***Dynamic programming***

Dynamic programming is a special approach to solving problems. There is no single definition of dynamic programming, but let's try to form it anyway. The idea is that the optimal solution can often be found by considering all possible ways to solve the problem, and choosing the best among them.

The work of dynamic programming is very similar to recursion with the memorization of intermediate solutions — such recursion is also called memoization. Recursive algorithms tend to break down a large task into smaller subtasks and solve them. Dynamic algorithms divide the problem into pieces and calculate them one by one, building up solutions step by step. Therefore, dynamic algorithms can be represented as a bottom-up recursion.

The magic of dynamic programming lies in the clever handling of subproblem solutions. "Smart" in this context means "not solving the same subtask twice." For this, solutions to small subtasks must be saved somewhere. For many real-world dynamic programming algorithms, this data structure is a table.

In the simplest cases, this table will consist of only one row — similar to a regular array. These cases will be called one-dimensional dynamic programming, and consume O(n) memory. For example, an algorithm for efficiently calculating Fibonacci numbers uses a regular array to store the calculated intermediate results. The classic recursive algorithm does a lot of meaningless work — it calculates for the millionth time what has already been calculated in neighboring branches of recursion.

In the most common cases, this table will look familiar and consist of rows and columns. A regular table, similar to the tables from Excel. This is called two-dimensional dynamic programming, which, with n rows and n columns of the table, consumes O(n\*n) = O(n^2) memory. For example, a square table of 10 rows and 10 columns will contain 100 cells. Just such a task will be discussed in detail below.

***Greedy Algorithms***

So, the greedy algorithm is an algorithm that makes the locally best choice at every step in the hope that the final solution will be optimal.

For example, Dijkstra's algorithm for finding the shortest path in a graph is quite greedy, because at every step we look for the vertex with the lowest weight, which we have not yet visited, and then update the values of other vertices. At the same time, it can be proved that the shortest paths found at the vertices are optimal.

By the way, Floyd's algorithm, which also looks for shortest paths in the graph (though between all vertices), is not an example of a greedy algorithm. Floyd demonstrates another method, the dynamic programming method.

Using a greedy algorithm is pretty standard. Let's look at it using the example of the following task:

**The task of the schedule (CODE SOLUTION IN SEPARATE FILE).**

Let freelance programmer X be given n tasks. Each task has its own deadline, as well as its cost (that is, if he does not complete this task, then he loses so much money). X is so cool that he can do one task in one day. You can start completing the task from the moment 0. You need to maximize profits.

A classic example of the use of greed: It is profitable for X to do the most "expensive tasks", and the least expensive ones can not be performed — then the profit will be maximized. The question arises: how to distribute the tasks? We will sort through the tasks in descending order of cost and fill out the schedule as follows: if there is at least one more free place in the schedule for the order before its deadline, then we will put it at the very last of such places, otherwise we cannot complete it on time, so we will put it at the end of the available places.

Sometimes it may be tempting to use a greedy algorithm wherever possible, but on some tasks this is unacceptable. For example, the problem of a backpack: a thief broke into a warehouse that stores three items weighing 10 kg, 20 kg and 30 kg and worth 60, 100 and 120 dollars, respectively. A thief can carry a maximum of 50 kg. It is necessary to maximize the profit of the thief. If you act greedily here and choose the most valuable thing (that is, $ 6 per kg of the first piece, $ 5 per kg of the second and $ 4 per kg of the third), then the thief should take the first thing anyway, then there will be room for the second thing, but the optimal solution is the second and third thing.

Conclusion: There are no general rules for using greedy algorithms, but they are a very powerful tool applicable in competitive programming