Programação Dinâmica

Algoritmos e Estrutura de Dados - IF672

Andreywid Yago Lima de Souza <ayls>
João Victor Nascimento <jvsn2>
Mateus Freire <mfvd>

Centro de Informática UFPE

Universidade Federal de Pernambuco

Programação Dinâmica

- Método Iterativo de solucionar problemas;
- Simplifica problemas complexos, transformando-os em subproblemas mais tragáveis
- Soluciona esses problemas mais simples primeiro, armazenando os valores numa tabela de resultados parciais;
- Troca o tempo de processamento por espaço do programa;
- Métodos
  - bottom-up: resolve todos os subproblemas;
  - top-down: resolve os subproblemas necessário;
- Alguns problemas: Fibonacci, menor caminho, knapsack;

Diferenças top-down, bottom-up

Top-down (memoization):

- Divide recursivamente o problema em subproblemas menores e guarda os valores na tabela de PD;
- Útil para problemas maiores e mais efetivo em guardar soluções de menos subproblemas;

Bottom-up (tabulação):

- Começa com o menor problema e gradualmente constrói a solução final, guardando seus valores na tabela de PD;
- Mais útil para problemas menores onde a solução ótima é mais facilmente encontrada a partir dos seus subproblemas

Exemplo: Fibonacci para um valor n

Na força bruta:

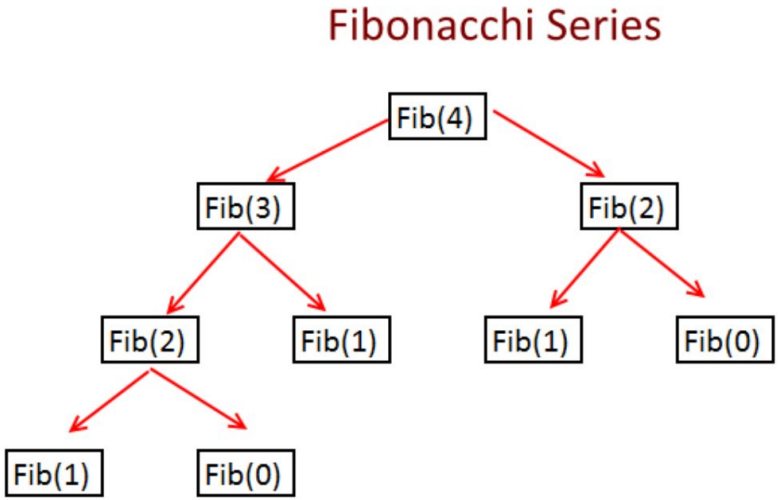
- fib(0) = 0 e fib(1) = 1;
- Para encontrar fib(n), calcula todos os valores abaixo de n: onde fib(n) = fib(n-1) + fib(n-2);

Com PD:

- Para cada f(n) checa a tabela para os valores f(n-1) e f(n-2);
- Garante que cada valor na tabela vai ser calculado apenas 1 vez;

Exemplo: Fibonacci para um valor n

Com PD:



Fibonacchi(N) = 0 for n = 0
= 1 for n = 1
= Fibonacchi(N-1)+Finacchi(N-2) for n > 1

Algoritmos usados em PD:

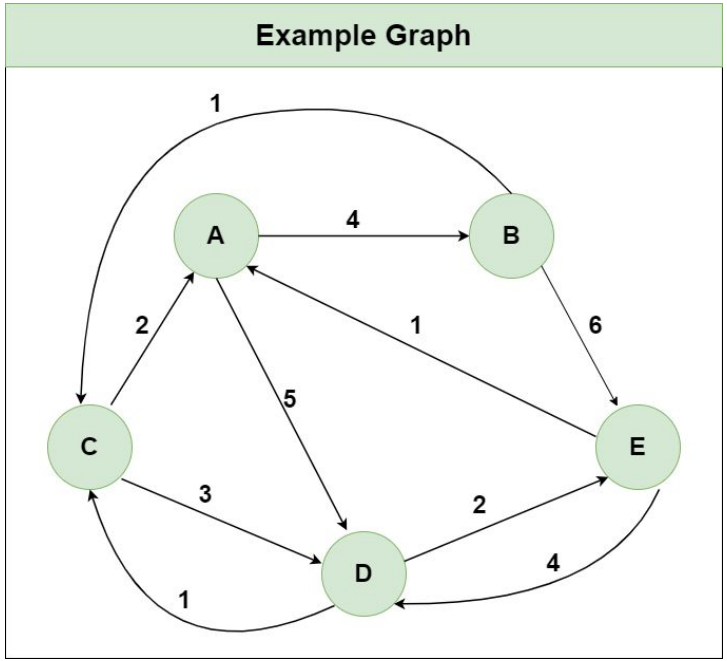
- **Floyd-Warshall:**
  - Usa PD para encontrar o menor caminho entre todos os pares de vértices de um grafo;
  - Funciona tanto para grafos dirigidos ou não dirigidos;
  - Usa a tabela de PD para guardar os valores entre os pares i.e. *Tabela[i][j] guarda o menor caminho entre os vértices i e j*

Algoritmos usados em PD:

- **Floyd-Warshall Algoritmo:**
  - Inicializa a matriz igual ao input da matriz de grafo;
  - Atualiza a matriz de solução considerando todos os vértices como intermediários no caminho;
  - Pega todos os vértices e atualiza todos os menores caminhos que incluem o vértice atual como intermediário no menor caminho;
  - Quando pegamos um vértice n, já consideramos todos os vértices < n como intermediários
  - Para cada par tem dois possíveis casos:
    - n não é um vértice intermediário, e então tab[i][j] continua igual;
    - atualizamos tab[i][j] como tab[i][n] + tab[n][j]
      se tab[i][j] > tab[i][n] + tab[n][j]

Algoritmos usados em PD:

- Exemplo de Floyd-Warshall:



Exemplo - Floyd-Warshall

Step 2: Using Node A as the Intermediate node

Distance[i][j] = min (Distance[i][j], Distance[i][A] + Distance[A][j])

	A	B	C	D	E
A	0	4	∞	5	∞
B	∞	?	?	?	?
C	2	?	?	?	?
D	∞	?	?	?	?
E	1	?	?	?	?

→

	A	B	C	D	E
A	0	4	∞	5	∞
B	∞	0	1	∞	6
C	2	6	0	3	12
D	∞	∞	1	0	2
E	1	5	∞	4	0

Exemplo - Floyd-Warshall

Step 3: Using Node B as the Intermediate node

Distance[i][j] = min (Distance[i][j], Distance[i][B] + Distance[B][j])

	A	B	C	D	E
A	?	4	?	?	?
B	∞	0	1	∞	6
C	?	6	?	?	?
D	?	∞	?	?	?
E	?	5	?	?	?

	A	B	C	D	E
A	0	4	5	5	10
B	∞	0	1	∞	6
C	2	6	0	3	12
D	∞	∞	1	0	2
E	1	5	6	4	0

Exemplo - Floyd-Warshall

Step 4: Using Node C as the Intermediate node

Distance[i][j] = min (Distance[i][j], Distance[i][C] + Distance[C][j])

	A	B	C	D	E
A	?	?	5	?	?
B	?	?	1	?	?
C	2	6	0	3	12
D	?	?	1	?	?
E	?	?	6	?	?

	A	B	C	D	E
A	0	4	5	5	10
B	3	0	1	4	6
C	2	6	0	3	12
D	3	7	1	0	2
E	1	5	6	4	0

Exemplo - Floyd-Warshall

Step 5: Using Node D as the Intermediate node

Distance[i][j] = min (Distance[i][j], Distance[i][D] + Distance[D][j])

	A	B	C	D	E
A	?	?	?	5	?
B	?	?	?	4	?
C	?	?	?	3	?
D	3	7	1	0	2
E	?	?	?	4	?

	A	B	C	D	E
A	0	4	5	5	7
B	3	0	1	4	6
C	2	6	0	3	5
D	3	7	1	0	2
E	1	5	5	4	0

Exemplo - Floyd-Warshall

Step 6: Using Node E as the Intermediate node

Distance[i][j] = min (Distance[i][j], Distance[i][E] + Distance[E][j])

	A	B	C	D	E
A	?	?	?	?	7
B	?	?	?	?	6
C	?	?	?	?	5
D	?	?	?	?	2
E	1	5	5	4	0

	A	B	C	D	E
A	0	4	5	5	7
B	3	0	1	4	6
C	2	6	0	3	5
D	3	7	1	0	2
E	1	5	5	4	0

O problema do knapsack

- Suponhamos que tenhamos uma mochila de capacidade x, onde queremos preencher com itens de valor y, com o melhor valor;

- Faz uma tabela com nº de colunas = x+1 e y+1 linhas;
- Inicializa a matriz com coluna[0] = 0 e linha[0] = 0;
- Faz o algoritmo linha por linha;
- Se a solução [i][j] não é o caso ótimo, repete o valor da linha anterior;
- Solução em [x][y];

O problema do knapsack

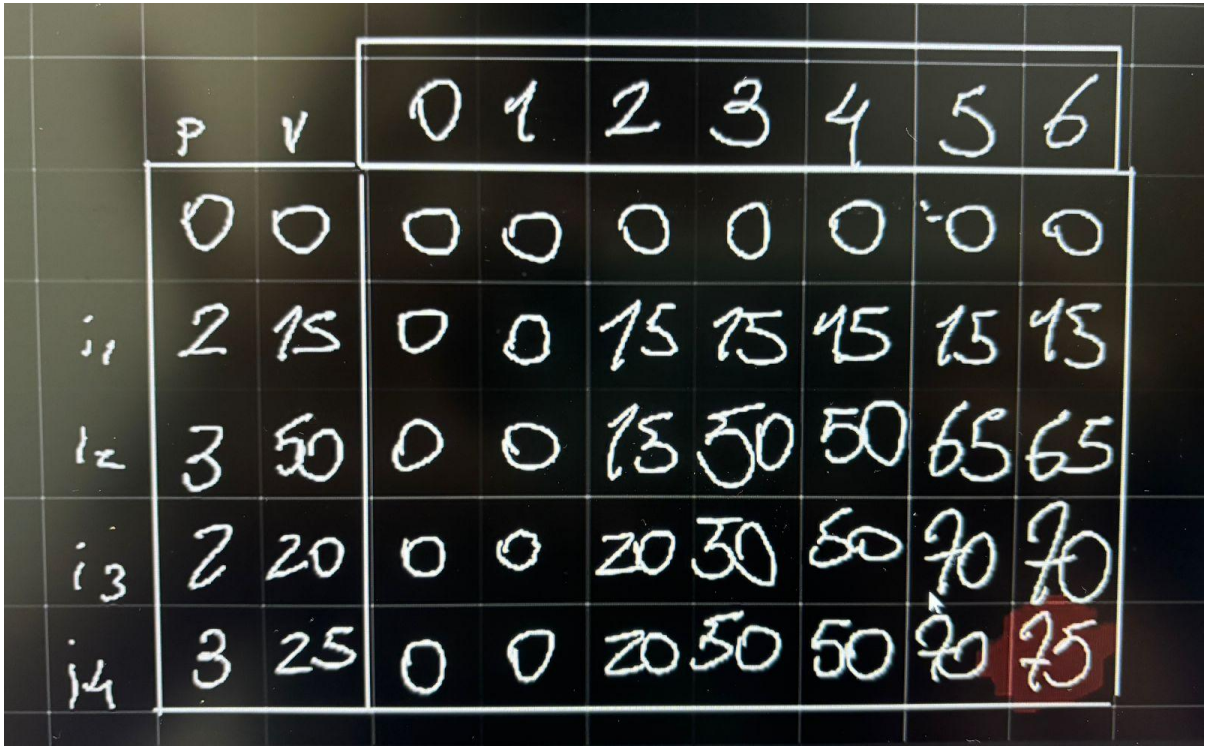
- Suponhamos que tenhamos uma mochila de capacidade x, onde queremos preencher com itens de valor y, com o melhor valor;
- W = {1, 2, 3}; P = {10, 15, 20}

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

Praticando com Zé

5. {2,0 pt.} Considerando uma mochila com capacidade de 6 kg, e os itens (peso, valor): $i_1 = (2, 15), i_2 = (3, 50), i_3 = (2, 20), i_4 = (3, 25)$, encontre o subconjunto de itens mais valioso que cabe na mochila. Só existe uma unidade de cada item e os itens são indivisíveis. Use programação dinâmica (*bottom-up*) e apresente a matriz construída na busca. A matriz tem tamanho *item* + 1 por *capacidade* + 1 e a primeira linha/coluna são preenchidas com 0.

Praticando com Zé



	p	v	0	1	2	3	4	5	6
	0	0	0	0	0	0	0	0	0
i_1	2	15	0	0	15	15	15	15	15
i_2	3	50	0	0	15	50	50	65	65
i_3	2	20	0	0	20	30	50	70	70
i_4	3	25	0	0	20	50	50	70	75

Centro de Informática UFPE

Universidade Federal de Pernambuco

Obrigado !!!

Estudem !!!