Estruturas de Dados

Aula 17 — Programação Dinâmica

Prof Ana Carolina Sokolonski

Bacharelado em Sistemas de Informação
Instituto Federal da Bahia — Campus Feira de Santana

2026

Agenda

Subestrutura Ótima e Sobreposição de Subproblemas

Memoização vs Tabulação

Longest Common Subsequence (LCS)

0/1 Knapsack

Edição de Distância (Levenshtein)

Exercícios

Subestrutura Ótima e Sobreposição

O que torna um problema "programável dinamicamente"

O que é Programação Dinâmica?

Definição

Programação Dinâmica (PD) é uma técnica para resolver problemas de otimização **quebrando-os em subproblemas menores**, resolvendo cada subproblema **uma única vez** e armazenando o resultado para reutilização futura.

Aplicações típicas

- Comparação de sequências (LCS, alinhamento de DNA)
- Otimização de recursos (mochila, escalonamento)
- Correção ortográfica e busca difusa

Ideia central

Se um subproblema já foi resolvido, **guarde o resultado** (não recalcule). "Quem não lembra o passado é condenado a recalculá-lo."

Duas propriedades necessárias

1. **Subestrutura ótima**: a solução ótima do problema pode ser construída a partir de soluções ótimas dos subproblemas.
2. **Sobreposição de subproblemas**: os mesmos subproblemas reaparecem repetidas vezes durante a recursão.

Subestrutura Ótima

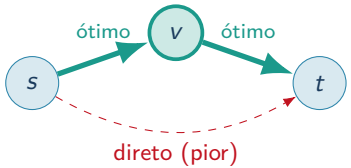
▪ Exemplo: caminho mínimo

Se o caminho mínimo de s a t passa por v, então o trecho de s a v **também é** um caminho mínimo de s a v.

▪ Por que isso ajuda?

Isso permite construir a solução global combinando soluções ótimas de subproblemas – é a base de Dijkstra, Bellman-Ford e Floyd-Warshall.

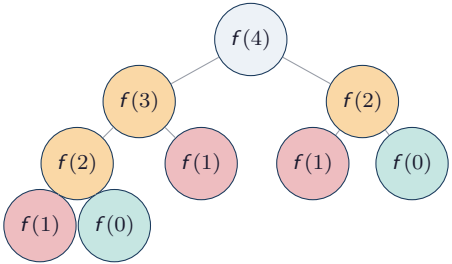
Combinação de caminhos ótimos



Caminho ótimo $s \to t$ = combinação de caminhos ótimos $s \to v$ e $v \to t$

Sobreposição de Subproblemas: Fibonacci Ingênuo

$fib(n) = fib(n-1) + fib(n-2)$, com $fib(0) = 0$, $fib(1) = 1$.



- $f(2)$ é calculado **2 vezes**, $f(1)$ é calculado **3 vezes**
- Para $f(n)$, a recursão ingênua faz $\mathcal{O}(2^n)$ chamadas – a maioria **repetida**
- Solução: **armazenar** cada subproblema resolvido (memoização/tabulação)

Memoização vs Tabulação

Top-down e Bottom-up

Duas Estratégias de Implementação

	Memoização (Top-Down)	Tabulação (Bottom-Up)
Direção	Recursiva, problema maior $\rightarrow$ menor	Iterativa, dos casos-base $\rightarrow$ maior
Estrutura	Recursão + cache (vetor/hash)	Laços + tabela (array)
Calcula	Apenas os subproblemas necessários	Geralmente todos os subproblemas
Overhead	Pilha de chamadas recursivas	Sem recursão, mais rápido na prática

- **Ambas têm a mesma complexidade assintótica**

Cada subproblema é resolvido **uma única vez** – a diferença é só a direção do cálculo e o uso de memória extra (pilha vs tabela).

Fibonacci – Memoização (Top-Down)

```c
#define N 100
long memo[N];
int  calculado[N] = {0};

long fib_memo(int n) {
    if (n <= 1) return n;
    if (calculado[n]) return memo[n];// já resolvido!
    memo[n] = fib_memo(n-1) + fib_memo(n-2);
    calculado[n] = 1;
    return memo[n];
}
```

Complexidade: $\mathcal{O}(n)$ tempo, $\mathcal{O}(n)$ espaço (cache + pilha de recursão).

Fibonacci – Tabulação (Bottom-Up)

```c
long fib_tab(int n) {
    if (n <= 1) return n;
    long dp[n+1];
    dp[0] = 0;
    dp[1] = 1;
    for (int i = 2; i <= n; i++)
        dp[i] = dp[i-1] + dp[i-2];// constrói de baixo p/ cima
    return dp[n];
}

// versao otimizada: O(1) de espaco (so guarda os 2 ultimos)
long fib_otimo(int n) {
    long a = 0, b = 1;
    for (int i = 2; i <= n; i++) {long c = a+b; a = b; b = c;}
    return (n == 0) ? a : b;
}
```

Sem recursão $\Rightarrow$ sem risco de estouro de pilha. $\mathcal{O}(n)$ tempo, $\mathcal{O}(1)$ espaço na versão otimizada.

Longest Common Subsequence

A subsequência comum mais longa

LCS – Definição

Problema

Dadas duas sequências X e Y, encontrar a maior subsequência (não necessariamente contígua, mas que preserva a ordem) comum a ambas.

Aplicações

- `diff` entre arquivos e versionamento de código
- Bioinformática (comparação de DNA/proteínas)
- Correção de digitação e plágio

Subestrutura ótima

Sejam X_i, Y_j os prefixos de tamanho i e j. Se $X[i] = Y[j]$, o último caractere **pertence** à LCS; caso contrário, a LCS de X_i, Y_j é a melhor entre ignorar o último de X ou o último de Y.

Exemplo

$X =$ "ABCB", $Y =$ "BDCB"
LCS = "BCB" (tamanho 3)

LCS – Recorrência

```c
// dp[i][j] = tamanho da LCS entre X[1..i] e Y[1..j]
int recorrencia_lcs(int **dp, char *X, char *Y, int i, int j) {
    if (i == 0 || j == 0)
        return 0; // caso-base
    if (X[i-1] == Y[j-1])
        return dp[i-1][j-1] + 1;// caractere comum: aproveita
    return (dp[i-1][j] > dp[i][j-1]) ? dp[i-1][j] : dp[i][j-1];
    // melhor entre ignorar X[i] ou Y[j]
}
```

- Tabela de tamanho $(m+1) \times (n+1)$, preenchida em $\mathcal{O}(m \cdot n)$ tempo e espaço
- A resposta final está em $dp[m][n]$
- Para **reconstruir** a subsequência, percorre-se a tabela de $dp[m][n]$ até $dp[0][0]$ seguindo as decisões que geraram cada valor

LCS – Tabela de Exemplo

$X =$ "ABCB" $\quad Y =$ "BDCB"

	B	D	C	B
A	0	0	0	0
B	1	1	1	1
C	1	1	2	2
B	1	1	2	3

$dp[4][4] = 3 \Rightarrow$ LCS tem tamanho **3** (a subsequência "BCB").

LCS – Implementação em C

```c
int lcs(char *X, char *Y, int m, int n) {
    int dp[m+1][n+1];

    for (int i = 0; i <= m; i++)
        for (int j = 0; j <= n; j++) {
            if (i == 0 || j == 0)
                dp[i][j] = 0;
            else if (X[i-1] == Y[j-1])
                dp[i][j] = dp[i-1][j-1] + 1;
            else
                dp[i][j] = (dp[i-1][j] > dp[i][j-1]) ?
                           dp[i-1][j] : dp[i][j-1];
        }
    return dp[m][n];      // tamanho da LCS
}

// uso: lcs("ABCB", "BDCB", 4, 4) retorna 3
```

Complexidade: $\mathcal{O}(m \cdot n)$ tempo e espaço.

0/1 Knapsack

O problema da mochila

Knapsack 0/1 – Definição

Problema

Dados n itens, cada um com peso w_i e valor v_i, e uma mochila de capacidade W, escolher um subconjunto de itens que **maximize o valor total** sem exceder a capacidade. Cada item é usado **no máximo uma vez** (0 ou 1, daí o nome).

Aplicações

- Alocação de orçamento
- Corte de estoque e empacotamento
- Seleção de projetos com recursos limitados

Por que não é guloso?

Escolher sempre o item de maior valor/peso pode deixar capacidade desperdiçada – é preciso considerar **todas as combinações** viáveis, o que a PD faz eficientemente.

Knapsack 0/1 – Recorrência

$dp[i][w]$ = valor máximo usando os primeiros i itens com capacidade w.

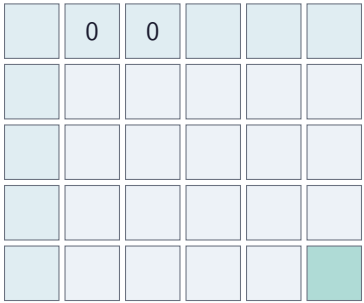
```c
// dp[i][w] = valor maximo usando os i primeiros itens, capacidade w
int recorrencia_knapsack(int **dp, int w[], int v[], int i, int cap) {
    if (i == 0 || cap == 0)
        return 0; //caso-base
    if (w[i-1] > cap)
        return dp[i-1][cap]; //item não cabe

    int sem = dp[i-1][cap];//não inclui item i
    int com = v[i-1] + dp[i-1][cap - w[i-1]];//inclui item i
    return (sem > com) ? sem : com;
}
```

- Em cada item, decide-se: **incluir** ou **não incluir**
- Se incluir, ganha v_i mas reduz a capacidade restante para $w - w_i$
- Tabela $(n+1) \times (W+1)$, preenchida em $\mathcal{O}(n \cdot W)$

Knapsack 0/1 – Exemplo

4 itens, capacidade $W = 5$:

Item	1	2	3	4
Peso (w_i)	2	3	4	5
Valor (v_i)	3	4	5	6



$dp[4][5] = 7 \Rightarrow$ valor máximo: **7** (itens 1 e 2: peso $2 + 3 = 5$, valor $3 + 4 = 7$).

Knapsack 0/1 – Implementação em C

```c
int knapsack(int W, int w[], int v[], int n) {
    int dp[n+1][W+1];

    for (int i = 0; i <= n; i++) {
        for (int cap = 0; cap <= W; cap++) {
            if (i == 0 || cap == 0) {
                dp[i][cap] = 0;
            } else if (w[i-1] > cap) {
                dp[i][cap] = dp[i-1][cap];          // nao cabe
            } else {
                int sem = dp[i-1][cap];
                int com = v[i-1] + dp[i-1][cap - w[i-1]];
                dp[i][cap] = (sem > com) ? sem : com;
            }
        }
    }
    return dp[n][W];
}
```

Complexidade: $\mathcal{O}(n \cdot W)$ tempo e espaço (otimizável para $\mathcal{O}(W)$ de espaço).

Edição de Distância

Distância de Levenshtein

Distância de Edição – Definição

Problema

Dadas duas strings X e Y, encontrar o **número mínimo de operações** (inserir, remover, substituir um caractere) necessárias para transformar X em Y.

Exemplo

Transformar "SALA" em "MESA" custa **3** operações (substituir S→M, A→E, L→S; o último A permanece).

Aplicações

- Corretores ortográficos
- `diff` de texto
- Alinhamento de sequências (bioinformática)
- Busca difusa (*fuzzy search*)
- Detecção de plágio

Distância de Edição – Recorrência

$dp[i][j]$ = distância de edição entre $X[1..i]$ e $Y[1..j]$.

```c
// dp[i][j] = distância de edição entre X[1..i] e Y[1..j]
int recorrencia_edicao(int **dp,char *X,char *Y,int i,int j){
    if (j == 0) return i; // remove tudo de X
    if (i == 0) return j; // insere tudo de Y
    if (X[i-1] == Y[j-1])
        return dp[i-1][j-1]; // caracteres iguais: sem custo
    int remover    = dp[i-1][j] + 1;
    int inserir    = dp[i][j-1] + 1;
    int substituir = dp[i-1][j-1] + 1;
    int menor = (remover < inserir) ? remover : inserir;
    return (menor < substituir) ? menor : substituir;
}
```

As três opções no caso geral correspondem a:

- $dp[i-1][j]+1 \rightarrow$ **remover** $X[i]$
- $dp[i][j-1]+1 \rightarrow$ **inserir** $Y[j]$ em X
- $dp[i-1][j-1]+1 \rightarrow$ **substituir** $X[i]$ por $Y[j]$

Distância de Edição – Tabela de Exemplo

$X =$ "SALA" $Y =$ "MESA"

	M	E	S	A
S	1	2	2	3
A	2	2	3	2
L	3	3	3	3
A	4	4	4	3

$dp[4][4] = 3 \Rightarrow$ distância de edição entre "SALA" e "MESA" é **3**.

Custo Computacional

Problema	Recorrência ingênua	Com PD	Espaço
Fibonacci	$\mathcal{O}(2^n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$ ou $\mathcal{O}(1)$
LCS	$\mathcal{O}(2^{m+n})$	$\mathcal{O}(m \cdot n)$	$\mathcal{O}(m \cdot n)$
Knapsack 0/1	$\mathcal{O}(2^n)$	$\mathcal{O}(n \cdot W)$	$\mathcal{O}(n \cdot W)$
Edição de distância	$\mathcal{O}(3^{\max(m,n)})$	$\mathcal{O}(m \cdot n)$	$\mathcal{O}(m \cdot n)$

▪ Padrão geral

Em todos os casos, a PD troca uma explosão **exponencial** por um custo **polinomial**, pagando apenas o preço de armazenar os resultados intermediários.

Exercícios

Fixando os conceitos

Exercícios

1. **Memoização (LCS):** Implemente a versão com memoização (top-down) para o problema da LCS e compare o número de chamadas recursivas com a versão ingênua, para $X =$"ABCBDAB", $Y =$"BDCABA".

2. **Reconstrução (Knapsack):** Modifique o código de Knapsack 0/1 para também **reconstruir** quais itens foram escolhidos na solução ótima (não apenas o valor).

3. **Trace manual (Edição de distância):** Calcule manualmente, preenchendo a tabela, a distância de edição entre "GATO" e "PATA". Indique a sequência de operações usada.

4. **Análise (Troco mínimo):** Mostre que o problema do **troco mínimo de moedas** (dado um conjunto de moedas, encontrar o menor número de moedas que somam um valor V) tem subestrutura ótima e sobreposição de subproblemas, e escreva sua recorrência de PD.

Fim da Aula 17

Dúvidas e próximos passos

Encerrando a Aula 17

Resumo da aula

- **PD**: subestrutura ótima + sobreposição de subproblemas
- **Memoização**: top-down, recursão + cache
- **Tabulação**: bottom-up, laços + tabela
- **LCS**: $\mathcal{O}(m \cdot n)$, base para `diff` e bioinformática
- **Knapsack 0/1**: $\mathcal{O}(n \cdot W)$, decisão incluir/não-incluir
- **Levenshtein**: $\mathcal{O}(m \cdot n)$, base de correção ortográfica

Próxima aula — Tries e Indexação de Texto

- Estrutura de Trie (árvore de prefixos)
- Busca e inserção de palavras
- Aplicações: autocompletar, corretores
- Comparação com hash tables

carolsoko@ifba.edu.br

Dúvidas? Mande uma mensagem!