Saturday, May 20, 2017

Notes on Dynamic Programming

- Iteration vs. recursion
-
- After you read some introductory texts on dynamic programming (which I highly recommend), pretty much all the source code examples in them use bottom-up technique with iteration (i.e. using for-cycles). For example calculating the length of the longest common subsequence of two strings A and B of length N, would look like this:
-

```
int dp[N+1][N+1];
for (int i = 0; i <= N; ++i)
  dp[0][i] = dp[i][0] = 0;
for (int i = 1; i <= N; ++i)
  for (int j = 1; j <= N; ++j) {
    dp[i][j] = max(dp[i-1][j], dp[i][j-1]);
    if (A[i-1] == B[j-1])
      dp[i][j] = max(dp[i][j], dp[i-1][j-1]+1);
  }
int answer = dp[N][N];
```

-
-
- There are couple of reasons why it is coded in this way:
- iteration is much faster than recursion
- one can easily see time and space complexity of the algorithm
- source code is short and clean

- Looking at such source code, one can understand how and why it works, but it is much harder to understand how to come up with it.
-
- The biggest breakthrough in my learning of dynamic programming was, when I started to think about the problems in the top-down fashion, instead of bottom-up.
-
- On the first look it doesn't look as such a revolutionary insight, but these two approaches directly translates in two different source codes. One uses iteration (bottom-up fashion) and the other one uses recursion (top-down fashion). The latter one is also called the memoization technique. The two solutions are more or less equivalent and you can always transform one into the other.
-
- In the following paragraphs I will show you how to come up with a memoization solution for a problem.
-
- Motivation problem
-
- Imagine you have a collection of N wines placed next to each other on a shelf. For simplicity, let's number the wines from left to right as they are standing on the shelf with integers from 1 to N, respectively. The price of the i-th wine is pi (prices of different wines can be different).
-
- Because the wines get better every year, supposing today is the year 1, on year y the price of the i-th wine will be y*pi, i.e. y-times the value that current year.
-
- You want to sell all the wines you have, but you want to sell exactly one wine per year, starting on this year. One more constraint - on each year you are allowed to sell only either the leftmost or the rightmost wine on the shelf and you are not allowed to reorder the wines on the shelf (i.e. they must stay in the same order as they are in the beginning).
-

- You want to find out, what is the maximum profit you can get, if you sell the wines in optimal order.
-
- So for example, if the prices of the wines are (in the order as they are placed on the shelf, from left to right): p1=1, p2=4, p3=2, p4=3
- The optimal solution would be to sell the wines in the order p1, p4, p3, p2 for a total profit 1*1 + 3*2 + 2*3 + 4*4 = 29
-
- Wrong solution
-
- After playing with the problem for a while, you'll probably get the feeling, that in the optimal solution you want to sell the expensive wines as late as possible. You can probably come up with the following greedy strategy:
-
- Every year, sell the cheaper of the two (leftmost and rightmost) available wines.
-
-
- Although the strategy doesn't mention what to do when the two wines cost the same, this strategy kinda feels right. But unfortunately, it isn't, as the following example demonstrates. If the prices of the wines are: p1=2, p2=3, p3=5, p4=1, p5=4
-
- The greedy strategy would sell them in the order p1, p2, p5, p4, p3 for a total profit 2*1 + 3*2 + 4*3 + 1*4 + 5*5 = 49
-
- But we can do better if we sell the wines in the order p1, p5, p4, p2, p3 for a total profit 2*1 + 4*2 + 1*3 + 3*4 + 5*5 = 50
-
- This counter-example should convince you, that the problem is not so easy as it can look on a first sight and I will tell you, that it can be solved using DP.

Saturday, May 20, 2017

-
- Write a backtrack
-
- When coming up with the memoization solution for a problem, I always start with a backtrack solution that finds the correct answer. Backtrack solution enumerates all the valid answers for the problem and chooses the best one. For most of the problems it is easy to come up with such solution.
-
- Here are some restrictions I put on a backtrack solution:
- it should be a function, calculating the answer using recursion
- it should return the answer with return statement, i.e. not store it somewhere
- all the non-local variables that the function uses should be used as read-only, i.e. the function can modify only local variables and its arguments.
-
- So for the problem with wines, the backtrack solution will look like this:
-

```
- int p[N]; // read-only array of wine prices
- // year represents the current year (starts with 1)
- // [be, en] represents the interval of the unsold wines on the shelf
- int profit(int year, int be, int en) {
-   // there are no more wines on the shelf
-   if (be > en)
-     return 0;
-   // try to sell the leftmost or the rightmost wine, recursively calculate the
-   // answer and return the better one
-   return max(
-     profit(year+1, be+1, en) + year * p[be],
```

- profit(year+1, be, en-1) + year * p[en]);
- }
-
-
- We can get the answer by calling:
-
- int answer = profit(1, 0, N-1); // N is the total number of wines
-
-
- This solution simply tries all the possible valid orders of selling the wines. If there are N wines in the beginning, it will try 2^N possibilities (each year we have 2 choices). So even though now we get the correct answer, the time complexity of the algorithm grows exponentially.
-
- The correctly written backtrack function should always represent an answer to a well-stated question. In our case profit function represents an answer to a question: "What is the best profit we can get from selling the wines with prices stored in the array p, when the current year is year and the interval of unsold wines spans through [be, en], inclusive?"
- You should always try to create such a question for your backtrack function to see if you got it right and understand exactly what it does.
-
- Minimize the state space of function arguments
-
- In this step I want you to think about, which of the arguments you pass to the function are redundant. Either we can construct them from the other arguments or we don't need them at all. If there are any such arguments, don't pass them to the function. Just calculate them inside the function.
-

- In the above function profit, the argument year is redundant. It is equivalent to the number of wines we have already sold plus one, which is equivalent to the total number of wines from the beginning minus the number of wines we have not sold plus one. If we create a read-only global variable N, representing the total number of wines in the beginning, we can rewrite our function as follows:
-

```
- int N; // read-only number of wines in the beginning
- int p[N]; // read-only array of wine prices
- int profit(int be, int en) {
-   if (be > en)
-     return 0;
-   // (en-be+1) is the number of unsold wines
-   int year = N - (en-be+1) + 1; // as in the description above
-   return max(
-     profit(be+1, en) + year * p[be],
-     profit(be, en-1) + year * p[en]);
- }
```

-
-
- I also want you to think about the range of possible values the function arguments can get from a valid input. In our case, each of the arguments be and en can contain values from 0 to N-1. In valid inputs we also expect be <= en+1. Using big-O notation we can say, there are O(N^2) different arguments our function can be called with.
-
- Now cache it!
-
- We are now 99% done. To transform the backtrack function with time complexity O(2^N) into the memoization solution with time complexity O(N^2) we will use a little trick which doesn't require almost any thinking.

-
- As noted above, there are only O(N^2) different arguments our function can be called with. In other words, there are only O(N^2) different things we can actually compute. So where does O(2^N) time complexity comes from and what does it compute?!
-
- The answer is - the exponential time complexity comes from the repeated recursion and because of that, it computes the same values again and again. If you run the above code for an arbitrary array of N=20 wines and calculate how many times was the function called for arguments be=10 and en=10 you will get a number 92378. That's a huge waste of time to compute the same answer that many times. What we can do to improve this is to cache the values once we have computed them and every time the function asks for an already cached value, we don't need to run the whole recursion again. See the code below:
-

```
int N; // read-only number of wines in the beginning
int p[N]; // read-only array of wine prices
int cache[N][N]; // all values initialized to -1 (or anything you choose)
int profit(int be, int en) {
  if (be > en)
    return 0;
  // these two lines save the day
  if (cache[be][en] != -1)
    return cache[be][en];
  int year = N - (en-be+1) + 1;
  // when calculating the new answer, don't forget to cache it
  return cache[be][en] = max(
    profit(be+1, en) + year * p[be],
    profit(be, en-1) + year * p[en]);
}
```

-
-
- And that's it! With that little trick it runs O(N^2) time, because there are O(N^2) different arguments our function can be called with and for each of them, the function runs only once with O(1) time complexity.
-
- Note: when the values are cached, you can treat every recursive call inside the function as it would run in O(1) time complexity.
-
- Summary
-
- To sum it up, if you identify that a problem can be solved using DP, try to create a backtrack function that calculates the correct answer. Try to avoid the redundant arguments, minimize the range of possible values of function arguments and also try to optimize the time complexity of one function call (remember, you can treat recursive calls as they would run in O(1) time). Finally cache the values and don't calculate the same things twice.
-
- The final time complexity of the solution is:
- range_of_possible_values_the_function_can_be_called_with x
- time_complexity_of_one_function_call.