



Análise de algoritmos

# PROGRAMAÇÃO DINÂMICA

*Baseado no livro Entendendo Algoritmos, Um guia ilustrado para programadores e outros curiosos. Aditya Y. Bhargava, Novatec, 2017.*





Análise de algoritmos

# PROGRAMAÇÃO DINÂMICA

Retomando o problema da bagagem, você precisa organizar a sua para realizar uma viagem que possui limitação de espaço e itens candidatos a irem na mala, uma vez que cada item possui peso e valor. O objetivo é selecionar itens que maximizem o valor total dentro da capacidade da mala.

Com a programação dinâmica, é possível identificar os conjuntos possíveis dos itens para maximizar o valor de forma que a bagagem possa ser dividida em bagagens menores independentes até descobrir a resolução do problema. Ao contrário, o algoritmo guloso indica a melhor escolha dentre os possíveis no momento baseado em um critério sem prever “danos colaterais”.

Sendo assim, os algoritmos de aproximação garantem proximidade da solução ótima, mas não é a própria fornecida por meio da programação dinâmica. Abaixo, vamos explorar algumas diferenças entre a programação dinâmica e o algoritmo guloso:





Análise de algoritmos

# PROGRAMAÇÃO DINÂMICA E ALGORITMO GULOSO

	ALGORITMO GULOSO	PROGRAMAÇÃO DINÂMICA
	. . . . .	. . . . .
TEMPO DE EXECUÇÃO	$O(n^2)$ , considerado de execução mais rápida.	$O(2^n)$ , considerado de execução lenta.
VANTAGENS	Simplicidade e eficiência.	Assegura a solução ótima e flexibilidade.
DESVANTAGENS	Não garante a solução ótima e depende de heurísticas*.	Complexidade e implementação.
COMPLEXIDADE DE ESPAÇO	Menor.	Maior.

\*a qualidade da solução depende da heurística utilizada para selecionar o próximo item como ordenar itens pela relação de variáveis e adicionar itens à bagagem, se possível.





Análise de algoritmos

# PROGRAMAÇÃO DINÂMICA

Relacionados às características da programação dinâmica, serão listadas com foco no problema da bagagem, mas que poderão ser aplicados aos demais casos:

- Todo problema de programação dinâmica inicia com uma tabela, em que o eixo  $X$  refere-se às capacidades das bagagens em quilos e no eixo  $Y$  são os itens que podemos escolher;
- Para cada linha/item (eixo  $X$ ), devemos avaliar qual coluna estará o valor do item que caberá na bagagem. Todos as possibilidades/capacidades disponíveis considerando somente a existência do item analisado, enquanto os demais itens ainda se encontram indisponíveis;
- A última coluna do eixo  $X$  corresponde ao melhor palpite atual do item, isto é, referente ao subproblema;
- Conforme for descendo as linhas da tabela, é importante considerar as linhas anteriores + linha atual, buscando maximizar os valores dos itens, ocupando a maior capacidade disponível possível;





Análise de algoritmos

# PROGRAMAÇÃO DINÂMICA

- Quando tivermos dois ou mais itens que caibam na bagagem, o critério de desempate será o valor máximo;
- A última linha da última coluna preenchida sempre será a combinação que maximiza o valor e capacidade ocupada pelos itens para o problema da bagagem, mas pode não se aplicar aos demais casos, como acontece para o problema da maior *substring* comum que considera o maior número da tabela, independente de onde esteja na tabela;
- Caso houver a necessidade de inclusão de algum item, incluir e calcular como última linha da tabela;
- Com o passar dos itens, o valor da coluna sempre deve ser igual ou superior ao anterior;
- A mudança da ordem das linhas não muda o resultado;
- Não é recomendável preencher a tabela a partir das colunas ao invés das linhas, pois o resultado pode mudar;





Análise de algoritmos

# PROGRAMAÇÃO DINÂMICA

- Caso houver algum item que não possui o peso correspondente na coluna, a tabela precisa ser modificada para atender esse item, sendo dividido proporcionalmente na tabela;
- Não é possível selecionar frações de um item quando não é possível selecionar o produto na totalidade. Nesse caso, usamos o algoritmo guloso partindo do item mais caro em diante e;
- A programação dinâmica só funciona quando os seus subproblemas são discretos, quando não são dependentes entre si. Exemplo: atrações turísticas de um mesmo lugar não será contabilizado o tempo total de deslocamento.

Sendo assim, a tabela começa vazia e quando a tabela for preenchida, a resposta do problema terá sido encontrada.





Análise de algoritmos

# PROGRAMAÇÃO DINÂMICA

## APLICAÇÕES

Amplamente empregado por buscar a solução ótima dentre de inúmeras possibilidades, a programação dinâmica encontra-se nos seguintes setores:

### OTIMIZAÇÃO DE ROTAS

- Caminho mínimo em grafos: algoritmos como *Dijkstra* e *Bellman-Ford* utilizam a programação dinâmica para encontrar o caminho mais curto entre dois vértices em um grafo ponderado;
- Problema do caixeiro viajante: a programação dinâmica pode ser utilizada para encontrar soluções ótimas para instâncias menores ou aproximações para instâncias maiores.

### BIOINFORMÁTICA

- Alinhamento de sequências: algoritmos como *Needleman-Wunsch* e *Smith-Waterman* utilizam a programação dinâmica para alinhar sequências biológicas identificando similaridades e diferenças;





Análise de algoritmos

# PROGRAMAÇÃO DINÂMICA

- Dobramento de proteínas: predição da estrutura tridimensional de proteínas.

## FINANÇAS

- Precificação de opções financeiras: como opções americanas;
- Problemas de portfólio: seleção de ativos para otimizar o retorno de um portfólio, considerando riscos e restrições.

## ENGENHARIA

- Corte de materiais: problemas de corte de materiais como chapas de metal, para minimizar o desperdício;
- Controle de estoque: gestão de estoque para minimizar custos e atender à demanda pode ser otimizada.







Análise de algoritmos

# PROGRAMAÇÃO DINÂMICA

## JOGOS

- Jogos de tabuleiro: modelados como problemas de decisão sequencial e resolvidos com programação dinâmica;
- Jogos de vídeo: a inteligência artificial de personagens em jogos de vídeo pode utilizar a programação dinâmica para tomar decisões estratégicas.

## COMPRESSÃO DE DADOS

- Codificação de *Huffman*: algoritmo de compressão de dados sem perdas, utiliza a programação dinâmica para construir uma árvore de códigos ótima.

Na sequência, veremos um problema de programação dinâmica que será repartido em problemas menores e sobrepostos, fundamental para analisar possibilidades de saltos, evitar cálculos redundantes e construção da solução de forma gradual.





Análise de algoritmos

# PROGRAMAÇÃO DINÂMICA

Um sapo está atravessando um rio. O rio é dividido em um certo número de unidades, e em cada unidade, pode ou não existir uma pedra. O sapo pode pular sobre uma pedra, mas não pode pular na água.

Dado uma lista de posições de pedras (em unidades) em ordem crescente, determine se o sapo pode atravessar o rio pulando na última pedra. Inicialmente, o sapo está na primeira pedra e presume-se que o primeiro salto deve ser de 1 unidade.

Se o último salto do sapo for de  $k$  unidades, seu próximo salto deve ser de  $k - 1$ ,  $k$  ou  $k + 1$  unidades. O sapo só pode saltar para a frente.

## Exemplo 1:

Entrada: **stones** = [0,1,3,5,6,8,12,17]

Saída: *true*

Explicação: O sapo pode saltar para a última pedra pulando 1 unidade





Análise de algoritmos

# PROGRAMAÇÃO DINÂMICA

para a 2ª pedra, depois 2 unidades para a 3ª pedra, depois 2 unidades para a 4ª pedra, depois 3 unidades para a 6ª pedra, 4 unidades para a 7ª pedra e 5 unidades para a 8ª pedra.

## Exemplo 2:

Entrada: **stones** = [0,1,2,3,4,8,9,11]

Saída: *false*

Explicação: Não há como pular para a última pedra, pois a lacuna entre a 5ª e a 6ª pedra é muito grande.

## Restrições:

- $2 \leq \text{stones.length} \leq 2000$
- $0 \leq \text{stones}[i] \leq 231 - 1$
- $\text{stones}[0] == 0$
- stones está ordenado em ordem estritamente crescente.





# 🔍 PROGRAMAÇÃO DINÂMICA



```
package main
```

```
import (  
    "fmt"  
)
```

```
type stone struct {  
    position int  
}
```





# PROGRAMAÇÃO DINÂMICA



```
type key struct {  
    index int  
    jump int  
}
```

```
func canCrossRecursive(stones []stone, index int, jump  
int, memory map[key]bool) bool {  
    if index == len(stones) - 1 {  
        return true  
    }  
}
```

```
k := key{index, jump}
```





# PROGRAMAÇÃO DINÂMICA



```
result := false
for j := index + 1; j < len(stones); j++ {
    newJump := stones[j].position -
stones[index].position
    if newJump >= jump - 1 && newJump <= jump + 1
&& canCrossRecursive(stones, j, newJump, memory) {
        result = true
        break
    }
}

memory[k] = result
```





# PROGRAMAÇÃO DINÂMICA



```
    return result
}

func canCross(stones []stone) bool {
    if stones[1].position != 1 {
        return false
    }

    memory := make(map[key]bool)

    return canCrossRecursive(stones, 1, 1, memory)
}
```





# PROGRAMAÇÃO DINÂMICA



```
func main() {  
    stoneSequence1 := []stone{{0}, {1}, {3}, {5}, {6}, {8},  
    {12}, {17}}  
    fmt.Println(canCross(stoneSequence1))  
  
    stoneSequence2 := []stone{{0}, {1}, {2}, {3}, {4}, {8},  
    {9}, {11}}  
    fmt.Println(canCross(stoneSequence2))  
}
```







# SAÍDA DO PROGRAMA



**CENÁRIO OTIMISTA**

true

**CENÁRIO PESSIMISTA**

false





Análise de algoritmos

# CONSIDERAÇÕES

Para o exemplo apresentado, seguiu-se as boas práticas de desenvolvimento com *clean code* e SOLID, na tentativa de simular um cenário otimista.

Essa iniciativa vai de encontro com a ideia de trazer conteúdos relevantes altamente abordados em processos seletivos e desmitificar a ideia de algoritmos e estrutura de dados. Espero que seja de bom proveito e bons estudos.