

Filipp Rukhovich

Hello Muscat International Bootcamp 2019

CP: common situation

The main goal of our Workshop is preparing for different stages of International Collegiate Programming Contest (ICPC) and other similar programming competitions.

Before discussing the main topic of our lection, we'll discuss some common things about competitive programming, or CP.

During the CP event, some number of problems is given to you; to solve the problem, you are to write a program which usually reads a data from *stdin*, standard input stream ("from screen"), calculate the result and print it into *stdout*, standard output stream ("to screen" too).

CP: testing system

When the program for some problem is ready (as you think), you should submit a source of your program in automatic testing system (TS). This system runs the submitted program on some number of **secret** *tests*.

Each test contains a sample of what can be done in input, and correct answer for this input. Input data are written in strict format and strictly satisfy constraints; these format and constraints described in problem statement (**Input** section). During each test, TS runs the submitted program and emulate keyboard input using input data described above, and then, checks the answer which the program printed in *stdout*.

Attention!

The answer should be printed in strict format which is described in problem statement (**Output** section). For example, if the problem is to find a print a sum of two numbers, then the answer can be accepted only if you print **only the answer**, without any phrases like “Enter numbers a, b”!!! Don’t print these phrases unless it’s a part of output format!

CP: testing system

Test is considered as passed by the solution if on this test the solution's work lasts no more seconds then written in statement as time limit, uses no more memory then written as memory limit, and the information printed by the program is correct and satisfies format conditions.

For each problem, jury prepared some sequence of tests (usually from 10 to 100). These tests are unknown to participants during the contest, except (in most cases) first one or two tests which are equal to sample tests given in problem statement. Having got a submitted solution, testing system runs the program on prepared tests one-by-one until achieving the end or first unpassed test. Sequence of tests is fixed and does not depend on submit, team, language of solution etc.

CP: ICPC rules

In ICPC-style contest, the solution is accepted by the testing system if and only if the solution passes **all the tests**; the problem is solved by the team if and only if team submitted the solution which was accepted by testing system.

Participants are ranked by two parameters:

1. number of problems being solved (more is better);
2. in case of a tie, total penalty time for all problems. If the problem is not solved by the team, then team does not gets any penalty (independently of number of submits) for this problem; but if the problem was solved, then the penalty for it is a number of minutes between start of contest and time of submit which was accepted plus 20 minutes for each incorrect submit (only for this problem, of course). Less time is better.

CP: time limit

One of the things which makes CP interesting is time limit. Most CP problems has simple brute-force, but slow solution. For example, consider the following algorithm:

```
1: $ans := 0$;
2: for $i = 1, 2, \dots, n$ do
3:     for $j = 1, 2, \dots, n$ do
4:         $ans = ans + j\%i$;
```

Time of work of this algorithm depends on n. For example, if $n = 10000$, then it works in 546 ms; for $n = 30000$ — in 4.9 seconds, for $n = 40000$ — in 8.7 s, for $n = 50000$ — more than 10 s.

One of the most popular time limits in CP - 1 second. So, our algorithm could be accepted only in assumption $n \leqslant 10^4$. In our case, it was easy to implement and test the algorithm; but in common case, we would like to have a way to estimate a speed of algorithm before implementing. How could we do it?

CP: time limit and asymptotic analysis

Obvious idea is to estimate the number of actions and divide it by some experimental constant *CAP* - "how many actions could be done in a second". But there are fundamental problems:

1. it's very difficult to say which concrete operations are really made by computer;
2. different basic operations consume different time — for example, operation % works a couple of times longer than operation +;
3. different compilers/interpretators work with different speed — for example, C/C++ is the fastest programming language, but Python is really slower.

By the way, because of its low speed, for almost any problem you can see on ICPC contests, **there are no guarantees that there are correct and fast enough solution on Python** — such guarantees can be given only for C/C++ and Java languages.

CP: time limit and asymptotic analysis

To abstract from these problems, one can observe that, for example, in our algorithm, there exists some constant $C > 0$, such that for any n, number of actions performed by the program, is not more than Cn^2. In this case, we will say that the complexity of our algorithm is $O(n^2)$. We abstract from concrete C because of many factors concrete value C depends on.

After that, take maximum possible n from problem statement and substitute it in function in $O(\dots)$ (assume that C is equal to 1). It gives us some number — approximate maximum number of operations our algorithm would perform; let it be t_{max}. Then, compare this number with "psychological" constant $CAP = 10^8$ (value is given from practice):

CP: time limit and asymptotic analysis

- If t_{max} is "much less" than 10^8 then the solution will works in 1-2 seconds "almost surely", and you can start implementing;
- If t_{max} is "much more" than 10^8 then the solution will NOT works in 1-2 seconds "almost surely", and you would better try to build a new faster algorithm;
- If t_{max} has the same order as 10^8 then the success will depend on the hidden constant C; estimating of this constant depends on implementation and can be estimating using personal experience.

The "magic number" 10^8 is an approximate number; this number works good for C/C++ and Java, but badly works for Python. **We strongly recommend you avoid using Python during ICPC contests and implement only on C++ and Java languages.**

DP: what is it?

Dynamic programming(DP) is some technique which helps solve some combinatorial, optimization etc. problems.

The idea of technique is to split problem set of smaller subproblems similar to the initial one; then, solve all these problems in some order in such a way that each next subproblem can be solved using solutions for previous problems.

DP is very close to method of mathematical induction because in many cases, solution of each subproblem is simple formula which uses solutions for previous problems.

How does it work on practice?

DP: nails problem

For example, consider the following problem about nails:

- on very-very long and narrow board, $n > 1$ nails are hammered with coordinates $x[1] < x[2] < \ldots < x[n]$;
- any pair of nails l, r can be connected with string (without letters, of course) of length $|x[r] - x[l]|$;
- we are to connect some pairs of nails in such a way that each of nails should be connected with at least one other nail.
- What minimal total length of strings is necessary for it?

DP: nails problem

How to solve this problem?

Notice that we have a sense to connect each nail only with its neighbours (because, for example, instead of connecting nails 2 and 5, we can connect 2 and 3, 3 and 4, 4 and 5 with the same total length of strings);

It means that nails $n-1$ and n should be definitely connected.

But the question is: should we connect nails $n-1$ and $n-2$?

DP: nails problem

The answer depends on coordinates of nails; but we can notice that:

- if we decide not to connect nails $n-2$ and $n-1$ then first $n-2$ nail will be connected “independently” of nails n and $n-1$;
- if we decide to connect nails $n-2$ and $n-1$ then first $n-1$ nail will be connected “independently” of nail n.

So, we found that to connect n nails optimally we should:

- connect nails n and $n-1$;
- connect optimally first $n-1$ or $n-2$ nails, independently of n.

DP: nails problem

So, we reduce the problem of connecting n nails which we cannot solve to the problems of connection of $n-1$ or $n-2$ nails which we cannot solve too.

On the other hand, if $n = 2$ or $n = 3$ then we can solve the problem simply - the answer is that $x[n] - x[1]$!

It brings us to the following recursive solution:

```
int x[1..n];

int f(int n) {
    if (n <= 3)
        return x[n] - x[1];
    else
        return x[n] + x[n-1] + std::min(f(n-1), f(n-2));
}
```

DP: nails problem

It's easy to see that this solution works; but how fast is it?

Let $T(n)$ be time of work of the algorithm for n nails.

We can write $T(2) = O(1)$, $T(3) = O(1)$, and for $n \geqslant 4$:
$T(n) = T(n-1) + T(n-2) + O(1)$.

The solution is: $T(n) = O(\phi^n)$, $\phi = \frac{1+\sqrt{5}}{2}$ is maximal root of equation $\lambda^2 = \lambda + 1$. It is fast enough only for $n \geqslant 20, 25$, may be 30.

Can we optimize the algorithm to make it fast, for example, $forn = 1000$?

DP: nails problem and common technology

The problem of our solution is that some of our subproblems are solved by our algorithm for many times. For example, $f(n-2)$ is called 2 times (from $f(n)$ and $f(n-1)$), $f(n-3)$ — 3 times (from $f(n-1)$ and from $f(n-2)$ twice etc.).

To avoid duplication, apply the following **common technology**:

1. Define all(!) subproblems we want to solve;
2. Define the formula which uses answers for subproblems and gives the answer to the initial problem;
3. Define which formulas we will use to calculate answer for subproblems using answers for other subproblems; particularly, find such subproblems that can be solved “offline”;
4. Define order of solving problems;
5. Calculate answer for defined subproblems using defined formulas; after that, calculate the answer.

DP: nails problem and common technology

Apply such a technology to our nails problem:

1. we will calculate function $dp[i]$, $2 \leqslant i \leqslant n$, for $dp[i]$ as minimal possible total length of strings we need to connect first i nails without any other nails;
2. the answer for the problem is $dp[n]$;
3. As we saw earlier, the following is true:
   - $dp[i] = x[i] - x[1]$, $i = 2, 3$;
   - $dp[i] = x[i] - x[i-1] + min(dp[i-1], dp[i-2])$, $i \geqslant 4$.
4. subproblems can be solved in order of increasing i.

Last part of technology includes writing a program (following slide):

DP: nails problem and common technology

```
int x[1..n], d[1..n];

int f(int n) {
    d[2] = x[2] - x[1];
    d[3] = x[3] - x[1];
    for (int i = 4; i <= n; ++i)
        d[i] = x[i] - x[i-1] + std::min(d[i-1], d[i-2]);
}
```

So, we have made a more deep analysis, avoided duplication of solving equal problems and got a $O(n)$ solution!

As a bonus, we have avoided recursion!

DP: avoid-two-neighbouring-ones problem

Consider other problem. Given positive integer n; how many sequences of n zeroes and ones such that no any two ones occur in neighbouring positions?

For example, sequences 10010, 00001, 01010101 are "good" for our problem, but "000110", "1111", "1110101" are not.

If $n = 1$ or $n = 2$ then we can solve the problem easily; consider some more "non-trivial" n. Let $S = S[1]S[2]\dots S[n]$ is some good sequence. There are two possible cases:

- $S[n] = 0$; then, $S[1]S[2]\dots S[n-1]$ can be arbitrary "good" sequence of length $n-1$;
- $S[n] = 1$; two ones can not be neighbouring, so $S[n-1]$ should be 0; then, $S[1]S[2]\dots S[n-2]$ can be arbitrary "good" sequence.

DP: avoid-two-neighbouring-ones problem

It brings us to the following solution:

1. Let $dp[i]$, $i = 1, 2, \ldots, n$, is number of “good” sequences of length n.
2. the answer for the problem is $dp[n]$;
3. as it is implied by previous reasonings, $dp[i] = dp[i-1] + dp[i-2]$, $i \geqslant 3$. One can easily see that $dp[1] = 2$, $dp[2] = 3$;
4. subproblems can be solved in order of increasing i.

Writing a program is left to the reader. The complexity of our solution is $O(n)$ too.

DP: LIS problem

The following classic optimization problem will be a bit harder:

- Given a sequence $A = (A[1], A[2], \ldots, A[n])$ of numbers. You are to find the longest increasing subsequence, or LIS, of A, i.e. such indexes $1 \leqslant i_1 < i_2 < \ldots < i_k \leqslant n$ for as big k as possible that $A[i_1] < A[i_2] < \ldots < A[i_k]$.

Firstly, learn how to find length of LIS of A.

So, let $dp[i]$, $1 \leqslant i \leqslant n$, is a length of maximal possible increasing subsequence of A, such that its last element is $A[i]$, i-th element of A.

Consider such optimal sequence for some i. Notice that such a sequence consist of only one element $A[i]$ or of optimal subsequence for some $j < i$, such that $A[j] < A[i]$, and also element $A[i]$.

DP: LIS problem

Then, $d[i] = \max(1, \max\{1 + d[j] \mid j < i, A[j] < A[i]\})$; it can be easily proved by induction.

All values of such a function $d[i]$ can be calculated in $O(n^2)$ time; after that, length of subsequence can be calculated in $O(n)$ as $\max\{d[i] \mid 1 \leqslant i \leqslant n\}$.

But how to find the LIS itself, not only its length?

Let $pr[i]$, $1 \leqslant i \leqslant n$, be:

- $argmax\{1 + d[j] \mid j < i, A[j] < A[i]\}$ if there exist at least one suitable j;
- 0, otherwise.

Then, for any i we can easy restore the longest increasing subsequence with end in $A[i]$ as the one which is set by reversed sequence of indices $i, pr[i], pr[pr[i]], \ldots$ of length $d[i]$.

DP: LIS problem

```
int n;
int a[1..n], d[1..n], pr[1..n];

void calculateLISDynamics() {
    for (int i = 1; i <= n; ++i) {
        d[i] = 1;
        pr[i] = 0;
        for (int j = 1; j < i; ++j)
            if (a[j] < a[i] && d[i] < d[j] + 1) {
                d[i] = d[j] + 1;
                pr[i] = j;
            }
    }
}

std::vector<int> restoreLIS(int lastIndex) {
    std::vector<int> lis;
    do {
        lis.push_back(lastIndex);
        lastIndex = pr[lastIndex];
    }
    while (lastIndex > 0);

    std::reverse(lis.begin(), lis.end());
    return lis;
}

std::vector<int> findLIS() {
    calculateLISDynamics();
    return restoreLIS(std::max_element(d + 1, d + n + 1) - d);
```

DP: LCS problem

Next problem is the longest common subsequence problem, or LCS:

- given two sequences $A = (A[1], A[2], \ldots, A[n])$ and $B = (B[1], B[2], \ldots, B[m])$, we are to find such two sequences $1 \leqslant i_1 < i_2 < \ldots < i_k \leqslant n$, $1 \leqslant j_1 < j_2 < \ldots < j_k \leqslant m$ for maximal possible k, such that $A[i_1] = B[j_1]$, $A[i_2] = B[j_2]$, $\ldots$, $A[i_k] = B[j_k]$.

In this problem, subproblems will be $dp[i][j]$, $0 \leqslant i \leqslant n$, $0 \leqslant j \leqslant m$, for $dp[i][j]$ as length of the longest common subsequence of sequences $A[1..i] = (A[1], A[2], \ldots, A[i])$, $B[1..j] = (B[1], B[2], \ldots, B[j])$.

It's obvious that $dp[i][j] = 0$ if $i = 0$ or $j = 0$. Otherwise, there exist two possible cases:

DP: LCS problem

- $A[i] = B[j]$; then, it makes sense to take $A[i] = B[j]$ as last element of common subsequence; then, $dp[i][j] = 1 + dp[i-1][j-1]$;
- $A[i] \neq B[j]$; then, for and optimal $LCS(A[1..i], B[1..j])$, $A[i]$ does not lie into tis LCS or $B[j]$ does not lie into this LCS (these two statements can be true at the same time, but at least one must be true definitely). Then, $dp[i][j] = max(dp[i-1][j], dp[i][j-1])$.

It brings us to the following simple code:

DP: LCS problem

```
int n, m;
int a[1..n], b[1..m], dp[0..n][0..m];

void calculateLCSDynamics() {
    for (int i = 0; i <= n; ++i)
        dp[i][0] = 0;
    for (int j = 0; j <= n; ++j)
        dp[0][j] = 0;

    for (int i = 1; i <= n; ++i)
        for (int j = 1; j <= m; ++j)
            if (a[i] == b[j])
                dp[i][j] = 1 + dp[i-1][j-1];
            else
                dp[i][j] = std::max(dp[i-1][j], dp[i][j-1]);
    }
}
```

The answer for the problem is $O(n^2)$. If it is necessary to restore LCS itself, one can build array $pr[1..n][1..m]$ and store information about one of two cases where the $dp[i][j]$ was found from, and then restore the answer by the same way as for LCS problem. The complexity of the solution is $O(n^2)$.