Hands-on 03: Dynamic Programming

Student Jan Krejčí
Course Competitive Programming and Contests
Date December 2022

Holiday planning

First, the program verifies the validity of the input. If the input does not meet the task conditions the program exits with the appropriate message: `panic!("Invalid input! type of error")`. A new test set has been created for the purpose of input validity testing. The final implementation uses parts of an article on the GeeksforGeeks website (0-1 Knapsack Problem)[1].

In the next subsections, the naive recursion method and its optimized version using dynamic programming are described.

Naive Recursion Solution

A naive solution is to consider all subsets of visiting cities on different days and calculate the total number of visited attractions. The algorithm goes through all cities and considers all possible options (number of days spent in a current city). Then the variable **d** (number of remaining days) and **a** (number of visited attractions so far) is modified. Similarly with the rest of the cities. Recursion stops when there are no days left (d=0, circled in green) or the algorithm already iterated through all the cities. Finally, the maximum number of visited attractions is reported as the result.

This algorithm is implemented in function `lib::brute_force_method()`. Since this is a brute-force algorithm that is iterating through all subsets of the input the time complexity is exponential.

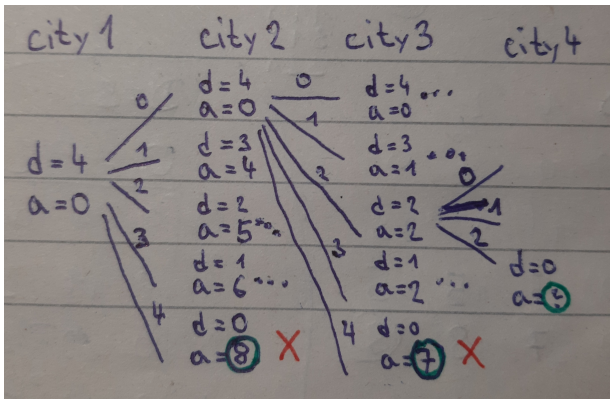

Figure 1: Visualisation of the naive solving method using recursion.

Solution using dynamic programming

The problem is that the previous algorithm is calculating some cases multiple times, which increases time complexity. It is a redundant case if it is the same pair `(city, number of days)`. As you can see in the example in Figure 2 (the number of visited attractions is not important). The whole idea is to store calculated results and use them when needed.

The practical solution is to store intermediate results in an extra 2D array. The array has a width of a number of days and the height of a number of cities and it is initialized with undefined values. The algorithm is a modified brute force method with only a few extra lines of code that handle an array. Before the function returns the result, it is stored in the array. This array is checked before every other calculation. If a result for combination `(city, number of days)` was already calculated, the value from the array is used.

[1] `https://www.geeksforgeeks.org/0-1-knapsack-problem-dp-10/`

Accessing an array is executed in constant time. In the worst case, the algorithm goes through all cities n and all possible combinations of days are $D * D$, as redundant calculations are avoided. Overall time complexity is $O(nD^2)$.

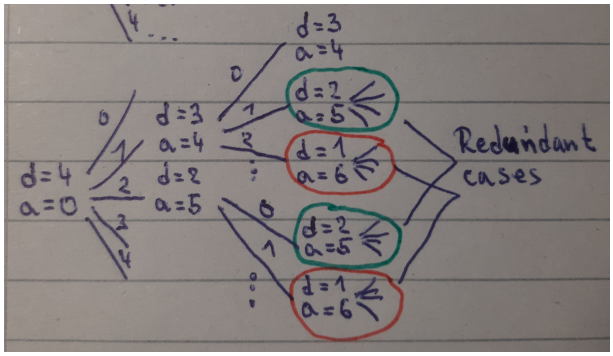

Figure 2: Visualisation of the problem of calculating redundant cases.

Xmas lights

Computing the number of patriotic selections of three houses can be done in $O(n)$ time by going through an array `H[1,n]` and using 3 counters for each color:

- *red counter* is incremented every time a red color house is found,
- *white counter* is incremented by a red counter value every time a white color house is found,
- *green counter* is incremented by a white counter value every time a green color house is found. In the end, the value in this counter is the total number of patriotic selections of three houses in array H.

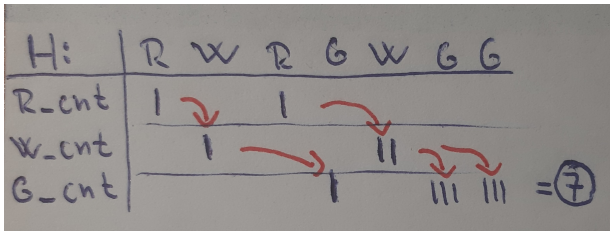

Figure 3: Visualisation of the linear process for getting the number of patriotic selections of three houses.

Unassigned light color

To solve the problem of houses with unassigned lights we need to use another counter. When the house with the color X is found, all counters are multiplied by 3 which simulates the state space expansion. The value of a new counter is used for increasing the red counter. That still preserves the linear time complexity. The final code is very simple as you can see below.

```
for c in h {
    match c {
        'R' => r_cnt += x_cnt,
        'W' => w_cnt += r_cnt,
        'G' => g_cnt += w_cnt,
        'X' => {
            g_cnt = base * g_cnt + w_cnt;
            w_cnt = base * w_cnt + r_cnt;
            r_cnt = base * r_cnt + x_cnt;
            x_cnt = base * x_cnt;
}}}
```