

Dynamic Programming

Jonathan Backer
backer@cs.ubc.ca

Department of Computer Science
University of British Columbia



July 5, 2007

Introduction

Reading:

- ▶ “Dynamic Programming”, 15 CLRS
- ▶ “Dynamic Programming”, 5.3 GT

Greedy works if some optimal solution contains the greedy choice.

- ▶ Dijkstra’s algorithm always adds the cheapest vertex to the shortest path tree (greedy).
- ▶ Dijkstra’s algorithm may not work with negative edge weights.

Dynamic programming tries all possible choices.

- ▶ Bellman-Ford’s algorithm attempts every one edge extension of shortest paths (exhaustive).
- ▶ Bellman-Ford’s algorithm works with negative edge weights.

Optimal substructure

Both Dijkstra’s and Bellman-Ford’s algorithms work because you can extend the optimal solution of a subproblem.

Definition

A problem has **optimal substructure** if some optimal solution is

- ▶ an optimal solution to a subproblem combined with
 - ▶ an optimal choice.
-
- ▶ Often don’t know which choice to make, so try them all.
 - ▶ May be efficient if subproblems overlap
 - ▶ $|V|$ paths to extend in Bellman-Ford (subproblems)
 - ▶ $|E|$ edges to extend with (choices)

Making change with coins

Problem

Given: Coin values $c_1, c_2, \dots, c_t$ with which to make change and the amount of change to be made n .

Wanted: Number of each coin to use $n_1, n_2, \dots, n_t$ such that sum of coins is n and fewest coins are used.

Denominations chosen so that greedy algorithm works, but not true in general.

Example

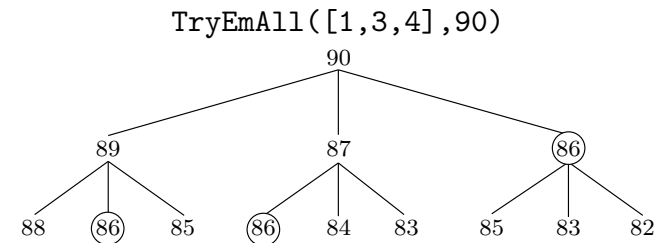
Coins: 1¢, 3¢, and 4¢	Greedy	→	4¢, 1¢, and 1¢
Change to make: 6¢	Optimal	→	3¢ and 3¢

Exhaustive coin changing

```
Algorithm TryEmAll( $C, n$ )
  int  $N[C.length]$ 
  for  $i \leftarrow 0$  to  $N.length-1$  do
     $N[i] \leftarrow 0$ 
  if  $n = 0$  then
    return  $N$ 
   $N[1] \leftarrow \infty$ 
  for  $i \leftarrow 0$  to  $C.length-1$  do
    if  $n \geq C[i]$  then
       $subprob \leftarrow TryEmAll(C, n - C[i])$ 
      if  $subprob.sum() + 1 < N.sum()$  then
         $N \leftarrow subprob$ 
         $N[i] \leftarrow N[i] + 1$ 
  return  $N$ 
```

Recursion tree for TryEmAll

This is inefficient because it recomputes the same subproblems over and over again.



A better idea: replace each recursive call with a table look-up.

- ▶ Construct a table to store the optimal solution for each n .
- ▶ Iteratively increase n and compute its entry.

Dynamic programming solution

```
Algorithm DPCoinChange( $C, n$ )
  int  $N[n+1][C.length]$ 
  for  $i \leftarrow 0$  to  $C.length-1$  do
     $N[0][i] \leftarrow 0$ 
  for  $m \leftarrow 1$  to  $n$  do
     $N[m][0] \leftarrow \infty$ 
    for  $i \leftarrow 0$  to  $C.length-1$  do
      if  $m \geq C[i]$  then
        if  $N[m - C[i]].sum() + 1 < N[m].sum()$  then
           $N[m] \leftarrow N[m - C[i]]$ 
           $N[m][i] \leftarrow N[m][i] + 1$ 
  return  $N[n]$ 
```

Runtime complexity

What is the runtime complexity of this algorithm?

- ▶ If it updates $N[m]$ every time, then $n \times t$ updates.
- ▶ Each update copies t integers.
- ▶ So $O(nt^2)$.

Advantages of eliminating the recursion:

- ▶ Counting argument for runtime complexity.
- ▶ No call stack overhead!

Why copy solution during update when we only chose one coin?

- ▶ Faster to remember the optimal choice and
- ▶ backtrack to recover the solution.

Faster solution

```
Algorithm FastDPCoinChange(C, n)
  int minCoins[n + 1], bestChoice[n + 1]
  minCoins[0] ← 0
  bestChoice[0] ← -1
  // main loop
  for m ← 1 to n do
    minCoins[0] ← ∞
    for i ← 0 to C.length-1 do
      if m ≥ C[i] then
        if minCoins[m - C[i]] + 1 < minCoins[m] then
          minCoins[m] ← minCoins[m - C[i]] + 1
          bestChoice[m] ← i
```

Backtracking

```
// backtracking
int N[C.length]
for i ← 0 to N.length-1 do
  N[i] ← 0
while n > 0 do
  N[bestChoice[n]] ← N[bestChoice[n]] + 1
  n ← n - C[bestChoice[n]]
return N
```

Eliminates copying and adding t integers from the innermost loop.
▶ Total runtime complexity is $O(nt)$.

Example: 1¢, 3¢, and 4¢ coins											
n	0	1	2	3	4	5	6	7	8	9	10
N	0	1	2	1	1	2	2	2	2		
bC	∅	1	1	3	4	1	3	3	4		

How do you make 9¢ and 10¢ change?

Designing a dynamic programming algorithm

Decide on the parameters the problem will have.

- ▶ This gives the “shape” of the table and determines the runtime complexity.
- ▶ FastDPCoinChange only used n to determine *bestChoice*, so table is an array.
- ▶ Limited supply of coins
 - ▶ Solution depends on n and number of each type of coin available ($a_1, a_2, \dots, a_t$).
 - ▶ Table has one dimension for n , another for a_1 , another for a_2 , etc.

What do we need to store in the table:

- ▶ DPCoinChange stored all of the best choices made so far.
- ▶ FastDPCoinChange just stored the last best choice.

Design (cont'd)

Express the problem in terms of smaller problems.

- ▶ FastDPCoinChange

$$\text{minCoins}[n] = \min\{1 + \text{minCoins}[n - C[i]] : C[i] \leq n\}$$

Determine how to fill-in the table.

- ▶ A subproblem solution must be computed before those that rely on it.
- ▶ Trickier for multi-dimensional tables. Typically row-by-row, column-by-column, or diagonal-by-diagonal.

Memoization (top-down)

Use divide-and-conquer to fill-in the table.

- ▶ Return value if already computed.
- ▶ Recurse otherwise.
- ▶ Save solution in table before returning.

Pros:

- ▶ If some subproblem is irrelevant, memoization won't solve it.
- ▶ If you cannot figure out how to fill the table, divide-and-conquer will do it for you.

Cons:

- ▶ Recursive function calling overhead (stack frame).
- ▶ Sometimes miss tricks like we used in `FastDPCoinChange`.