Curso Estruturas de Dados e Algoritmos Expert

Prof. Nelio Alves

Programação dinâmica (parte 1)

DEVSUPERIOR

https://devsuperior.com.br

Por que estudar Programação Dinâmica?

https://devsuperior.com.br

Prof. Dr. Nelio Alves

Programação Dinâmica

- Na programação, muitos problemas desafiadores envolvem **otimização**
  - Encontrar solução que **minimize ou maximize** alguma função

- Exemplos conhecidos
  - Qual o **número mínimo de moedas** para dar o troco?
  - Qual o **lucro máximo** ao comprar e vender ações?
  - Qual o **menor caminho** até uma localização?

- Este tipo de algoritmo requer que provemos que sempre retornam a melhor resolução possível (otimalidade).

Programação Dinâmica

- Como abordar problemas de otimização?

- **Busca Completa (força bruta)**
  - **tenta todas as possibilidades** e seleciona a que produz melhor resposta
  - muitas vezes o **custo de tempo é proibitivo**
  - no problema do troco: testar todas as combinações possíveis de moedas, escolher a que resolve com menos moedas

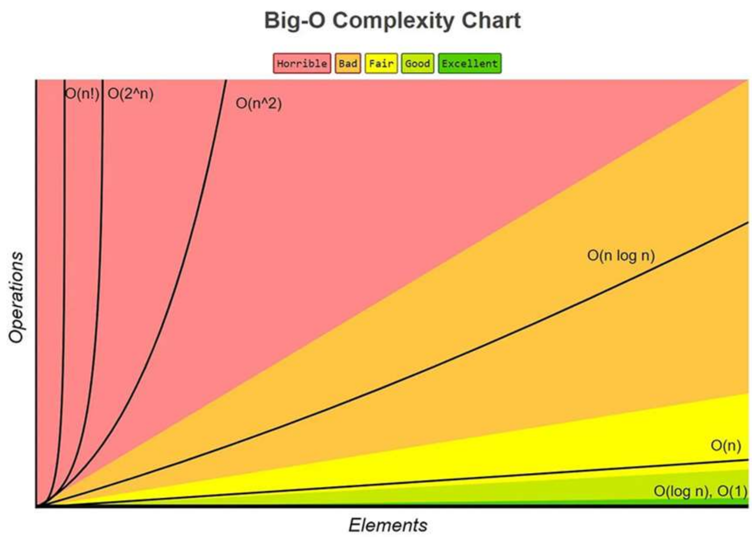
- **Algoritmo Guloso**
  - **toma a melhor decisão local**, na esperança de gerar a melhor globalmente.
  - são eficientes, mas **não garantem otimalidade**
  - no problema do troco: escolher a maior moeda possível a cada moeda e montar o troco dessa forma

Programação Dinâmica

- Como abordar problemas de otimização?
- Programação Dinâmica
  - combina o melhor das duas estratégias
  - nos dá uma forma sistemática de busca todas as possibilidades (**garante otimalidade**)
  - armazena resultados parciais para evitar recálculo (**eficiente**)
  - muitas vezes, torna possível que uma solução exponencial seja otimizada para polinomial

Programação Dinâmica

- torna possível que uma solução exponencial seja otimizada para polinomial

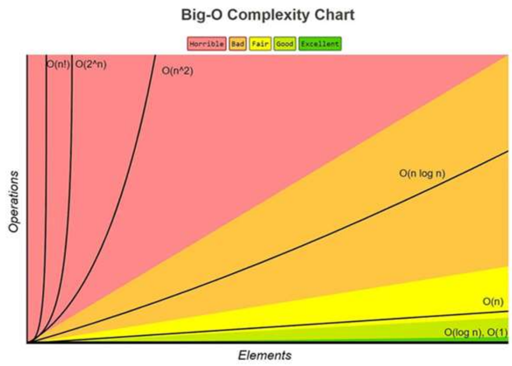


Fonte:
https://www.bigocheatsheet.com/

Programação Dinâmica

- torna possível que uma solução exponencial seja otimizada para polinomial

Solução com PD (Polinomial)	Solução Ingênua (Exponencial)
$O(n^2)$	$O(2^n)$



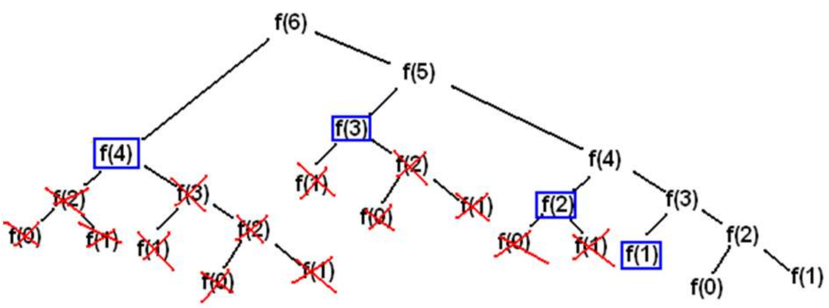
Para n = 100

- $n^2 = 100^2 = 10000$ operações
  $= 10^4$ operações
- $2^n = 2^{100} = 1267650600228229401496703205376$ operações
  $= 1{,}26 * 10^{30}$ operações

Como a PD otimiza as soluções?

É uma técnica eficiente de implementar **algoritmos recursivos** armazenando resultados parciais (memoização!)

- O truque é perceber quando um algoritmo recursivo original calcula a mesma coisa várias vezes
- Nesse caso, armazenar resposta de cada sub-problema. Assim, ao consultar a resposta armazenada, evitamos calcular novamente o que já foi calculado.



Por que estudar Programação Dinâmica?

É uma ferramenta poderosa no arsenal de um programador!

- **Resolução de problemas complexos:** é uma ferramenta poderosa para resolver problemas intratáveis por métodos diretos ou força bruta
- **Redução da complexidade temporal:** capaz de transformar algoritmos exponenciais em algoritmos polinomiais
- **Ampla aplicabilidade:** aplicável em problemas de diversas áreas, como computação, matemática, economia e bioinformática.
- **Base para algoritmos avançados:** desenvolve um pensamento estruturado e exercita vários conceitos fundamentais, usados em algoritmos mais complexos

Introdução à Programação Dinâmica

https://devsuperior.com.br

Prof. Dr. Nelio Alves

Programação Dinâmica

O que é programação dinâmica?

É uma técnica que divide problemas complexos em subproblemas menores, armazena suas soluções e reutiliza essas soluções para resolver o problema original da maneira eficiente.

- Combina a **corretude da força bruta** e a **eficiência do algoritmo guloso**
- Casos de uso comuns:
  1. Encontrar a **solução ótima**
  2. Contar o **número de soluções** para um problema
- Geralmente partirá de uma solução recursiva já existente

Números de Fibonacci

Uma famosa sequência matemática é a sequência de Fibonacci, esta sequência é definida por:

$$fib(1) = fib(2) = 1$$
$$fib(n) = fib(n - 1) + fib(n - 2)$$

Escreva uma função que dado N retorne o **n-ésimo número** de Fibonacci.

n	1	2	3	4	5	6	7	8
fib(n)	1	1	2	3	5	8	13	21

Formulação recursiva do problema

Casos base ⟶ fib(1) = fib(2) = 1
Fórmula recursiva ⟶ fib(n) = fib(n - 1) + fib(n - 2)

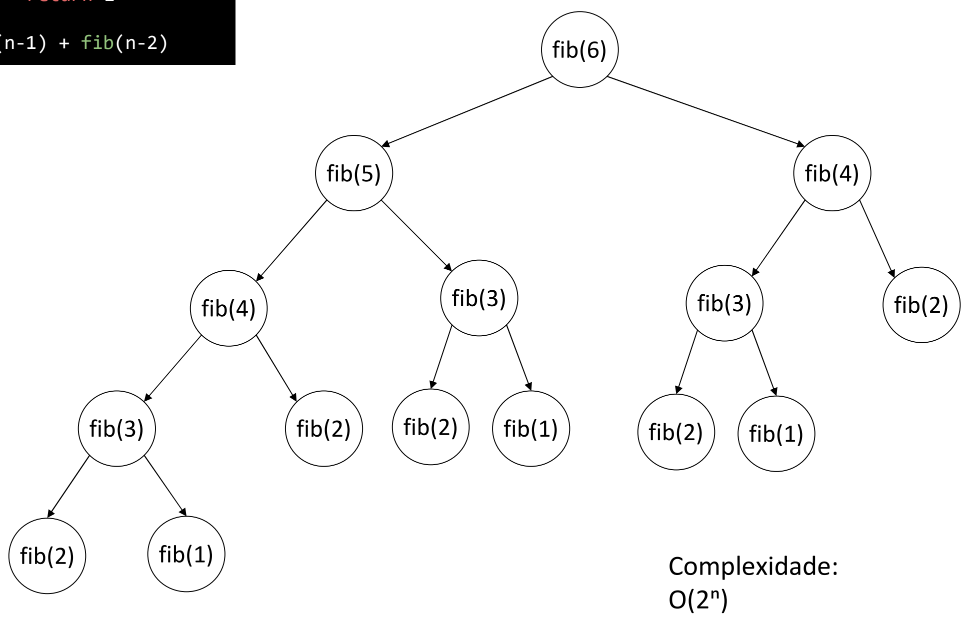
Função fibonacci pseudocódigo

```
fib(n)
        if n <= 2
                return 1

    return fib(n-1) + fib(n-2)
```



```
fib(n)
        if n <= 2
                return 1

    return fib(n-1) + fib(n-2)
```

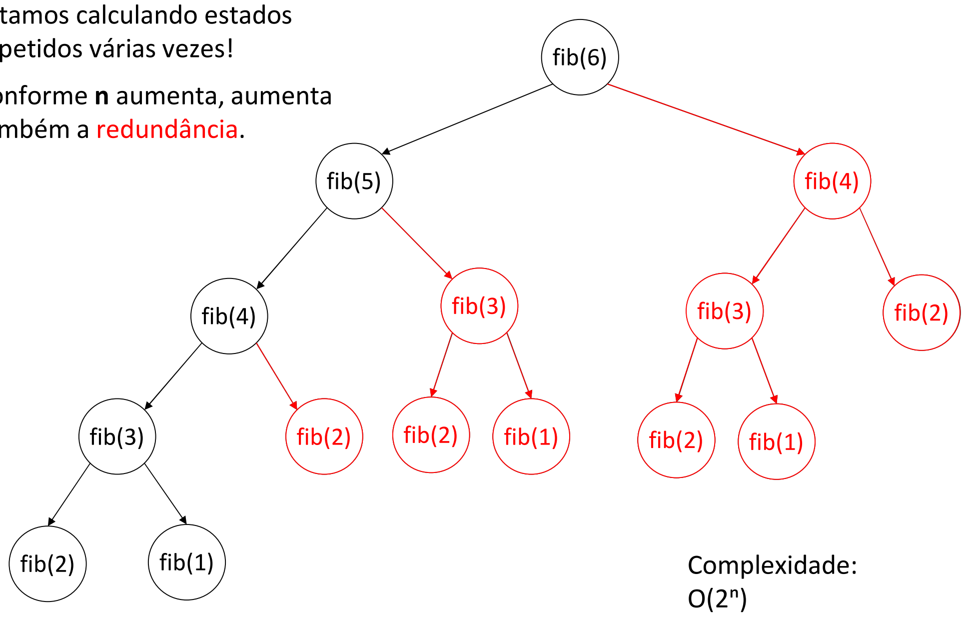
Árvore de recursão para n = 6



Complexidade:
$O(2^n)$

Por que a solução atual é lenta?

- Estamos calculando estados repetidos várias vezes!
- Conforme **n** aumenta, aumenta também a redundância.



Complexidade:
$O(2^n)$

Como aplicar Programação Dinâmica no problema?

Função fibonacci ingênua

```
fib(n)
      if n <= 2
            return 1

   return fib(n-1) + fib(n-2)
```

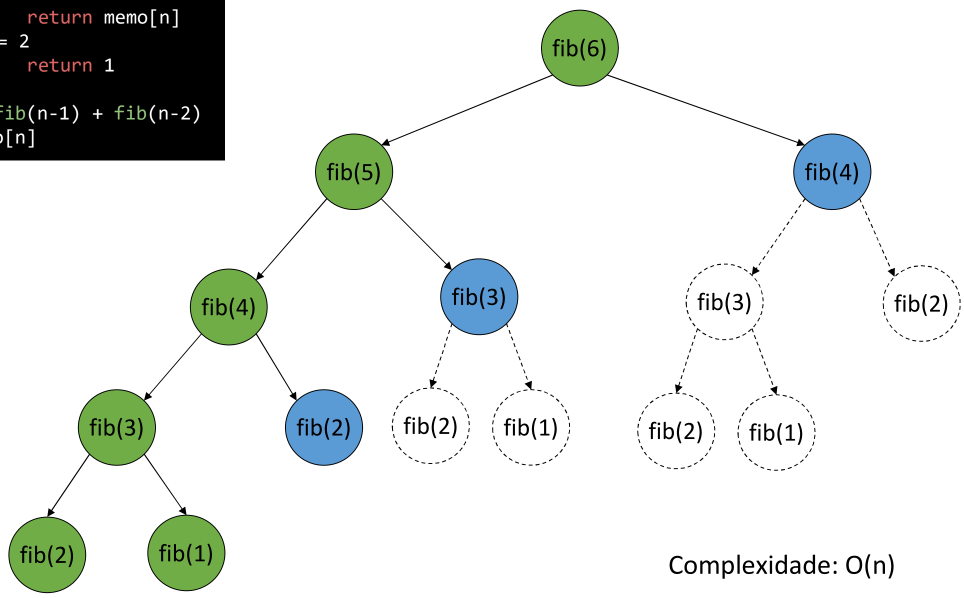
Função fibonacci com PD

```
memo = []
fib(n)
   if memo[n] not null
            return memo[n]
      if n <= 2
            return 1

   memo[n] = fib(n-1) + fib(n-2)
   return memo[n]
```

Recursão + Memoização = Programação Dinâmica

```
memo = []
fib(n)
    if memo[n] not null
            return memo[n]
        if n <= 2
            return 1

    memo[n] = fib(n-1) + fib(n-2)
    return memo[n]
```

Árvore de recursão para n = 6



Complexidade: O(n)

Resolução alternativa para números de Fibonacci

Na solução já vista:

1. Começamos a partir do problema grande
2. Quebramos o problema complexo em subproblemas
3. Resolvemos os subproblemas e os armazenamos
4. Combinamos as soluções para resolver o problema original

Resolvemos o problema “de cima para baixo” (Top-Down)

E se fizermos o caminho contrário?

1. Começar a partir dos subproblemas menores
2. Combinar suas soluções até resolver o problema maior

Resolver o problema “de baixo para cima” (Bottom-Up)

Abordagem Bottom-Up

Se quisermos saber fib(6), podemos começar a partir dos casos base para esse cálculo

n	1	2	3	4	5	6
fib(n)	1	1	?	?	?	?

Abordagem Bottom-Up

Se quisermos saber fib(6), podemos começar a partir dos casos base para esse cálculo

fib(**3**) = fib(1) + fib(2)

n	1	2	**3**	4	5	6
fib(n)	1	1	2	?	?	?

Abordagem Bottom-Up

Se quisermos saber fib(6), podemos começar a partir dos casos base para esse cálculo

fib(**4**) = fib(2) + fib(3)

n	1	2	3	**4**	5	6
fib(n)	1	1	2	3	?	?

Abordagem Bottom-Up

Se quisermos saber fib(6), podemos começar a partir dos casos base para esse cálculo

fib(**5**) = fib(3) + fib(4)

n	1	2	3	4	**5**	6
fib(n)	1	1	2	3	5	?

Abordagem Bottom-Up

Se quisermos saber fib(6), podemos começar a partir dos casos base para esse cálculo

fib(**6**) = fib(4) + fib(5)

n	1	2	3	4	5	**6**
fib(n)	1	1	2	3	5	8

Abordagem Bottom-Up

Se quisermos saber fib(6), podemos começar a partir dos casos base para esse cálculo

n	1	2	3	4	5	**6**
fib(n)	1	1	2	3	5	8

Resposta:

fib(6) = 8

Abordagem Bottom-Up

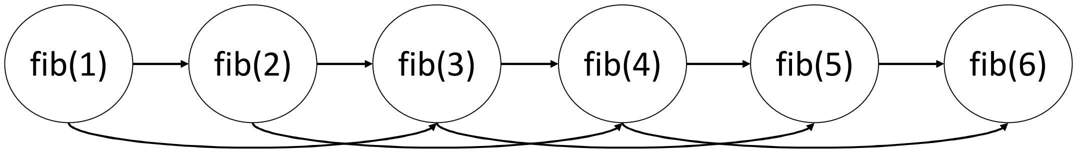
Função fibonacci com PD

```
fib(n)
   memo = []
   memo[1] = memo[2] = 1

   for i=3 to n
      memo[i] = memo[i-1] + memo[i - 2]

   return memo[n]
```

Complexidade: O(n)

Ordem de resolução dos problemas

Para calcular um estado, é necessário que todos os subproblemas dos quais ele depende já tenham sido resolvidos. Ou seja, os problemas devem ser resolvidos de acordo com a sua **ordenação topológica**.



Ordenação topológica: esquema que mostra dependência entre os subproblemas

Abordagens em Programação Dinâmica

Top-Down (Recursão + Memoização)	Bottom-Up (Iteração + Tabulação)
• Vamos "de cima para baixo" • Utilizamos recursão para achar os subproblemas menores • A recursão nos dá a ordem correta de resolução dos subproblemas	• Vamos "de baixo para cima" • Iteramos sobre os subproblemas a partir dos casos base, preenchendo uma tabela • A ordem em que resolvemos os subproblemas importa (ordenação topológica)

Comparando Abordagens em Programação Dinâmica

Top-Down (Recursão + Memoização)	Bottom-Up (Iteração + Tabulação)

• **Mais intuitiva:** geralmente mais fácil de pensar primeiro
• **Desvantagens:**
  ○ Pode ter overhead de chamadas recursivas
  ○ Possibilidade de stack overflow em problemas grandes
	• **Mais eficiente:** melhora desempenho ao evitar recursão
	• **Vantagens:**
	  ○ Elimina overhead das chamadas recursivas
	  ○ Usa menos memória de pilha
	  ○ Geralmente mais rápida

Em geral: começar com abordagem top-down para entender a estrutura do problema, e converter para bottom-up se possível.

Conceitos em Programação Dinâmica

https://devsuperior.com.br

Prof. Dr. Nelio Alves

Programação Dinâmica

É uma técnica de resolução de problemas que quebra um problema complexo em subproblemas menores e mais simples, resolve cada um desses subproblemas uma vez e armazena suas soluções para que possam ser reutilizadas.

Quando podemos utilizar essa técnica?

- **Estrutura Ótima de Subproblemas:** a solução ótima de um problema pode ser composta pelas soluções ótimas de seus subproblemas
- **Sobreposição de Subproblemas:** os subproblemas se repetem várias vezes e podemos armazenar essas soluções para serem reutilizadas

Como esses conceitos aparecem no Fibonacci?

Escreva uma função que dado N retorne o **n-ésimo número** de Fibonacci.

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

- **Estrutura Ótima de Subproblemas:** a solução ótima de fib(n) é composta pelas soluções ótimas de fib(n - 1) e fib(n - 2)
- **Sobreposição de Subproblemas:** os subproblemas se repetem várias vezes e podemos utilizar cálculos anteriores

Como esses conceitos aparecem no Fibonacci?

Escreva uma função que dado N retorne o **n-ésimo número** de Fibonacci.

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

- **Formulação de estado:** o estado definido por n conterá o valor do n-ésimo número de Fibonacci, fib(n)
- **Casos base:** casos triviais para os quais sabemos solução, nesse caso, fib(1) = fib(2) = 1
- **Relação entre os estados:** fib(n) = fib(n - 1) + fib(n - 2)

Próximos passos

Nem sempre saberemos que um problema é resolvido com PD, e raramente teremos a formulação recursiva em mãos. Por isso precisamos aprender a:

1. identificar problemas de DP
2. escolher os subproblemas corretos (estados)
3. relacionar os estados por uma relação de recorrência
4. incluir memória nos nossos programas para reutilizar cálculos
   a. memoização (top-down)
   b. tabulação (bottom-up)

Como fazer isso?

Resolver problemas clássicos para construir uma intuição que vem com a experiência.