Programação Dinâmica pt. 2

André Fakhoury
SCC0210 - Lab. Algoritmos Avançados 1

Relembrando...

Programação Dinâmica

- Estados e transições
- Calcular um estado atual a partir de outros
- **Memoização** para que não seja necessário calcular novamente

Ex: Fibonacci

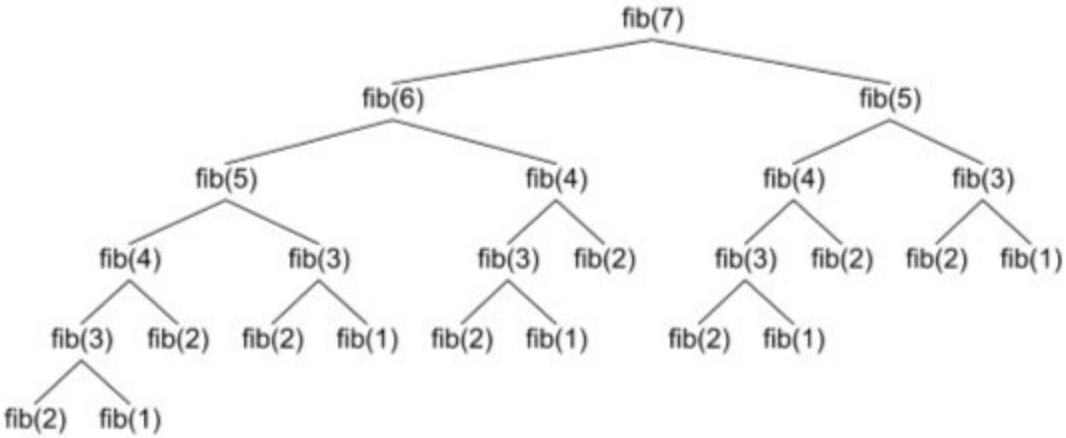
Recursivo

```
int f(int n) {
  if (n <= 1) return n;
  return f(n-1) + f(n-2);
}
```

Ex: Fibonacci

Recursivo

```
int f(int n) {
  if (n <= 1) return n;
  return f(n-1) + f(n-2);
}
```



Fonte: [1]

Ex: Fibonacci

Recursivo

- Calculando algumas f(n) várias vezes
- Complexidade exponencial ~ O(2n)
- Ideia: armazenar valores já computados para re-aproveitar
- **Memoização**

Ex: Fibonacci

Recursivo + Memoização

```
const int MAXN = 1e5 + 5;
int f_memo[MAXN];

int f(int n) {
  if (n <= 1) return n;

  if (f_memo[n] == 0) { // ainda não calculado
     f_memo[n] = f(n-1) + f(n-2);
  }

  return f_memo[n];
}
```

Ex: Fibonacci

Recursivo + Memoização

```
const int MAXN = 1e5 + 5;
int f_memo[MAXN];

int f(int n) {
  if (n <= 1) return n;

  if (f_memo[n] == 0) { // ainda não calculado
     f_memo[n] = f(n-1) + f(n-2);
  }

  return f_memo[n];
}
```

Cada f(n) será calculado apenas uma vez em O(1)

Complexidade linear: O(n)

Ex: Fibonacci

Recursivo + Memoização

- Chamamos essa abordagem de **top-down** ou **recursiva**
  - Começa de um valor n qualquer até o caso base
  - Basicamente recursão + memoização

Ex: Fibonacci

Recursivo + Memoização

- Chamamos essa abordagem de **top-down** ou **recursiva**
  - Começa de um valor n qualquer até o caso base
  - Basicamente recursão + memoização
- Também podemos fazer de forma **bottom-up** ou **iterativa**
  - Começa dos casos base até um n qualquer

Ex: Fibonacci

Iterativo

```
const int MAXN = 1e5 + 5;
int f_memo[MAXN];

void calc_fibonacci(int N) {
  f_memo[0] = 0;
  f_memo[1] = 1;

  for (int n = 2; n <= N; n++) {
     f_memo[n] = f_memo[n-1] + f_memo[n-2];
  }
}
```

Longest Increasing Subsequence - LIS

“Dado um vetor de inteiros, calcular a maior subsequência de valores estritamente crescentes”

- Subsequência: formada a partir de uma sequência eliminando alguns elementos e mantendo sua ordem
- **Exemplos:**
  - [2, 4] é subsequência de [1, 2, 3, 4, 5]
  - [2, 1] **não** é subsequência de [1, 2, 3, 4, 5]

Longest Increasing Subsequence - LIS

Exemplo:

a = [10, 9, 2, 5, 3, 7, 101]

Longest Increasing Subsequence - LIS

Exemplo:

a = [10, 9, 2, 5, 3, 7, 101]

DP[i] será a resposta atual para [i..n] (ou seja, ignorando as posições anteriores)

Quais serão as transições?

Longest Increasing Subsequence - LIS

Exemplo:

a = [10, 9, 2, 5, 3, 7, 101]

DP[i] será a resposta atual para [i..n] (ou seja, ignorando as posições anteriores)

Quais serão as transições?

Considerando que *i* seja o primeiro elemento da resposta. Qual pode ser o segundo?

Longest Increasing Subsequence - LIS

Exemplo:

a = [10, 9, 2, 5, 3, 7, 101]

DP[i] será a resposta atual para [i..n] (ou seja, ignorando as posições anteriores)

Quais serão as transições?

Considerando que *i* seja o primeiro elemento da resposta. Qual pode ser o segundo?

Todos os elementos posteriores, e que sejam maiores que i

Longest Increasing Subsequence - LIS

Exemplo:

a = [10, 9, 2, 5, 3, 7, 101]

DP[i] será a resposta atual para [i..n] (ou seja, ignorando as posições anteriores)

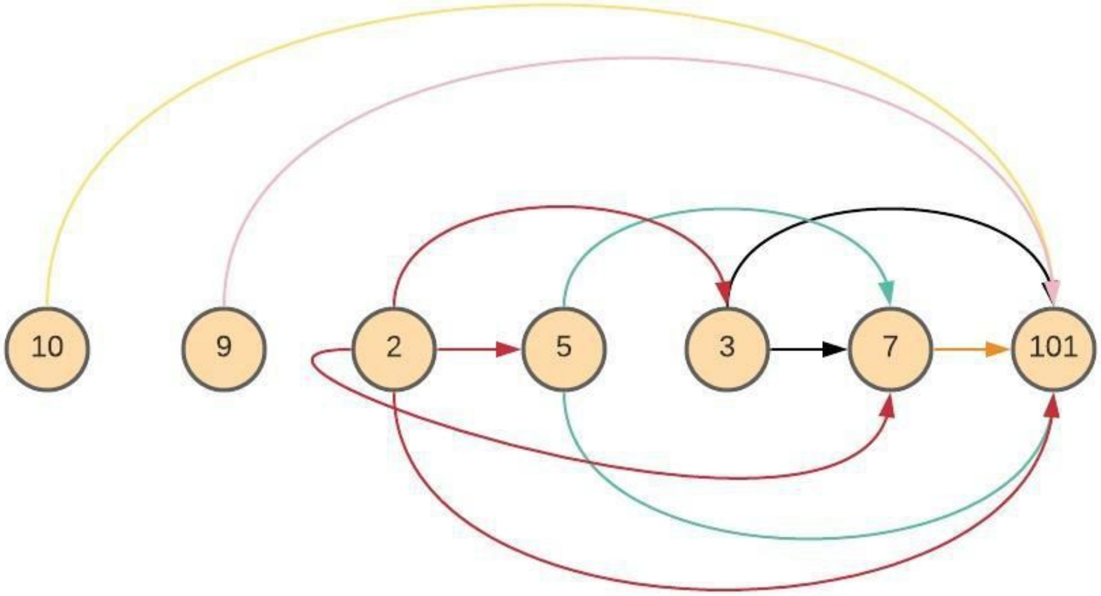
Quais serão as transições?

Considerando que *i* seja o primeiro elemento da resposta. Qual pode ser o segundo?

Todos os elementos posteriores, e que sejam maiores que i

Então basta pegar o elemento que possui a maior resposta!

Longest Increasing Subsequence - LIS



Longest Increasing Subsequence - LIS

```
a        = [10, 9, 2, 5, 3, 7, 101]
indices      0   1   2   3   4   5    6
```

Longest Increasing Subsequence - LIS

```
     a    = [10, 9, 2, 5, 3, 7, 101]
indices      0   1   2  3  4  5   6
```

```
dp[6] = max{1}
```

Longest Increasing Subsequence - LIS

```
a       = [10, 9, 2, 5, 3, 7, 101]
indices     0   1   2  3  4  5    6
```

```
dp[6] = max{1}
dp[5] = max{1, 1 + dp[6]}
```

Longest Increasing Subsequence - LIS

```
a     = [10, 9, 2, 5, 3, 7, 101]
indices     0   1    2   3   4   5     6
```

```
dp[6] = max{1}
dp[5] = max{1, 1 + dp[6]}
dp[4] = max{1, 1 + dp[5], 1 + dp[6]}
```

Longest Increasing Subsequence - LIS

```
        a    = [10, 9, 2, 5, 3, 7, 101]
     indices     0    1   2   3   4   5    6
```

```
dp[6] = max{1}
dp[5] = max{1, 1 + dp[6]}
dp[4] = max{1, 1 + dp[5], 1 + dp[6]}
dp[3] = max{1, 1 + dp[5], 1 + dp[6]}
```

Longest Increasing Subsequence - LIS

```
a   = [10, 9, 2, 5, 3, 7, 101]
indices   0  1  2  3  4  5   6
```

```
dp[6] = max{1}
dp[5] = max{1, 1 + dp[6]}
dp[4] = max{1, 1 + dp[5], 1 + dp[6]}
dp[3] = max{1, 1 + dp[5], 1 + dp[6]}
dp[2] = max{1, 1 + dp[3], 1 + dp[4], 1 + dp[5], 1 + dp[6]}
```

Longest Increasing Subsequence - LIS

```
a       = [10, 9, 2, 5, 3, 7, 101]
indices    0   1   2   3   4   5    6
```

```
dp[6] = max{1}
dp[5] = max{1, 1 + dp[6]}
dp[4] = max{1, 1 + dp[5], 1 + dp[6]}
dp[3] = max{1, 1 + dp[5], 1 + dp[6]}
dp[2] = max{1, 1 + dp[3], 1 + dp[4], 1 + dp[5], 1 + dp[6]}
dp[1] = max{1, 1 + dp[6]}
```

Longest Increasing Subsequence - LIS

```
        a    = [10, 9, 2, 5, 3, 7, 101]
  indices       0   1   2  3  4  5   6
```

```
dp[6] = max{1}
dp[5] = max{1, 1 + dp[6]}
dp[4] = max{1, 1 + dp[5], 1 + dp[6]}
dp[3] = max{1, 1 + dp[5], 1 + dp[6]}
dp[2] = max{1, 1 + dp[3], 1 + dp[4], 1 + dp[5], 1 + dp[6]}
dp[1] = max{1, 1 + dp[6]}
dp[0] = max{1, 1 + dp[6]}
```

Longest Increasing Subsequence - LIS

```
dp[i] = max{  1,
              1 + dp[j], ∀ j > i e a[j] > a[i]
            }
```

```
dp[i] representa a LIS do subarray [i..n] começando em i
```

Longest Increasing Subsequence - LIS

```
const int MAXN = 1e5 + 5;
int a[MAXN], n;
int memo[MAXN];

void lis(int i) {
  if (memo[i] != 0)
    return memo[i];

  memo[i] = 1;
  for (int j = i + 1; j < n; j++)
    if (a[j] > a[i])
      memo[i] = max(memo[i], lis(j) + 1);
  return memo[i];
}
```

Problema do troco - Coin Change

- Dado um valor e um conjunto de moedas, retornar o mínimo de moedas necessárias para formar o valor

Problema do troco - Coin Change

- Dado um valor e um conjunto de moedas, retornar o mínimo de moedas necessárias para formar o valor

Exemplo: quantas moedas de real preciso para somar 16 centavos?

Problema do troco - Coin Change

- Dado um valor e um conjunto de moedas, retornar o mínimo de moedas necessárias para formar o valor
- Exemplo: `valor = 16, moedas = {1, 5, 10, 25, 50, 100}`

Problema do troco - Coin Change

- Dado um valor e um conjunto de moedas, retornar o mínimo de moedas necessárias para formar o valor
- Exemplo: `valor = 16, moedas = {1, 5, 10, 25, 50, 100}`
    - `{1, 5, 10}, 3 moedas`

Problema do troco - Coin Change

- Dado um valor e um conjunto de moedas, retornar o mínimo de moedas necessárias para formar o valor
- Exemplo: `valor = 16, moedas = {1, 5, 10, 25, 50, 100}`
  - `{1, 5, 10}, 3 moedas`
- Exemplo: `valor = 45, moedas = {1, 5, 10, 25, 50, 100}`

Problema do troco - Coin Change

- Dado um valor e um conjunto de moedas, retornar o mínimo de moedas necessárias para formar o valor
- Exemplo: `valor = 16, moedas = {1, 5, 10, 25, 50, 100}`
  - `{1, 5, 10}, 3 moedas`
- Exemplo: `valor = 45, moedas = {1, 5, 10, 25, 50, 100}`
  - `{10, 10, 25}, 3 moedas`

Problema do troco - Coin Change

- Dado um valor e um conjunto de moedas, retornar o mínimo de moedas necessárias para formar o valor
- Exemplo: `valor = 16, moedas = {1, 5, 10, 25, 50, 100}`
  - `{1, 5, 10}, 3 moedas`
- Exemplo: `valor = 45, moedas = {1, 5, 10, 25, 50, 100}`
  - `{10, 10, 25}, 3 moedas`
- Para esse conjunto de moedas, basta ir pegando a maior moeda possível -> solução **gulosa** (ou *greedy*)

Problema do troco - Coin Change

- Será que essa solução sempre funciona?

Problema do troco - Coin Change

- Será que essa solução sempre funciona?
- Exemplo: `valor = 6, moedas = {1, 3, 4}`

Problema do troco - Coin Change

- Será que essa solução sempre funciona?
- Exemplo: `valor = 6, moedas = {1, 3, 4}`
  - `{1, 1, 4}` *(guloso)*
  - `{3, 3}` *(ótima)*

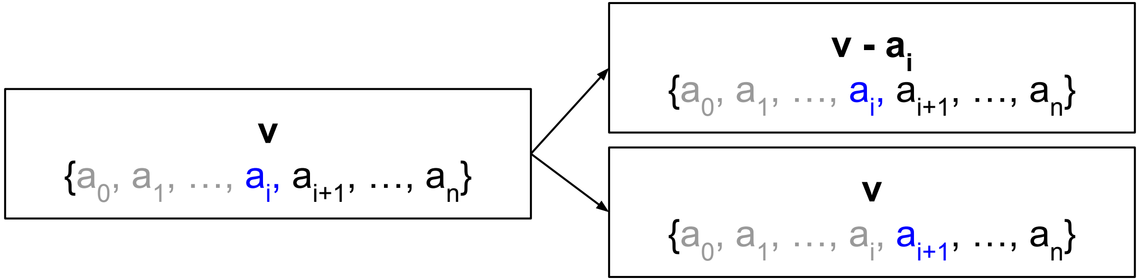
Problema do troco - Coin Change

- Será que essa solução sempre funciona?
- Exemplo: `valor = 6, moedas = {1, 3, 4}`
    - `{1, 1, 4}` *(guloso)*
    - `{3, 3}` *(ótima)*
- Nem sempre, mas podemos utilizar DP para resolver :)
    - Primeiro passo: encontrar uma solução recursiva que funcione
    - Segundo passo: memoizar
    - Alternativamente, dá pra tentar pensar na solução iterativa direto

Problema do troco - Coin Change

Tente usar, para o valor *v* atual, a moeda de posição *i*. Terão duas opções:

- Utilizar a moeda *i*, e decrementar o valor v (v - moeda[i]);
- Não utilizar a moeda *i*, e ir para a próxima moeda (*i + 1*)



Problema do troco - Coin Change

```
const int INF = 1e9;
int n_moedas, moedas[MAXN];

int solve(int at, int valor) {
  if (valor == 0) return 0; // pagou tudo
  if (at == n) return INF;  // acabaram as moedas, impossível continuar

  int ret = INF;

  // utiliza a moeda atual
  if (valor >= moedas[at]) ret = min(ret, 1 + solve(at, valor - moedas[at]));

  // não utiliza a moeda atual
  ret = min(ret, solve(at + 1, valor));

  return ret;
}
```

Problema do troco - Coin Change

```
const int INF = 1e9;
int n_moedas, moedas[MAXN];
int memo[MAXN][MAX_VAL];

int solve(int at, int valor) {
  if (valor == 0) return 0; // pagou tudo
  if (at == n) return INF;  // acabaram as moedas, impossível continuar
  if (memo[at][valor] != -1) return memo[at][valor]; // calculado

  int ret = INF;

  // utiliza a moeda atual
  if (valor >= moedas[at]) ret = min(ret, 1 + solve(at, valor - moedas[at]));

  // não utiliza a moeda atual
  ret = min(ret, solve(at + 1, valor));

  memo[at][valor] = ret;

  return ret;
}
```

Problema do troco - Coin Change

- Complexidade: O(N_Moedas * Valor_Max)
- Importante inicializar os valores do vetor de memoização
    - Valor que nunca será resposta
    - Normalmente, -1 dá conta
    - `memset(memo, -1, sizeof memo);` (cuidado ao usar, memset é byte a byte!)

Problema do troco - Coin Change

```
void calcula_troco(int valor_max) {
  for (int i = 0; i < n; i++) // caso base valor == 0
     memo[i][0] = 0;
  for (int i = 1; i <= valor_max; i++) // caso base at == n
     memo[n][i] = INF;

  for (int at = n - 1; at >= 0; at--) {
     for (int valor = 0; valor <= valor_max; valor++) {
        memo[at][valor] = memo[at + 1][valor];
        if (valor >= moedas[at])
           memo[at][valor] = min(memo[at][valor], 1 + memo[at][valor - moedas[at]]);
     }
  }
}
```

Problema da mochila - Knapsack

Existem n itens, enumerados de 1 a n. Cada item i possui peso w_i e valor v_i.
Qual o valor máximo de itens que podem ser carregados em uma mochila de peso P?

Problema da mochila - Knapsack

Existem *n* itens, enumerados de 1 a n. Cada item *i* possui peso w_i e valor v_i.
Qual o valor máximo de itens que podem ser carregados em uma mochila de peso *P*?

Exemplo: w = 8

```
1. w = 1, v = 2
```

```
2. w = 3, v = 5
```

```
3. w = 4, v = 5
```

```
4. w = 5, v = 6
```

Problema da mochila - Knapsack

Existem *n* itens, enumerados de 1 a n. Cada item *i* possui peso w_i e valor v_i.
Qual o valor máximo de itens que podem ser carregados em uma mochila de peso *P*?

Exemplo: w = 8

```
1. w = 1, v = 2
```

```
2. w = 3, v = 5
```

```
3. w = 4, v = 5
```

```
4. w = 5, v = 6
```

Se pegar {1, 2}: Peso = 4, Valor = 7

Se pegar {2, 4}: Peso = 8, Valor = 11

Se pegar {1, 2, 3}: Peso = 8, Valor = 12

Se pegar {1, 2, 3, 4}: Peso = 13, Valor = 18

Problema da mochila - Knapsack

Cada estado armazena o item *i* a ser considerado e o peso restante na mochila. As opções são: pegar o item *i* (caso couber), ou não pegar este item, e tentar pegar o item *i+1*.

$$f(i, P) = \max \begin{cases} f(i+1, P), & \text{(não pega)} \\ v_i + f(i+1, P - w_i), \text{ se } P >= w_i & \text{(pega)} \end{cases}$$

Problema da mochila - Knapsack

```
int solve(int i, int P) {
  if (i == n) return 0;  // não dá mais pra pegar nada

  int ret = 0;

  // Pega o item atual
  if (P >= w[i])
    ret = max(ret, v[i] + solve(i + 1, P - w[i]));
  ret = max(ret, solve(i + 1, P));

  return ret;
}
```

Longest Common Subsequence - LCS

- Subsequência: formada a partir de uma sequência eliminando alguns elementos e mantendo sua ordem
  - axy é subsequência de zabxccyk
- LCS(a, b) = maior subsequência comum entre as string a e b
  - LCS(*abcd*, *aebe*) = ab
  - LCS(*abc*, *cba*) = a (ou b ou c)
  - LCS(*lasanha*, *lenhador*) = lnha
  - LCS(*ababcde*, *bdca*) = bd

Longest Common Subsequence - LCS

- Ideia da solução: estou considerando a[i..n] e b[j..n]
    - Se a[i] == b[j], considero utilizar esse caractere na resposta
    - Considero pular para i+1 em a e continuar em j em b
    - Considero pular para j+1 em b e continuar em i para a

Longest Common Subsequence - LCS

- Ideia da solução: estou considerando a[i..n] e b[j..n]
    - Se a[i] == b[j], considero utilizar esse caractere na resposta
    - Considero pular para i+1 em a e continuar em j em b
    - Considero pular para j+1 em b e continuar em i para a

$$f(i,j) = \begin{cases} \max\{f(i+1,j), f(i,j+1), f(i+1,j+1)+1\} & i \leq n \land j \leq n \land a_i = b_j \\ \max\{f(i+1,j), f(i,j+1)\} & i \leq n \land j \leq n \land a_i \neq b_j \\ 0 & i > n \lor j > n \end{cases}$$

Longest Common Subsequence - LCS

```
int solve(int i, int j) {
  if (i >= n || j >= m) return 0;
  int ans = max(solve(i+1, j), solve(i, j+1));
  if (a[i] == b[j])
     ans = max(ans, 1 + solve(i+1, j+1));
  return ans;
}
```

Longest Common Subsequence - LCS

```
int memo[MAXN][MAXN];
int solve(int i, int j) {
  if (i >= n || j >= n) return 0;
  if (memo[i][j] != -1) return memo[i][j];
  int ans = max(solve(i+1, j), solve(i, j+1));
  if (a[i] == b[j])
     ans = max(ans, 1 + solve(i+1, j+1));
  memo[i][j] = ans;
  return ans;
}
```

Reconstrução da Solução Ótima

- Além de retornar o tamanho ou valor máximo ou mínimo de algo, alguns problemas podem pedir para reconstruir a resposta
    - Ex: no LCS, em vez de retornar só o tamanho da maior subsequência, retornar os valores dela
- Com a matriz de DP calculada, podemos transitar nesse “grafo” e encontrar a melhor resposta

Reconstrução da Solução Ótima

- Exemplo (LCS):
    - Cada chamada LCS(i, j) terá escolhido alguma das 3 opções:
        - ➢ LCS(i + 1, j + 1), se s[i]==t[j]
        - ➢ LCS(i + 1, j)
        - ➢ LCS(i, j + 1)

Reconstrução da Solução Ótima

- Exemplo (LCS):
  - Cada chamada LCS(i, j) terá escolhido alguma das 3 opções:
    - ➢ LCS(i + 1, j + 1), se s[i]==t[j]
    - ➢ LCS(i + 1, j)
    - ➢ LCS(i, j + 1)
  - Para reconstruir a solução ótima, basta ver qual dessas 3 opções que ocorreu
    - ➢ E, caso s[i]==t[j], adicionar esse caractere na resposta

Reconstrução da Solução Ótima

```
int memo[MAXN][MAXN];
string recover() {
  int i = 0, j = 0;
  string ret;
  while(i < n && j < n) {
     if (memo[i][j] == memo[i+1][j]) {
        i++;
     } else if (memo[i][j] == memo[i][j+1]) {
        j++;
     } else { // memo[i][j] == 1 + memo[i+1][j+1]
        ret += a[i];
        i++, j++;
     }
  }
  return ret;
}
```