Belgian Olympiad in Informatics

Carnival Training 2015 — Day 2

Dynamic Programming

Trainer:
Floris Kint
Notes:
Floris Kint
Victor Lecomte

February 15, 2015

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 Example 1: Fibonacci sequence

We use the Fibonacci sequence as a first example to introduce Dynamic Programming.

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.[1] 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 only takes n steps.

```
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];
}
```

We actually only need the last two Fibonacci numbers in every iteration, so we can even improve the space complexity from $O(n)$ to $O(1)$. Note that for this particular problem, even better algorithms exist. We only show this version to illustrate dynamic programming.

1.2 Example 2: Longest increasing subsequence

However, finding which sub-problems to solve is not always as easy, as the second example shows.

In the longest increasing subsequence problem, you have to find the longest increasing sequence of (not necessarily consecutive) elements in an array of length n. 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.

1.2.1 Naive recursive implementation

Another idea might be to recursively find the longest increasing subsequence which can be concatenated to a prefix of the array. This naive (recursive) implementation is shown in Listing 1. However, this algorithm has a very bad time complexity.

1.2.2 Store maximal sequence length for a certain ending value

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 can try to

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

Listing 1: Naive recursive implementation

```
1  int lis(vector<int> &a, int min_value, int start_index)
2  {
3    int best = 0;
4    int smallest_found = 999999999;
5    for(int i = start_index; i < a.size(); ++i)
6    {
7      if(a[i] > min_value && a[i] < smallest_found)
8      {
9        best = max(best, 1+lis(a, a[i], i+1));
10       smallest_found = a[i];
11     }
12   }
13   return best;
14 }
```

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. A possible implementation is shown in Listing 2. Beware: this assumes that the max value in array a is smaller than n (we could remap the values if this 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).

1.2.3 Store smallest ending value index for a certain sequence length

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 $a[i]$. 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. The implementation is shown in Listing 3.

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.

Listing 2: Storing maximal sequence length for a certain ending value

```
1  int lis(vector<int> &a)
2  {
3    int L=1; // Longest so far
4    vector<int> smallest_end_for;
5    smallest_end_for.push_back(-1);
6    smallest_end_for.push_back(0);
7    for(int i=1; i<a.size(); i++)
8    {
9      // Binary search for the best length before a[i]
10     int lower=0, upper=L+1;
11     while(lower+1 < upper)
12     {
13       int middle = (lower+upper)/2;
14       if(a[smallest_end_for[middle]] < a[i])
15         lower = middle;
16       else
17         upper = middle;
18     }
19     int prev_len = lower;
20     // If the length is the best so far
21     if(prev_len + 1 > L)
22     {
23       smallest_end_for.push_back(i);
24       L++;
25     }
26     // If this value is the new smallest for this length
27     else if(a[i] < a[smallest_end_for[prev_len+1]])
28     {
29       smallest_end_for[prev_len+1] = i;
30     }
31   }
32   return L;
33 }
```

Listing 3: Storing smallest ending value index for a certain sequence length

```
1  int lis(vector<int> &a)
2  {
3    int L=1; // Longest so far
4    vector<int> smallest_end_for;
5    smallest_end_for.push_back(-1);
6    smallest_end_for.push_back(0);
7    for(int i=1; i<a.size(); i++)
8    {
9      // Binary search for the best length before a[i]
10     int lower=0, upper=L+1;
11     while(lower+1 < upper)
12     {
13       int middle = (lower+upper)/2;
14       if(a[smallest_end_for[middle]] < a[i])
15         lower = middle;
16       else
17         upper = middle;
18     }
19     int prev_len = lower;
20     // If the length is the best so far
21     if(prev_len + 1 > L)
22     {
23       smallest_end_for.push_back(i);
24       L++;
25     }
26     // If this value is the new smallest for this length
27     else if(a[i] < a[smallest_end_for[prev_len+1]])
28     {
29       smallest_end_for[prev_len+1] = i;
30     }
31   }
32   return L;
33 }
```

1.3 Approaches

1.3.1 Bottom-up

The previous examples showed how you can use dynamic programming to build up a solution to a problem based on previously calculated smaller sub-problems. The algorithms first calculated the solutions to all smaller sub-problems, always using the previously calculated solutions to their predecessors. This approach is called *bottom-up*.

1.3.2 Top-down

Another approach to dynamic programming consist of caching results whenever they are calculated. Instead of first calculating the solutions to all smaller problems, we now wait to calculate them until we need them. As soon as we have calculated them, they are stored in a caching table. Whenever the algorithm needs the same solution, it can find it in the caching table instead of recalculating it.

1.4 2-dimensional DP

We illustrate the two approaches using a simple grid problem.

You are asked to find the minimal-cost path from the top-left corner to the bottom-right corner of the grid. You can only walk right and down to adjacent tiles. The cost of the path equals the sum of the values of the tiles you walk through.

It is easy to see that the cost to reach a tile is equal to the value of the tile plus the cost to reach the tile on top of it or plus the cost to reach the tile left of it. Figure 1.4 show the problem. Figure 1.4 shows for every tile the cost to get there.

The first algorithm shows the top-down approach. Whenever we need the cost to reach a certain tile, it checks whether the cost has been calculated. If this is not the case, it calculates the solution for this tile. If the cost was calculated earlier, it just uses the previously calculated value. The code is shown in Listing 4.

The second algorithm shows the bottom-up approach. We first populate the top row and left column of the table. Next, the algorithm fills all elements in the table row per row from left to right. In this step, it uses the values of the tile on top of the current tile and the tile left of the current tile. The code is shown in Listing 5.

1.5 Recognizing dynamic programming problems

Typical problems involving a dynamic programming solution include:

1. Sequences
    (a) Longest increasing subsequence

1	2	1	3
5	1	10	8
3	12	1	8
13	5	3	2

Figure 1: The grid with on each tile the cost to walk through it.

1 (1)	2 (3)	2 (5)	3 (8)
5 (6)	1 (4)	10 (14)	8 (16)
3 (9)	12 (16)	1 (15)	8 (23)
13 (32)	5 (21)	3 (18)	2 (20)

Figure 2: The grid with on each tile the cost to walk through it and between brackets the cheapest way to get there (cost to walk throught the tile itself included).

Listing 4: Top-down approach

```
1  int cheapest(int row, int col){
2    if(row == 0 && col == 0)
3      return grid[0][0];
4    if(calculated[row][col]==-1){
5      if(row == 0)
6        calculated[row][col] = grid[row][col] + cheapest(
             row, col-1);
7      else if(col == 0)
8        calculated[row][col] = grid[row][col] + cheapest(
             row-1, col);
9      else
10       calculated[row][col] = grid[row][col] + min(
             cheapest(row-1, col), cheapest(row, col-1));
11   }
12   return calculated[row][col];
13 }
```

Listing 5: Bottom-up approach

```
1  int cheapest(int row, int col){
2    calculated[0][0] = grid[0][0];
3    for(int c = 1; c <= col; ++c)
4      calculated[0][c] = calculated[0][c-1]+grid[0][c];
5    for(int r = 1; r <= row; ++r)
6      calculated[r][0] = calculated[r-1][0] + grid[r][0];
7    for(int r = 1; r <= row; ++r)
8      for(int c = 1; c <= col; ++c)
9        calculated[r][c] = grid[r][c] + min(calculated[r
             -1][c], calculated[r][c-1]);
10   return calculated[row][col];
11 }
```

   (b) Longest common substring
   (c) Recursive definitions of a series of number (e.g. Fibonacci)

2. Knapsack (Given a backpack of capacity C and a set S of items with a given size s_i and value v_i, choose a subset of S that fits in the backpack, maximizing the total value.)

3. 'Divide and Conquer' algorithms with overlapping sub-problems.

1.6 Applications

The Dynamic Programming technique is used in various well-known algorithms, such as:

1. Bellman-Ford algorithm (Single-source shortest path that can handle negative edge weights)

2. Floy-Warshall algorithm (All-pairs shortest path problem)

3. Dijkstra's algorithm (Single-source shortest path)

1.7 Exercises

1. Introduction
   (a) UVa Online Judge 624: 0-1 Knapsack (Compare bruteforce and DP solution)
   (b) UVa Online Judge 562: 0-1 Knapsack
   (c) UVa Online Judge 108: Maximum 2D Sum
   (d) UVa Online Judge 111: Longest common substring

2. Advanced
   (a) Facebook Hacker Cup 2015 - Round 1: Winning at Sports
   (b) UVa Online Judge 709: Formatting text
   (c) `http://uva.onlinejudge.org/external/116/p11691.pdf`
   (d) `http://uva.onlinejudge.org/external/117/p11766.pdf`