Dynamic Programming & Split and List

Felix Weissenberger

October 18, 2017

Dynamic Programming – Outline

- Most of you know Dynamic Programming (DP) :)
- Many struggle to apply it :(
  - How to identify a DP problem
  - How to tackle it
  - How to implement it
- Today we start from scratch
- Focus on solving Algolab problems with DP

First Example: Fibonacci Numbers

Definition: $F_1 = 1$, $F_2 = 1$, and $F_n = F_{n-1} + F_{n-2}$ for $n > 2$

Task: compute F_n.

Solution: transform definition into recursive algorithm.

```
int f(int i) {
  if(i == 1 || i == 2) return 1;
  return f(i-1) + f(i-2);
}
```

Time complexity: $\Theta(\phi^n)$

Source of inefficiency? Overlapping Subproblems...

Fibonacci Numbers – Memoization

Recall: $F_1 = 1$, $F_2 = 1$, and $F_n = F_{n-1} + F_{n-2}$ for $n > 2$

Idea: do not recompute, recall from memory

```
map<int,int> memo;

int f(int i) {
  if(i == 1 || i == 2) return 1;
  if(memo[i] == memo.end()) // not in memory
    memo[i] = f(i-1) + f(i-2);
  return memo[i];
}
```

Time complexity: $\Theta(n)$

Memoization (or top-down DP) is simple and powerful :)

This was easy. Why is it difficult in general?

Essence of DP:

- Compute a solution from solutions of subproblems.
- Solve subproblems only once, by storing results.

Storing results is easy, just apply memoization.

Deriving a recursive algorithm is the difficult part!

Usually we do not get a recursive definition of the problem... :(

Second Example: Drink as much as possible

Task: given a sequence of n bottles with volumes $v_1, \ldots, v_n$. Drink as much as you can, without drinking from two adjacent bottles.

We want a recursive definition for $f(i) :=$ "max amount we can drink from first i bottles".

- Base cases: $f(0) = 0$ and $f(1) = v_1$.
- $f(i) = \max\{v_i + f(i-2), f(i-1)\}$

Now we can transform this definition into a recursive algorithm.

```
int f(int i) {
  if(i == 0 || i == 1)
    return volumes[i] // trick: set volumes[0] = 0 ;)
  return max(volumes[i] + f(i-2), f(i-1));
}
```

Time complexity: $\Theta(\phi^n)$ (same as Fibonacci)

Drink as much as possible – Memoization

Take recursive algorithm

```
int f(int i) {
  if(i == 0 || i == 1)
    return volumes[i] // trick: set volumes[0] = 0 ;)
  return max(volumes[i] + f(i-2), f(i-1));
}
```

and simply add memo:

```
map<int,int> memo;

int f(int i) {
  if(i == 0 || i == 1) return volumes[i];
  if(memo[i] == memo.end())
    memo[i] = max(volumes[i] + f(i-2), f(i-1));
  return memo[i];
}
```

Time complexity: $\Theta(n)$

General Strategy

Find recursive formulation for the problem.

Implement it as recursive algorithm, it will be correct but slow.

Are there overlapping subproblems?

Add memoization!

What is left?

We focus on examples to illustrate particular difficulties that often occur in problems.

- Iterative DP (table, bottom-up)
- Compare memoization and iterative DP
- Reconstruct solutions
- Example with “complicated and many" subproblems

Drink as much as possible – Iterative DP

Recall: $f(0) = 0$ and $f(1) = v_1$ and $f(i) = \max\{v_i + f(i-2), f(i-1)\}$

We can easily transform this into an iterative algorithm:

```
int f(int n) {
  vector<int> dp(n+1); // dp table
  dp[0] = 0;
  dp[1] = volumes[1];
  for(int i = 2; i <=n; ++i)
    dp[i] = max(volumes[i] + dp[i-2], dp[i-1]);
  return dp[n];
}
```

The DP table follows naturally from recursive definition.

Memoization vs Iterative DP

Usually both work, so use what feels more natural to you ;)

Memoization:

- Simple (once you have recurrence)
- Easy to use other subproblem descriptions (e.g. vectors...)
- Only computes necessary subproblems
- Overhead of function calls
- Sometimes time complexity not obvious

Iterative DP (with table):

- More effort to code
- Need to describe subproblems with integers
- Computes always all subproblems
- Can sometimes be optimized to reuse memory
- Time complexity obvious

Drink as much as possible – Reconstruct Solution

We computed how much we can drink. What if we want to know which bottles to take?

1. Compute DP table or memo
2. Reconstruct solutions using recurrence and a stack that remembers where we come from.

Recall recurrence: $f(0) = 0$ and $f(1) = v_1$ and $f(i) = \max\{v_i + f(i-2), f(i-1)\}$

```
stack<int> partial; // partial solutions

void reconstruct(int i) {
  if(i == 0) return; // p contains a solution
  if(i == 1) partial.push(1); return; // p contains solution
  else{
    if(volume[i] + dp[i-2] > dp[i-1]) // we took the i-th bottle
      partial.push(i);
      reconstruct(i-2);
    else // we did not take the n-th bottle
      reconstruct(i-1);
  }
}
```

Last Example: Longest Increasing Subsequence in $\Theta(n^2)$

Task: given a sequence of n integers $a_1, \ldots, a_n$. Compute the length of a longest increasing subsequence (LIS).

First attempt: $f(i) :=$ "length of LIS in $a_1, \ldots, a_i$".

- Base cases: $f(0) = 0$
- $f(i) = ???$

Final attempt: $f(i) :=$ "length of LIS in $a_1, \ldots, a_i$ that ends in a_i".

- Base cases: $f(0) = 0$
- $f(i) = \max_{j<i : a_j \leq a_i} \{1 + f(j)\}$

We had to reformulate the problem s.t. it admits a recursive formulation, this is difficult!

Time complexity: n function calls (with memo), i-th call takes $\Theta(i)$ time. Thus, $\Theta(n^2)$.

DP – Wrap Up

- Idea of DP: solve subproblems only once by storing solutions of subproblems
- Start by defining recurrence relation (on paper)
- Implement it. It will be correct but slow...
- Are there overlapping subproblems?
- Add memo (usually this does the trick) or construct DP table
- Practice finding recurrence relation on paper for well known DP problems (SubsetSum, Knapsack, Coin Change, LCS, Edit Distance, LIS...)

Brute Force Tricks

Brute force: some problems are hard and we only know how to solve them by trying everything.

However, one can often do it a little bit smarter:

1. Heuristics (important in practice, not in AlgoLab)
2. Improve worst case complexity :)

We will see a trick called Split and List.

This trick is why there is "DES" and "triple-DES" but no "double-DES"...

Example: SubsetSum

Task: given a set $S \subseteq \mathbb{N}$ of size n, and $k \in \mathbb{N}$. Is there a subset $S' \subseteq S$ such that $\sum_{s \in S'} s = k$?

NP complete

There is a DP solution in $\Theta(n \cdot k)$, good for small k.

Here we assume n is small and k is large and solve it with brute force.

We want to check all subsets!

- Recursive algorithm
- Iterative algorithm

SubsetSum – Recursive

Task: given a set $S = \{s_1, \dots, s_n\} \subseteq \mathbb{N}$, and $k \in \mathbb{N}$. Is there a subset $S' \subseteq S$ such that $\sum_{s \in S'} s = k$?

We want a recursive definition of $f(i,j) :=$ "is there $S' \subseteq \{s_1, \dots, s_i\}$ s.t. $\sum_{s \in S'} s = j$"

- Base cases: $f(i,0) = \mathit{True}$, for all i and $f(0,j) = \mathit{False}$, for all $j > 0$.
- $f(i,j) = f(i-1, j-s_i) \vee f(i-1, j)$

Recursive algorithm:

```
bool f(int i, int j) {
  if(j == 0) return true;
  if(i == 0 || j != 0) return false;
  return f(i-1,j-elements[i]) || f(i-1,j);
}
```

Time complexity: $\Theta(2^n)$, ok for $n \approx 25$

SubsetSum – Iterative

How can we iterate over all subsets of an n element set? Trick: encode set in integer.

```
bool subsetsu(int i) {
  for(int s = 0; s < 1<<n; ++s){ // iterate through all subsets
    int sum = 0;
    for(int i = 0; i < n; ++i){
      if(s & 1<<i) sum += elements[i]; // if i-th element in subset
    }
    if(sum == k) return true;
  }
  return false;
}
```

Time complexity: $\Theta(n \cdot 2^n)$, ok for $n \approx 25$

Subset Sum – Faster? Split and List

Lemma: Let $S = S_1 \cup S_2$ and $S_1 \cap S_2 = \emptyset$. The following statements are eqivalent:

- There is a $S' \subseteq S$ with $\sum_{s \in S'} s = k$
- There are $S'_1 \subseteq S_1$ and $S'_2 \subseteq S_2$ such that $\sum_{s \in S'_1} s + \sum_{s \in S'_2} s = k$

Idea: use second statement to check the first.

Algorithm sketch:

- Split S into S_1 and S_2 of size $\approx \frac{n}{2}$
- Lists all subset sums of S_1 and S_2 in L_1 and L_2
- Sort L_2
- For each k_1 in L_1 check if there is k_2 in L_2 such that $k_1 + k_2 = k$.

Time complexity: $\Theta(n \cdot 2^{\frac{n}{2}})$, ok for $n \approx 50$