9 Dynamic Programming

Thursday, November 9, 2023 12:13 PM

by Richard Bellman

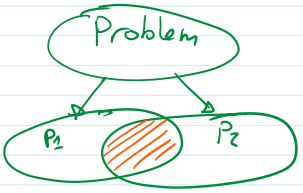
"Dynamic" "Programming"

"nothing called dynamic is ever bad"

scheduling.

optimizing.

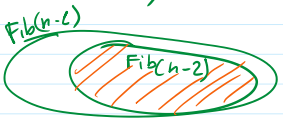
- Optimization Problems.
  - Problems with overlapping subproblems.



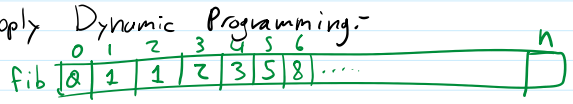
- Dynamic Programming - solve small subproblems only once, and store their solutions.

E.G. #0: Fibonacci

- Fib(n) = Fib(n-1) + Fib(n-2)



Apply Dynamic Programming:




```
FOR i = 2 TO n
    fib[i] = fib[i-1] + fib[i-2]
```

} runs n times.

Refine one more time.

```
q ← 0
p ← 1
FOR i = 2 TO n
    p ← p + q
    q ← p - q
RETURN p.
```

Not all Dynamic Programming Problems end up using a table, Sometimes that table collapses.

NOTE: Usually Dynamic Programming is used for Optimization Problems

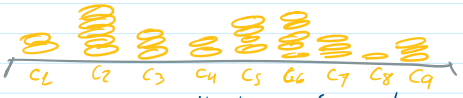
Principle of Optimality (true in many problems)

An Optimal solution to an instance of an optimization problem

Principle of Optimality (true in many problems)

An optimal solution to an instance of an optimization problem can be composed from the optimal solutions of its subinstances.

EG #1 The coin-Row problem:



$c_9 + F(7)$ vs $F(8)$

- pick up any collection of piles
- you cannot pick two adjacent piles

Goal: maximize the amount you pick.

- not as easy as "even" vs "odd"

- Consider Brute Force; 2^n candidate solutions.

Step 1: Formulate the optimal solutions in terms of optimal solutions to smaller instances:

$F(n)$: optimal solution to problem with n piles.

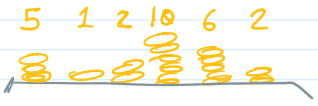
$$F(n) = \max \begin{cases} c_n + F(n-2) \\ F(n-1) \end{cases}$$

- $F(n) = \max\left(c_n + F(n-2),\ F(n-1)\right)$
- $F(0) = 0$
- $F(1) = c_1$

Algorithm.

```
Coin-Row( C[1...n] )
    VAR F[0...n]        <-- table of optimal solutions to subproblems.
    F[0] ← 0;  F[1] ← C[1]
    FOR i ← 2 TO n                          } n iterations.
    { F[i] ← max (C[i] + F[i-2], F[i-1])    }
    RETURN F[n].
```

TRACE:



$n = 6$

C	5	1	2	10	6	2	
F	0	5	5	7	15	15	17
	0	1	2	3	4	5	6
Sol	F	T	F	T	T	F	T

1, 4, 6

- Example #2: Change Making Problem.
  - Given a supply of coins of different denominations

- Example #2: Change Making Problem.
  - Given a supply of coins of different denominations

d_1	d_2	d_3	d_4	d_5	d_6

  - You want to complete a quantity n using the least number of coins:

e.g.: w American coins: $47¢$ = quarter + dime + dime + penny + penny
5 coins.

$F(n)$: optimal solution for n, number of coins.

$$F(n) = \begin{cases} d_1 + F(n-d_1) \\ d_2 + F(n-d_2) \\ d_3 + F(n-d_3) \\ \vdots \\ d_6 + F(n-d_6) \end{cases}$$

one of these must be the optimal solution

NOTE: $(n-d_i)$ cannot be negative.

Illustrate:

47¢
- quarter plus F(22)
- dime plus F(37)
- nickel plus F(42)
- penny plus F(46)
- half ∅

More formally:

$$F(n) = \min_{j \text{ s.t. } n \geq d_j} \{F(n-d_j)\} + 1$$

$$F(0) = 0$$

Algorithm

```
ChangeMake( D[1..m], n )   #PRE: D's are in increasing order.
VAR F[0...n]
F[0] ← 0
FOR i ← 1 TO n DO
    temp ← ∞
    j ← 1
    WHILE j ≤ m AND i ≥ D[j] DO
        temp ← min( F[i - D[j]], temp)
        j ← j + 1
    F[i] ← temp + 1
RETURN F[n].
```

Trace: D: 1 3 4 j

n = 8

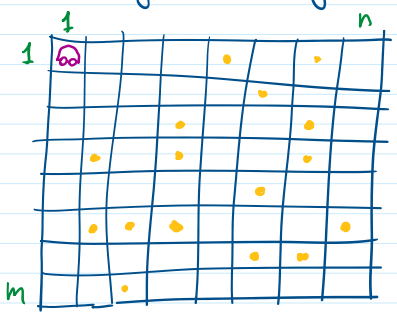
i	0	1	2	3	4	5	6	7	8
F:	0	1	2	1	1	2	2	2	2

Analysis:

Analysis:
- Recomputing solutions is avoided.
- $O(n \cdot m)$

- Example #3 : collecting coins:

Imagine the "grid-world"



: moves.

Goal: reach the bottom-right corner with the maximum coins collected.

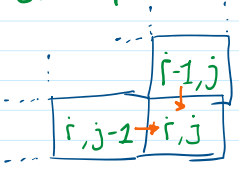
Brute force:

Consider all possible candidate plans:
- all plans consists of 7 moves right and 7 moves down
- there are 2^{14} candidate plans: $2^{14} \approx 16k$

Dynamic Programming.

$F(i,j)$: optimal solution: max coins collected, ending in i,j (location)

$F(m,n)$



Ask:

$F(i-1,j) + C(i,j)$

v.s.

$F(i,j-1) + C(i,j)$

$C(i,j)$: number of coins at i,j

$$F(i,j) = \max\{F(i-1,j), F(i,j-1)\} + C(i,j)$$

$F(0,j) = 0$
$F(i,0) = 0$

Note: table of optimal solutions is now a 2D array

Algorithm:

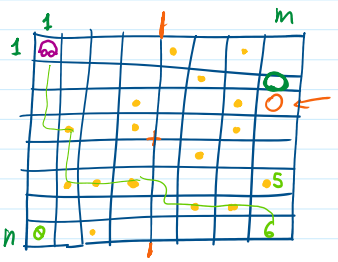
MarvinCoinCollecting ($C[1..n, 1..m]$)

VAR $F[0..n, 0..m]$: initialized to zeroes.

$F[1,1] \leftarrow C[1,1]$

- FOR $j \leftarrow 2$ TO m DO
  $F[1,j] \leftarrow F[1,j-1] + C[1,j]$
  # initialize first row.

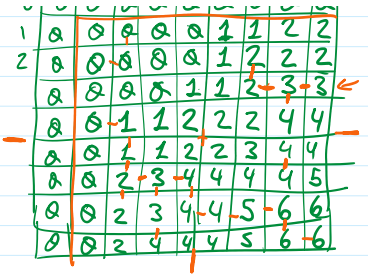
FOR $i \leftarrow 2$ TO n DO
  $F[i,2] \leftarrow F[i-1,1] + C[i,1]$
  # initialize first cell in row



F

	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	1	1	2	2
2	0	0	0	0	0	1	2	2	2
	0	0	0	0	1	1	2	3	3

```
    F[i,2] ← F[i-1,1] + C[i,1]
      # initialize first cell in row
    FOR j ← 2 to m DO
      F[i,j] ← max{ F[i-1,j], F[i,j-1] } + C[i,j]
RETURN F[n,m]
```



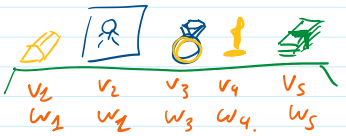
Analysis:

We move from an exponential Brute Force $2^{(n+m)}$ to filling a table of size $n+m$, $O(n \cdot m)$

Example 4#: KnapSack.

- PROBLEM Knapsack Problem

Given a collection of items. $e_1, e_2 \ldots e_n$
each with a weight $w_1, w_2 \ldots w_n$
and a value $v_1, v_2, \ldots v_n$.



and a knapsack of capacity W

? Which items to load on the knapsack to maximize value?

What you need is to select the subset $\{e_1, \ldots e_n\}$ that
- total weight is less than W
- maximizes value.

Brute Force = 2^n subsets.

- Restate the optimal solution to a problem instance in terms of optimal solutions to smaller instances.

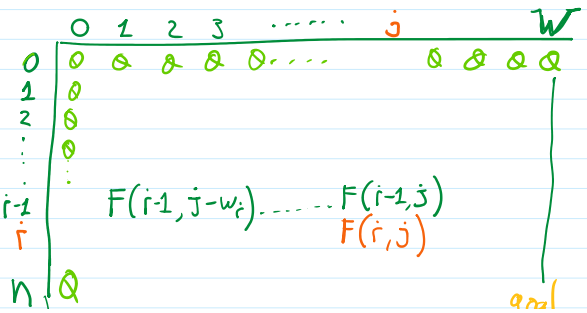
$F(i,j)$: optimal solution: for considering the first i items, and a bag of capacity j

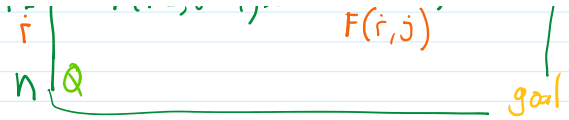
$$F(i,j) = \max \begin{cases} F(i-1,j) & \text{— not to carry item } i \\ \text{vs} & \\ F(i-1, j-w_i) + v_i & \leftarrow \text{do carry item } i \end{cases}$$

Note.: as long as $j \geq w_i$

$j - w_i$: remaining capacity of bag.

This restatement points to the use of a 2D table:





Algorithm:

```
DPKnapsack(w[1..n], v[1..n], W)
    VAR F[0..n, 0..W]
    Initialize F to zeroes.
    FOR i ← 1 TO n DO
      FOR j ← 1 TO W DO
          IF j - w[i] < 0 THEN
              F[i,j] ← F[i-1,j]
          ELSE
              F[i,j] ← max { F[i-1,j]
                           { F[i-1, j-w[i]] + v[i]
    RETURN F[n,W]
```

EOF.

Trace.

	1	2	3	4
w	2	1	3	2
v	$12	$10	$20	$15

W = 5

item \ w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	12	12	12
2	0	10	12	22	22	22
3	0	10	12	22	30	32
4	0	10	15	25	30	37

(1) (2) ~~(3)~~ (4)