

Programação dinâmica

Marcelo Keese Albertini
Universidade Federal de Uberlândia

3 de Julho de 2018

Aula de hoje

Nesta aula veremos:

- Recursividade
- Programação Dinâmica

Recursividade

Definição 1

A definição de recursividade pode ser encontrada na Definição 1 deste slide.

Resolução de problemas em partes

Divisão de um problema maior, em sucessivas partes menores e igualmente estruturadas, até chegar na menor parte de todas, cuja resposta é indivisível e trivial.

Recursividade

Sistemática

- uma função recursiva chama ela mesma, com valores de parâmetros diferentes

Memória

- cada nova chamada de função mais memória para as novas variáveis locais

Exemplo: fatorial

Naturalmente recursivo

fatorial(5) = 5 * fatorial(4)

```
1 int fatorial(n) {  
2   if (n > 1) {  
3     return n*fatorial(n-1);  
4   } else {  
5     return 1;  
6   }  
7 }
```

Em uma solução recursiva definir:

- caso base: solução trivial
- regra de recorrência

Recursão indireta

Quando uma função $f(x)$, chama $g(x)$ que chama $f(x)$, ou seja, $f(g(f(x)))$..

```
1 void f(int a) {
2     if (a < 20) {
3         a++;
4         g(a)
5     }
6     printf("a = %d", a);
7 }
8
9 void g(int b) {
10    if (b < 20) {
11        b = b + 2;
12        f(b)
13    }
14    printf("b = %d", b);
15 }
```

Execução:

```
1 f(10)
2   g(11)
3     f(13)
4       g(14)
5         f(16)
6           g(17)
7             f(19)
8               g(20)
9                 b = 20
10                a = 19
11                b = 17
12                a = 16
13                b = 14
14 ...
```

Criação de coelhos de Fibonacci

- 1 um novo coelho na gaiola

Criação de coelhos de Fibonacci

- 1 um novo coelho na gaiola
- 2 uma nova coelha na gaiola

Criação de coelhos de Fibonacci

- 1 um novo coelho na gaiola
- 2 uma nova coelha na gaiola
- 3 2 coelhos fazem um novo coelho, 3 ao todo

Criação de coelhos de Fibonacci

- 1 um novo coelho na gaiola
- 2 uma nova coelha na gaiola
- 3 2 coelhos fazem um novo coelho, 3 ao todo
- 4 3 coelhos fazem 2 novos coelhos, 5 ao todo

Criação de coelhos de Fibonacci

- 1 um novo coelho na gaiola
- 2 uma nova coelha na gaiola
- 3 2 coelhos fazem um novo coelho, 3 ao todo
- 4 3 coelhos fazem 2 novos coelhos, 5 ao todo
- 5 5 coelhos fazem 3 novos, 8 ..., **geração** $n=5$

Criação de coelhos de Fibonacci

- 1 um novo coelho na gaiola
- 2 uma nova coelha na gaiola
- 3 2 coelhos fazem um novo coelho, 3 ao todo
- 4 3 coelhos fazem 2 novos coelhos, 5 ao todo
- 5 5 coelhos fazem 3 novos, 8 ..., **geração** $n=5$
- 6 8 fazem mais 5, 13... ($t_6 = 8$), $n=6$

Criação de coelhos de Fibonacci

- 1 um novo coelho na gaiola
- 2 uma nova coelha na gaiola
- 3 2 coelhos fazem um novo coelho, 3 ao todo
- 4 3 coelhos fazem 2 novos coelhos, 5 ao todo
- 5 5 coelhos fazem 3 novos, 8 ..., **geração** $n=5$
- 6 8 fazem mais 5, 13... ($t_6 = 8$), $n=6$
- 7 13 mais 8, 21 ($t_7 = 13$), $n=7$

Criação de coelhos de Fibonacci

- 1 um novo coelho na gaiola
- 2 uma nova coelha na gaiola
- 3 2 coelhos fazem um novo coelho, 3 ao todo
- 4 3 coelhos fazem 2 novos coelhos, 5 ao todo
- 5 5 coelhos fazem 3 novos, 8 ..., **geração** $n=5$
- 6 8 fazem mais 5, 13... ($t_6 = 8$), $n=6$
- 7 13 mais 8, 21 ($t_7 = 13$), $n=7$
- 8 $21 = t_7 + t_6$, $n = 8$

Criação de coelhos de Fibonacci

- 1 um novo coelho na gaiola
- 2 uma nova coelha na gaiola
- 3 2 coelhos fazem um novo coelho, 3 ao todo
- 4 3 coelhos fazem 2 novos coelhos, 5 ao todo
- 5 5 coelhos fazem 3 novos, 8 ..., **geração** $n=5$
- 6 8 fazem mais 5, 13... ($t_6 = 8$), $n=6$
- 7 13 mais 8, 21 ($t_7 = 13$), $n=7$
- 8 $21 = t_7 + t_6$, $n = 8$

Criação de coelhos de Fibonacci

- 1 um novo coelho na gaiola
- 2 uma nova coelha na gaiola
- 3 2 coelhos fazem um novo coelho, 3 ao todo
- 4 3 coelhos fazem 2 novos coelhos, 5 ao todo
- 5 5 coelhos fazem 3 novos, 8 ..., **geração** $n=5$
- 6 8 fazem mais 5, 13... ($t_6 = 8$), $n=6$
- 7 13 mais 8, 21 ($t_7 = 13$), $n=7$
- 8 $21 = t_7 + t_6$, $n = 8$

Regra de recorrência

O número de coelhos da geração atual depende do número de coelhos das duas gerações anteriores.

$$t_n = t_{n-1} + t_{n-2}$$

Coelhos

```
1 /* Entrada: numero inteiro representando a geracao
2  *          que queremos saber a quantidade de coelhos
3  *
4  * Saida: numero de coelhos na geracao desejada
5  */
6 int contaCoelhos(int geracao) {
7     if (geracao == 1 || geracao == 2) {
8         return geracao;
9     } else {
10        int coelhosGen1 = contaCoelhos(geracao-1);
11        int coelhosGen2 = contaCoelhos(geracao-2);
12        return coelhosGen1 + coelhosGen2;
13    }
14 }
```

Busca de grupos de amigos

```
1 class Pessoa {  
2     int grupo;  
3     Pessoa [] amigos;  
4 }
```

- Temos uma rede social
- Atribuimos código de grupos para as pessoas amigas
- Como medir o tamanho de grupos?

Grupos de amigos

```
1 /* Método de um tipo "Pessoa".
2 * Entrada: informações disponíveis da própria Pessoa
   this
3 * Saída: número de pessoas que contituem o grupo em que
   a pessoa está conectada
4 */
5 int medirGrupo() {
6     int tam = 0;
7
8     if (this.grupo == -1) {
9         tam = 1;
10        this.grupo = 1;
11
12        for (int i = 0; i < this.amigos.length; i++) {
13            tam = tam + this.amigos[i].medirGrupo();
14        }
15    }
16
17    return(tam);
18 }
```

Recursão

- Alguns problemas são mais fáceis de resolver com recursividade
- É necessário cuidado para evitar uma **pilha de recursão** grande

Definição: **pilha de recursão**

Pilha de recursão é a sequência de chamadas de função para realizar a recursividade. Uma chamada pode envolver o uso de bastante memória.

- O cálculo do número de coelhos de certas gerações é repetido diversas vezes
 - possível evitar o re-cálculo por meio do armazenamento de soluções em tabelas
 - base de Programação Dinâmica

Programação dinâmica: motivação

Recursão Fibonacci: {1, 2, 3, 5, 8, 13}

$$f(6) = f(5) + f(4)$$

$$f(6) = (f(4) + f(3)) + (f(3) + f(2))$$

$$f(6) = ((f(3) + f(2)) + (f(2) + f(1))) + (f(2) + f(1) + 2)$$

$$f(6) = (((f(2) + f(1)) + 2) + (2 + 1)) + (2 + 1 + 2)$$

$$f(6) = (((2 + 1) + 2) + (2 + 1)) + (2 + 1 + 2)$$

$$f(6) = 13$$

Repetição de cálculos

Várias recursões podem ser evitadas: $f(4)$ é refeita 2 vezes, $f(3)$ refeita 3 vezes, $f(2)$ é refeita 5 vezes.

Programação dinâmica: memorização

Princípio

Em um problema de programação dinâmica, as soluções de instâncias dos problemas maiores são compostas a partir de soluções de instâncias menores.

Programação dinâmica

programação no sentido de planejamento (*programar as férias*).
dinâmica porque ocorre em uma sequência.

Como fazer?

Cada vez que uma solução é encontrada, guardar em uma tabela.

Programação dinâmica 1: Fibonacci

```
1 int [] sols = new int [0];
2 int fibonacci(int n) {
3     if (n<=1) return 1;
4     if (n==2) return 2;
5     if (sols.length <= n) {
6         sols = new int [2*n+1];
7         sols [0] = sols [1] = 1 ;
8     }
9     return fibrec(n);
10 }
11 int fibrec(int n) {
12     if (n>0) {
13         if (sols [n] > 0) return(sols [n]); // usa memória
14         else {
15             sols [n-1] = fibrec(n-1); // guarda
16             sols [n-2] = fibrec(n-2); // na memória
17             return (sols [n-1]+sols [n-2]);
18         }
19     } else return 1;
20 }
```

Programação dinâmica 2: Fibonacci

```
1 int fibonacci(int n) {
2     int [] f = new int[n];
3
4     // Condições iniciais
5     f[0] = 1;
6     f[1] = 2;
7
8     // Programação dinâmica
9     for (int i = 2; i < n; i++) {
10         f[i] = f[i-1] + f[i-2];
11     }
12
13     return f[n-1];
14 }
```


Programação dinâmica: eficiência

Estratégia será eficiente se

número de subproblemas resolvidos (ou seja, não consultados da tabela) for pequeno.

Pequeno se o tamanho da instância for n e o número de subproblemas resolvidos for uma função polinomial de n , com grau até 3.

Uso de espaço

Temos que armazenar soluções passadas

Programação dinâmica: problemas

Cada problema pode ter uma formulação diferente para a solução com programação dinâmica

Alguns problemas

- Multiplicação de cadeia de matrizes
- Problema da mochila
- Distância de edição entre strings
- Máxima subcadeia comum

Programação dinâmica (Cormen et al.)

- Subestrutura ótima: a solução ótima para o problema contém suas **subsoluções** i.e., soluções ótimas para os subproblemas
- Subsoluções sobrepostas: essas subsoluções podem ser recomputadas repetidamente quando o algoritmo é de força-bruta

Usando Programação Dinâmica

- Passo 1: **Estudar problema** e identificar subproblemas
- Passo 2: **Montar recorrência** que usa a solução pronta dos subproblemas
- Passo 3: **Escrever algoritmos**
 - Versão 1: **Recursão top-down**
 - Versão 2: **Recursão top-down com memos**
 - Versão 3: **Procedimento** não-recursivo **bottom-up** com memos
- Passo 4: **Análise** de custo

Análise de algoritmos de PD

- Análise:
 - Contar número de subproblemas: N
 - Custo por subproblema: M é custo de combinar soluções em tabela na solução do subproblema atual
 - Custo total: $M \times N$
- Se subproblemas puderem ser organizados sequencialmente:
 - Se subproblemas são **sufixos** a partir da i -ésima parte, então são $\Theta(N)$ subproblemas
 - Se subproblemas são **prefixos** antes da j -ésima parte, então são $\Theta(N)$ subproblemas
 - Se subproblemas são **substrings** entre i e j , então são $\Theta(N^2)$ subproblemas

Exemplo: problema do troco – Passo 1: estudar problema

De quantas maneiras diferentes posso dar um troco de valor N ?

- Entrada: troco esperado N e valores de moedas disponíveis $S[]$
- Saída: número de maneiras diferentes de formar o troco pedido.

Ideia

- Dividir número de soluções em 2 partes
 - Contar número de soluções **com** a moeda i
 - Contar número de soluções **sem** a moeda i

Exemplo: problema do troco – Passo 2: montar recorrência

Recorrência

A função $\text{contar}(S, m, N)$ obtém o número de maneiras de formar o troco N com as m primeiras moedas disponíveis em S .

- Se usar a moeda atual, então a sub-contagem será $\text{contar}(S, m, N - S[\text{atual}])$
- Se não usar a moeda atual, então a sub-contagem será $\text{contar}(S, m - 1, N)$
- Portanto a contagem deve ser

$$\text{contar}(S, m, N) = \text{contar}(S, m, N - m[\text{atual}]) + \text{contar}(S, m - 1, N)$$

Exemplo: troco – Passo 3: recursão top-down

```
1 /*
2 Entrada: S: array com valores de moedas em centavos ,
3           m: número de valores de moedas disponíveis ,
4           N: valor em centavos do troco a ser formado.
5
6 Saída: número de maneiras de formar o troco .
7 */
8 int contar(int [] S, int m, int N){
9     if (N == 0) // troco foi formado
10        return 1;
11     if (N < 0) // não é possível usar essas moedas
12        return 0;
13     if (m <= 0 && N >= 1) // acabaram os valores
14        return 0;
15
16     return contar(S, m-1, N) + contar(S, m, N-S[m-1]);
17 }
```


Exemplo: troco – Passo 3: recursão com memos

```
1 /* Entrada: S: array com valores de moedas em centavos ,
2           m: número de valores de moedas disponíveis ,
3           N: valor em centavos do troco a ser formado.
4           memo: new int[N+1][m];
5 Saída: número de maneiras de formar o troco.*/
6 int contar(int [] S, int m, int N){
7     if (N == 0) // troco foi formado
8         return 1;
9     if (N < 0) // não é possível usar essas moedas
10        return 0;
11    if (m <= 0 && N >= 1) // acabaram os valores
12        return 0;
13    if (memo[N][m] == 0)
14        memo[N][m] = contar(S, m-1, N)
15                        + contar(S, m, N-S[m-1]);
16    return memo[N][m];
17 }
```

Exemplo: troco – Passo 3: bottom-up

```
1 int contar(int S[], int m, int N) {
2     int memo[N+1][m]; // tabela de subproblemas
3     // inicializar para troco formado: n = 0
4     for (int i=0; i < m; i++) memo[0][i] = 1;
5     for (valor = 1; valor < N+1; valor++) {
6         for (int moeda_i = 0; moeda < m; moeda++) {
7             int com_i = 0; // contar incluindo moeda i
8             if (valor - S[moeda_i] >= 0)
9                 com_i = memo[valor - S[moeda_i]][moeda_i];
10
11             int sem_i = 0; // contar excluindo moeda_i
12             if (moeda_i >= 1) sem_i = memo[valor][moeda_i - 1];
13
14             memo[i][j] = com_i + sem_i;
15         }
16     }
17     return memo[N][m-1];
18 }
```

Exemplo: troco – Passo 4: análise

- Loop inicialização (linha 4): m memos
- Loops resolução (linhas 5 e 6): $N \times m$ memos (linha 14)
- Se $m = \Theta(N)$, então custo total é $\Theta(N^2)$

Exemplo: corte de barras – Passo 1: estudar problema

- Como achar sequência de cortes de uma barra para maximizar lucro

tam(cm)	1	2	3	4	5	6	7	8	9
valor P_i	1	5	8	9	10	17	17	20	24

- P_i é preço do corte do tamanho i e P_N da barra inteira
- Cortes $N = C_1 + C_2 + \dots + C_K$ correspondem ao lucro

$$L_N = P_{C_1} + P_{C_2} + \dots + P_{C_K}$$

- Objetivo é maximizar L_N

Exemplo: corte de barras – Passo 2: montar recursão

- Objetivo é maximizar

$$L_N = \max(P_N, L_1 + L_{N-1}, L_2 + L_{N-2}, \dots, L_{N-1} + L_1)$$

- Recorrência usando sequência de subproblemas do tipo sufixo
 - $L_i + L_{N-i}$ é lucro de fazer venda em 2 partes independentes
- Recorrência é:

$$L_N = \max_{1 \leq i \leq N} (P_i + L_{N-i})$$

Exemplo: corte de barras – Passo 3: recursão

```
1 corte(P, N)
2   if (N == 0) return 0
3   L = - inf
4   for (i in 1...N) {
5     L = max(L, P[i]+corte(p, N-i))
6   }
```

Exemplo: corte de barras – Passo 3: Recursão com memos

```
1 corte(P, N)
2   if (memo[N] >= 0) return memo[N]
3   if (N==0) L = 0;
4   else {
5       L = -inf
6       for (i in 1:N){
7           L = max(L, P[i]+corte(P,N-i ,memo));
8       }
9       memo[N] = L;
10  }
11  return L;
12 }
```

Exemplo: corte de barras – Passo 3: bottom-up

```
1 corte(P, N)
2     memo[0 ... N] = -inf
3     memo[0] = 0
4     for (j = 1; j <= N; j++) { // número de
5         // subproblemas
6         L = -inf
7         for (i = 1; i <= j; i++) { // resolver subproblema
8             L = max(L, P[i] + memo[j-i])
9         }
10        memo[j] = L
11    }
12    return memo[N]
13 }
```


Exemplo: corte de barras – Passo 4: Análise

- Como sequência de subproblemas é usando **sufixos**, tem-se $\Theta(N)$ subproblemas
- A resolução de um subproblema (linha 6) usa $\Theta(N)$ consultas a subproblemas anteriores
- Custo total: $\Theta(N^2)$
- Mais precisamente: $\sum_{j=1}^N j$

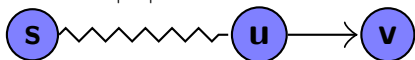
Problema: Justificação de textos

$$\text{badness}(i, j) = (\text{tam. pág} - \text{tam. da linha usando words}[i : j])^3$$

- $\text{words}[i : j]$ indica linha usando palavras de i a j
- 1) definir subproblemas: sufixo do texto atual quebrado antes de i : $\text{words}[i :]$
- 2) opções: lugares onde quebrar as palavras words para começar novas linha
 - “se temos n palavras e quebramos na i -ésima” são $n - i$ opções, no máximo
- 3) Recorrência: $J(i) = \min \{J(j) + \text{badness}(i, j)\}$ para $\forall j \in [i + 1, n + 1]$
- 4) Solução do problema original: usar ponteiros-pai para “lembrar da melhor escolha” para construir a solução:
 - $\text{pai}[i] = \arg \min(\dots) = j$
 - $0 \rightarrow \text{pai}[0] \rightarrow \text{pai}[\text{pai}[0]] \rightarrow \dots$
 - Reconstrução da solução em tempo linear

Problema: Menor Caminho

- Problema: menor caminho entre s e t
- $\delta_k(s, v)$, $s, v \in V$, usando $0 \leq k < |V|$ passos
- Subproblema: achar menor caminho entre qualquer s e v usando k arestas“
- Número de subproblemas: $|V|^2$ usando todo $v \in V$ e $0 \leq k < |V|$



onde u pode ser cada um dos vértices chegando a v

- Montar Recursão

$$\delta_k(s, v) = \min \{ \delta_{k-1}(s, u) \mid (u, v) \in E \} + \text{peso}(u, v)$$

- Extrair resposta usando “ponteiros-pai” para indicar de onde veio
- Análise: número de opções para cada recursão é $\sum_{v \in V} \text{indegree}(v) = 2|E|$
- Resolução por sufixos em u : $\Theta(N)$ problemas

Problema: Parentização de Matrizes

- Encontrar melhor ordem para multiplicar $A_0 \times A_1 \times A_2 \times \dots \times A_{N-1}$
- Escolha: qual é a última multiplicação

$$(A_0 \dots A_{k-1}) \times (A_k \dots A_{N-1})$$

- Subproblema: parentização de $A_i \dots A_{j-1}$
- Número de subproblemas é $\Theta(N^2)$
- Número de escolhas da multiplicação:
 $(A_i \dots A_{k-1}) \times (A_k \dots A_{j-1}), j - i + 1 = O(N)$
- $DP(i, j) = \min_{k \in [i+1, j]} (DP(i, k) + DP(k, j) + \text{custo de } A_{1:k} \times A_{k:j})$
 - custo de $\Theta(N)$ para resolver cada subproblema
- Custo total é $\Theta(N) \times \Theta(N^2) = \Theta(N^3)$
- Ordem de resolução: crescendo substring

Problema: distância de edição

- A distância de edição entre strings s_1 e s_2 é o número mínimo de operações básicas para transformar s_1 em s_2
- Distância de Levenshtein: operações básicas são **inserção**, **remoção** e **substituição**
- distância de *doa-do*: 1
- distância de *caro-carro*: 1
- distância de *pato-pata*: 1
- distância de *vôa-avô*: 2
- Resolução PD: subproblemas do tipo sufixos na dimensão de s_1 e em s_2

Distância de Levenshtein

distanciaLevenshtein(s_1, s_2)

```
1  for  $i \leftarrow 0$  to  $|s_1|$ 
2  do  $m[i, 0] = i$ 
3  for  $j \leftarrow 0$  to  $|s_2|$ 
4  do  $m[0, j] = j$ 
5  for  $i \leftarrow 1$  to  $|s_1|$ 
6  do for  $j \leftarrow 1$  to  $|s_2|$ 
7      do if  $s_1[i] = s_2[j]$ 
8          then  $m[i, j] = \min\{m[i-1, j]+1, m[i, j-1]+1, m[i-1, j-1]\}$ 
9          else  $m[i, j] = \min\{m[i-1, j]+1, m[i, j-1]+1, m[i-1, j-1]+1\}$ 
10 return  $m[|s_1|, |s_2|]$ 
```

Operações: inserir (custo 1), remoção (custo 1), substituir (custo 1), cópia (custo 0)

Distância de Levenshtein

		f	a	s	t
	<u> </u> 0	<u> 1 </u> 1 1	<u> 2 </u> 2 2	<u> 3 </u> 3 3	<u> 4 </u> 4 4
c	<u> 1 </u> 1	<u> 1 2 </u> 2 1	<u> 2 3 </u> 2 2	<u> 3 4 </u> 3 3	<u> 4 5 </u> 4 4
a	<u> 2 </u> 2	<u> 2 2 </u> 3 2	<u> 1 3 </u> 3 1	<u> 3 4 </u> 2 2	<u> 4 5 </u> 3 3
t	<u> 3 </u> 3	<u> 3 3 </u> 4 3	<u> 3 2 </u> 4 2	<u> 2 3 </u> 3 2	<u> 2 4 </u> 3 2
s	<u> 4 </u> 4	<u> 4 4 </u> 5 4	<u> 4 3 </u> 5 3	<u> 2 3 </u> 4 2	<u> 3 3 </u> 3 3

Cada célula da matriz de Levenshtein

custo de chegar aqui a partir do vizinho superior esquerdo (cópia ou substituição)	custo de chegar aqui a partir do vizinho superior (remoção)
custo de chegar aqui a partir do vizinho da esquerda (inserção)	o mínimo entre os três possíveis “movimentos”: o jeito mais barato de chegar aqui

Exercícios

- <http://br.spoj.com/problems/MOEDAS/>
- <https://br.spoj.com/problems/PARQUE/>
- <http://br.spoj.com/problems/TESOURO>
- <https://br.spoj.com/problems/DESCULPA/>
- <https://br.spoj.com/problems/GENEAL/>
- <https://br.spoj.com/problems/MINIMO/>
- <https://br.spoj.com/problems/NUVEMMG/>