9 Dynamic Programming

Thursday, November 21, 2024 11:51 AM

by Richard Bellman.

"Dynamic Programming" = Scheduling.

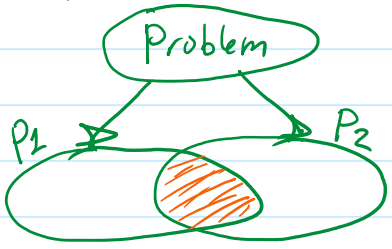
"nothing called Dynamic is ever bad"

you are trying to optimize schedules.

- Optimization Problems

Many problems are recurrences

Many problems have overlapping subproblems

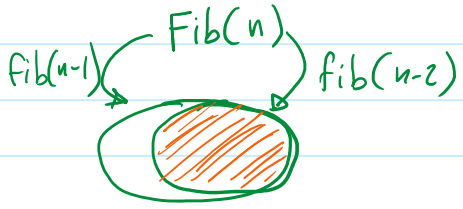


"Dynamic Programing" :- (the dummy version)

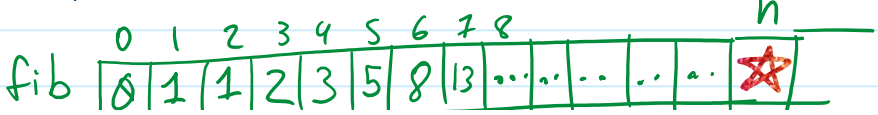
Solve small subproblems only once, and store their solutions.

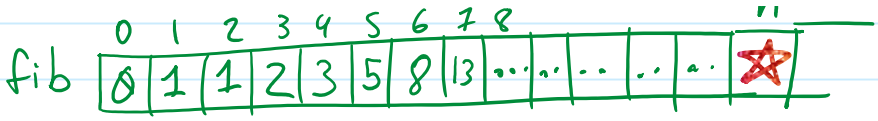
E.g #0: Fibonacci

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



Apply Dynamic Programming.





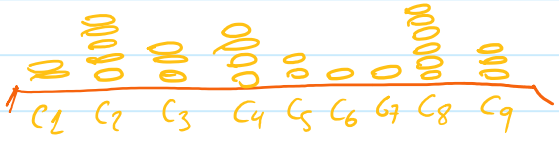

```
FOR i=2 to n DO
    fib[i] = fib[i-1] + fib[i-2]
```
} runs n times.

Note: not all Dynamic Programming Problems end up using a table.
Sometimes the table collapses.

NOTE: Dynamic Programming is used for Optimization Problems.

"Principle of Optimality"
"An Optimal Solution to an optimisation problem can be composed from the optimal solutions of it's subproblem"

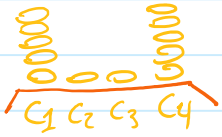
Problem: #1 - the coin-row-table problem.



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

Goal: Maximize the num coins you pick

"pick even piles" not optimal.



Consider Brute force: 2^n candidate solutions.

D.P. Step 1: Formulate optimal solution as a recurrance
i.e. in terms of optimal solutions to smaller instances

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

$$F(n): \begin{cases} C_n + F(n-2) & \text{pick pile } C_n \\ \text{or} & \\ F(n-1) & \text{reject pile } C_n \end{cases}$$

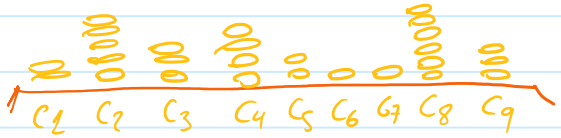
- $F(n) : \max\left(C_n + F(n-2),\ F(n-1)\right)$

$F(0) : 0$
$F(1) : C_1$ } Base Case

Step 2: Build a table of solutions from smaller to larger.

```
coin_row_select( C[1..n] )
    VAR  F[0...n]
    F[0] ← 0
    F[1] ← C[1]
    FOR i ← 2 to n
        F[i] ← max(C[i] + F[i-2], F[i-1])
    RETURN F[n]
```

Trace:



C: 2 5 3 4 2 1 1 6 3

F	0	2	5	5	9	9	10	10	16	16

Analysis.

Building solution table is $\Theta(n)$

Problem #2: Change Making Problem.

- given supply of coins of different denominations

d_1	d_2	d_3	d_4	d_5	d_6

complete a given quantity n
by using the least number of coins.

e.g. (American) 47¢ = quarter + dime + dime + penny + penny
5 coins.

Step 1:

$F(n)$: optimal solution for n,

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

one of these is the optimal solution

Note $n-d_i$ cannot be negative.

More Formally

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

$$F(0) = 0$$

Step #2: Build the table.

```
MakeChange ( D[1..m], n )
  VAR F[0...n]
  F[0] ← 0
  FOR i ← 1 to n DO
      temp ← ∞
      j ← 1
      WHILE j ≤ m AND i ≥ D[j] DO
```

Trace.

D: 1 3 4 6

n = 10

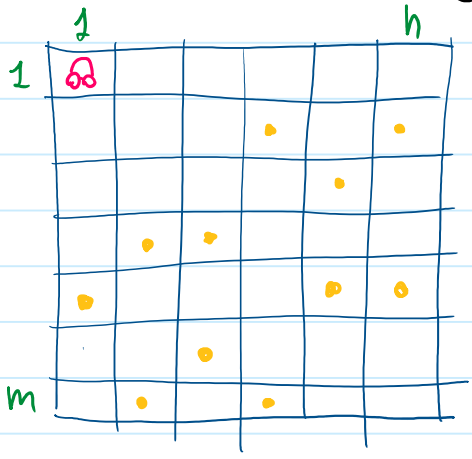
	0	1	2	3	4	5	6	7	8	9	10
F	0	1	2	1	1	2	1	2	2	2	2

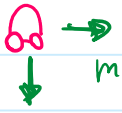
```
    j ← 1
    WHILE j ≤ m AND i ≥ D[j] DO
        temp ← min( F[i - D[j]], temp)
        j ← j + 1
    F[i] ← temp + 1
RETURN F[n]
```

Analysis:

worst case. $\Theta(n \cdot m)$

Problem #3: collecting coins in the grid world



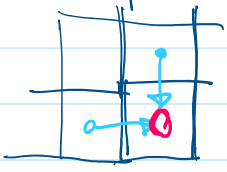


Steve's Goal:
collect the maximum number of coins.

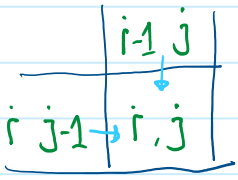
- Brute force

  Consider all possible plans.
  - all plans consist of 5 moves right and 6 moves down.
  - 2^{13} candidate plans. $\approx$ 8k plans.

- Step 1: Represent solution in terms of smaller solutions



F(i,j): coins collected ending in location i,j



$$F(i,j) \begin{cases} F(i-1,j) + C(i,j) \\ \max \\ F(i,j-1) + C(i,j) \end{cases}$$

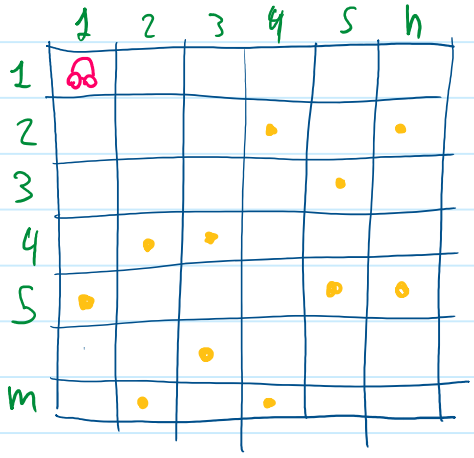
i.e.

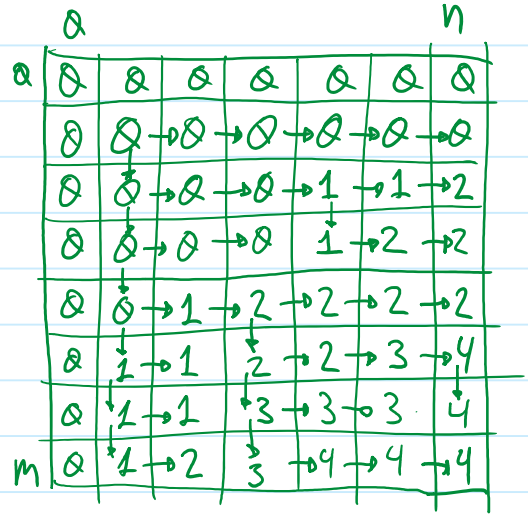
$$F(i,j) = \max\left(F(i-1,j), F(i,j-1)\right) + C(i,j)$$
$$F(1,1) = 0$$

Step 2 Build a table of solutions: 2D array

```
Steve Coin Collecting ( C[1..n, 1..m] )
 VAR F[0..n, 0..m] : initialized to zeroes
    F[1,1] ← C[1,1]
    FOR i ← 1 to n DO
      FOR j ← 1 to m DO
          F[i,j] = max (F[i-1,j], F[i,j-1])
                   + C[i,j]
    RETURN F[n,m]
```

Trace:

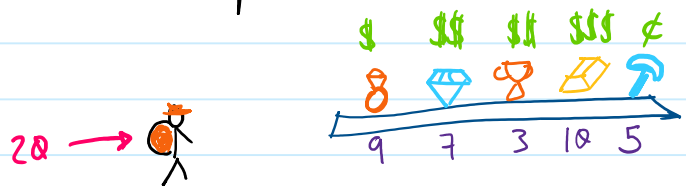




Analysis

Filling table is $\Theta(n \cdot m)$, not better $2^{n \cdot m}$

Problem #4: KnapSack.



Given a collection of items $e_1\ e_2\ e_3\ e_4\ \ldots\ e_n$
each with a weight $w_1\ w_2\ w_3\ w_4\ \ldots\ w_n$
and a value. $v_1\ v_2\ v_3\ v_4\ \ldots\ v_n$

and a "capacity" W

- Find a subset of the items s.t. the sum of the weights of the items is less than W that has maximum sum of values.

  Brute Force = check 2^n subsets.

- Apply Dynamic Programming

  Step 1: Reformulate solution in terms of solutions to smaller problems.

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

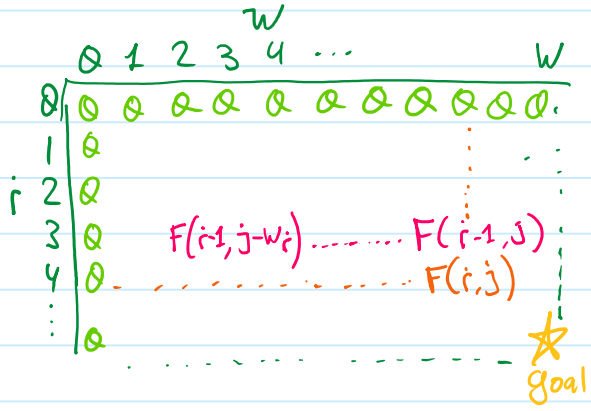
  $F(i,j)$: carry i'th item → $F(i-1,\ j-w_i) + v_i$ (weight of i; value of i)

  $F(i,j)$: don't carry i'th item. → $F(i-1,j)$

  $$F(i,j) = \max\left(F(i-1,j-w_i)+v_i,\ F(i-1,j)\right)$$
  $$F(x,0) = 0$$
  $$F(0,y) = 0$$

  Step-2 construct table of solutions

	0	1	2	3	4	...		W




```
DP knapsack (w[1..n], v[1..n], W)
VAR F[0..n, 0..W]
initialized 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]
```

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 ✗ ④

Analysis.

Filling table is $O(n \cdot W)$: lot better than 2^n