Laboratório de Programação Competitiva

Profa. Silvia Brandão

2024.2

Plano de Ensino

SISTEMA DE AVALIAÇÃO - **Laboratório de Programação Competitiva**

- **Momento N3:** 25 pontos + (Uniube+ e Avaliação Institucional)
  - PROJETO PRÁTICO: **T17 e T11** – 04/12, **T12** – 05/12, 20pts;
  - Avaliação contínua/Trabalhos/Beecrowd: 5pts;

- **2ª OPORTUNIDADE dos Momentos N1 e N2:**
  - PROVA: **T17 e T11** – 11/12, **T12** – 12/12, CADA UMA 10pts; (CONTEÚDO DE TODO O SEMESTRE)

- **RECUPERAÇÃO DE NOTAS para atingir 60pts:**
  - VALOR: 20pts de prova (substituirá 1ª e 2ª avaliações); porém o aluno não poderá ter sua nota do Uniube+ zerada ou ter menos de 40pts no total do semestre.

Notas de trabalhos e projetos não são recuperáveis.

Projeto Prático – Momento N3

- **Data de entrega:** 27.11 - **Valor:** 12 pontos
  - Documentação escrita e código

- **Apresentação:** 28.11 - **Valor: 8** pontos
  - Acontecerá na MostraTec.
  - A inscrição no evento deverá ser feita pelos alunos.

Aula de hoje

- Tópicos Avançados em Algoritmos
- Programação Dinâmica
- Programação Dinâmica em Python
- Princípio da otimalidade de Bellman, memoização, tabulação.

Programação Dinâmica (PD)

- É uma abordagem algorítmica que divide um problema em subproblemas menores e resolve cada um desses subproblemas de forma recursiva. Em seguida, combina as soluções dos subproblemas para obter a solução do problema original.

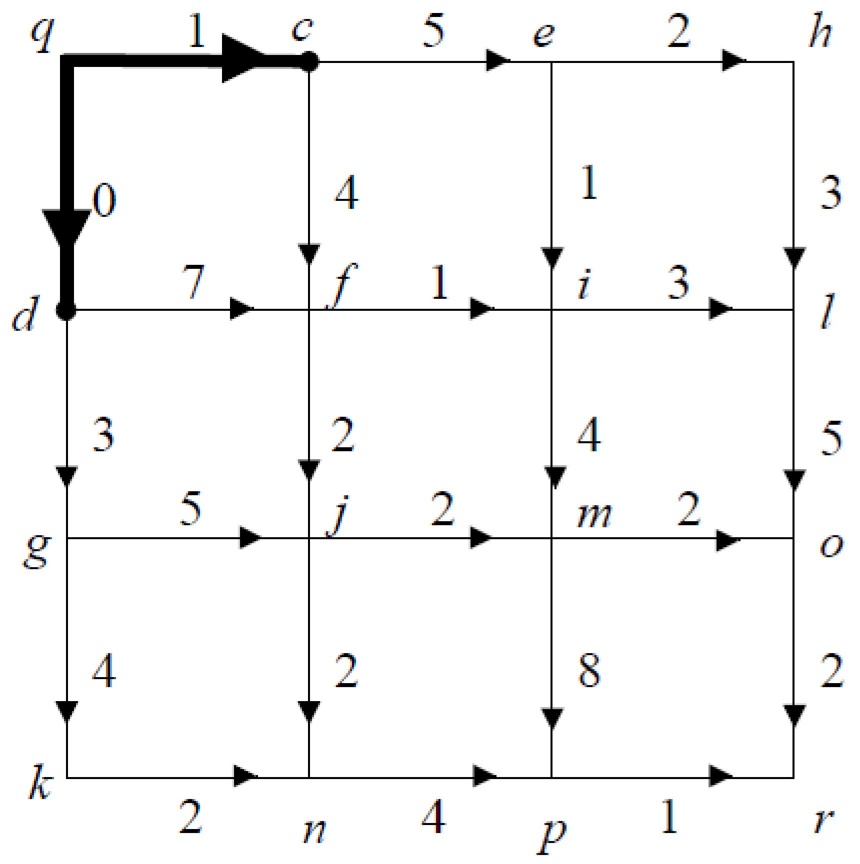
- Na programação dinâmica, é essencial **identificar as características dos subproblemas** e **encontrar uma relação de recorrência entre eles**. Essa relação permite que possamos armazenar as soluções dos subproblemas já resolvidos, evitando recálculos desnecessários e melhorando significativamente o desempenho do algoritmo.

Programação Dinâmica

- É baseada no Princípio da Otimalidade de Bellman: em uma sequência ótima de escolhas ou de decisões, cada subsequência também deve ser ótima. Ela reduz drasticamente o número total de verificações, pois evita aquelas que sabidamente não podem ser ótimas: a cada passo são eliminadas subsoluções que certamente não farão parte da solução ótima do problema.

Princípio de Otimalidade de Bellman:
"Uma trajetória ótima tem a seguinte propriedade: quaisquer que tenham sido os passos anteriores, a trajetória remanescente deverá ser uma trajetória ótima com respeito ao estado resultante dos passos anteriores, ou seja, uma política ótima é formada de subpolíticas ótimas." [Richard Bellman, 1957]

Exemplo: Caracterizando o problema



PROBLEMA DO CAMINHO MÍNIMO:

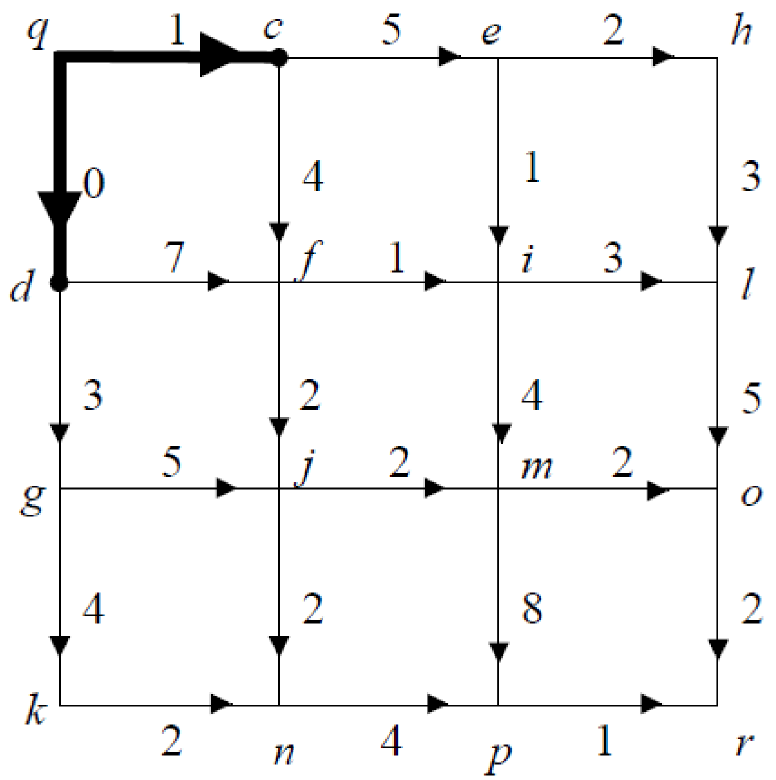
Qual é o caminho de menor esforço (tempo, custo, distância, etc.) entre q e r ?

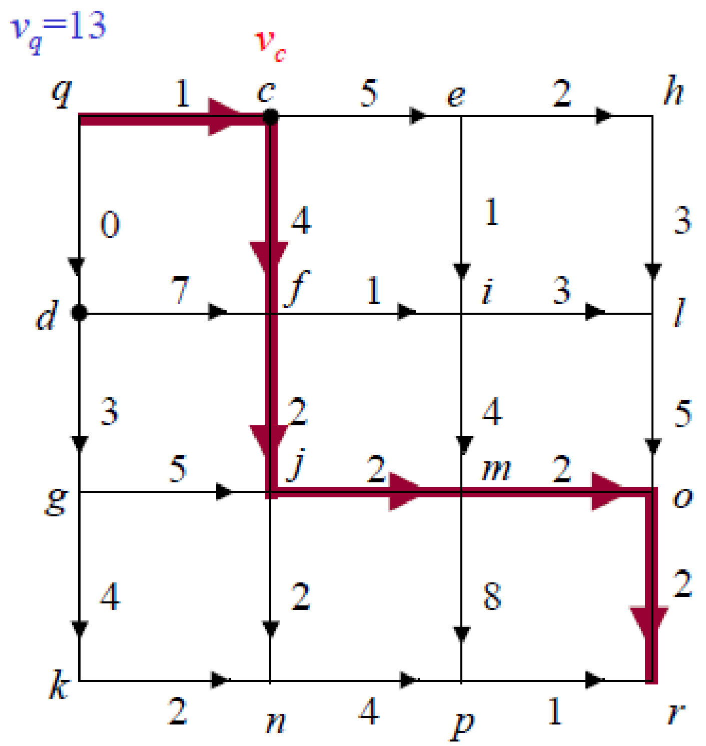
OBS.:

- 20 caminhos distintos
- 5 adições por caminho
- 19 comparações

Exemplo:

PROBLEMA DO CAMINHO MÍNIMO:





24 adições
9 comparações

Estratégia ótima

Exemplo: Caracterizando o problema

Sequência de Fibonacci: 0, 1, 1, 2, 3, 5, 8, 13,...

- Sendo que:
  - f[n] = f[n - 1] + f[n - 2], para n > 1;
  - f[0] = 0; f[1] = 1.

- A maior dificuldade para usar programação dinâmica não está na construção dos algoritmos em si, mas em discernir quando ou em que tipo de situação adotar a técnica. Além disso, por muitas vezes a especificação da solução não é trivial.

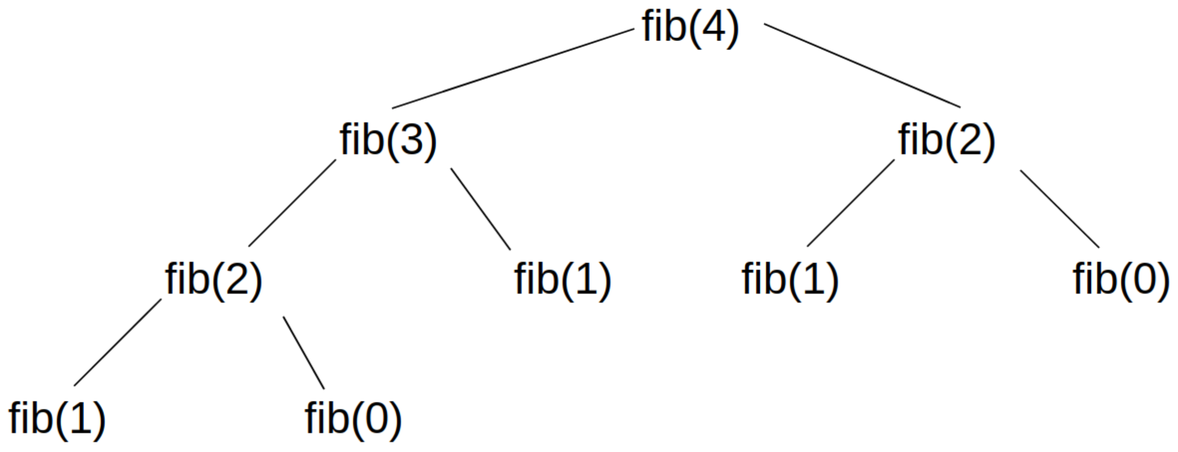
Existem quatro passos fundamentais para resolvermos problemas com essa metodologia, a saber:

1) Identificar o problema como sendo de programação dinâmica

- Duas são as características principais para a primeira etapa, a saber:
  - **Subestrutura ótima:** a solução ótima do problema provém das soluções de subproblemas dependentes. Note que já na especificação do problema vê-se que para poder encontrar o termo n, é necessário encontrar antes o termo n-1 e n-2, ou seja, a solução ótima depende da melhor resultado de outros dois subproblemas.
  - **Sobreposição de soluções:** a solução ótima passa pela resolução de subproblemas que aparecem duas ou mais vezes. Veja:

Tomando como exemplo **n = 4 ,** perceba que o cálculo de **fib(2)** é requisitado por duas vezes.

Assim, encontrar termos dessa série possui sobreposição de soluções.



Chamadas recursivas para cálculo de termo da série de Fibonacci.

Existem quatro passos fundamentais para resolvermos problemas com essa metodologia, a saber:

2) Definir o valor da solução ótima recursivamente

- Para o caso da série de Fibonacci, esta etapa é relativamente simples já que na própria especificação do problema observa-se que a própria função é demandada novamente para descoberta dos termos n-1 e n-2. Levando isso para o Python, tem-se:

```python
def fibonacci(n):
    if n == 1 or n == 0:
        return n
    return fibonacci(n-1) + fibonacci(n-2)
```

Existem quatro passos fundamentais para resolvermos problemas com essa metodologia, a saber:

3) Calcular valor da solução ótima da forma "bottom-up" ou "top-down"

- Aqui está a diferença no uso de programação dinâmica.
- A ideia da técnica é evitar cálculos repetidos na busca da solução ótima de um problema recursivo. Para isso, **cria-se uma estrutura na memória** a fim de guardar tais resultados. Há duas abordagens para alcançar tal objetivo as quais serão exploradas no slide a seguir.

Memorização e otimização de subestrutura

- Um dos principais conceitos da programação dinâmica é a chamada **"memorização"** ou **"memoização".**
- Essa técnica envolve o armazenamento das soluções dos subproblemas em uma estrutura de dados, como uma tabela ou matriz, para que possam ser reutilizadas posteriormente.
- A programação dinâmica se beneficia da "otimização de subestrutura", que permite construir a solução ótima de um problema a partir das soluções ótimas de seus subproblemas.

Existem quatro passos fundamentais para resolvermos problemas com essa metodologia, a saber:

3) Calcular valor da solução ótima da forma "bottom-up" ou "top-down"

Abordagem "Top-Down"

- Na abordagem "top-down" (memoização), partimos da solução geral ótima que se deseja encontrar e, então, analisa-se quais subproblemas são necessários resolver até que se chegue em um subproblema com resolução trivial. **OBS.:** Ao longo dos cálculos os resultados são armazenados para que sejam reutilizados (memo).
- Dessa forma, o algoritmo observa primeiramente na tabela se a solução ótima do subproblema já foi computado. Caso positivo, simplesmente extrai o valor. Caso negativo, resolve e salva o resultado na tabela (lista).
- O código a seguir mostra a solução para a série de Fibonacci usando essa abordagem.

```
def fibonacciTopDown(n, memo):
    if n == 1 or n == 0:
        return n
    memo[n] = fibonacciTopDown(n-1, memo) + fibonacciTopDown(n-2, memo)
    return memo[n]
```

Estrutura de dados (tabela, lista, matriz,...)

Existem quatro passos fundamentais para resolvermos problemas com essa metodologia, a saber:

3) Calcular valor da solução ótima da forma "bottom-up" ou "top-down"

Abordagem "Bottom-Up"

- Na abordagem "bottom-up" (tabulação), diferente da anterior, a solução ótima começa a ser calculada a partir do subproblema mais trivial.
- No caso da série de Fibonacci, basta entender que para se calcular o **termo n**, a resolução sempre inicia pelo fib(0), depois fib(1), fib(2) e assim sucessivamente até chegar em fib(n).

- O código abaixo mostra a implementação dessa abordagem.

Estrutura de dados (tabela, lista, matriz,...

```
def fibonacciBottomUp(n, table):
    table[0] = 0
    table[1] = 1
    for cont in range(2, n + 1):
        table[cont] = table[cont - 1] + table[cont - 2]
    return table[n]
```

```
#exemplo e uso
n = 10
memo = [None] * (n+1)
print(fibonacciTopDown(n, memo))
```

3) Calcular valor da solução ótima da forma "bottom-up" ou "top-down"

Abordagem top-down x bottom-up

- Qual a vantagem de se utilizar uma forma ou outra de implementação?
- A tabela abaixo lista características importantes de cada metodologia.

Quesitos	Top-down	Bottom-up
Especificação do problema	É mais fácil de se pensar no problema	Formular o problema pode ser complexo
Código	Codificação mais simples e com regras mais simples	Torna-se inviável quando há muitas condições
Resolução dos Subproblemas	Resolve apenas os subproblemas que são necessários	Resolve todos os subproblemas do problema original
Parâmetros na Tabela	A tabela é preenchida apenas com as soluções necessárias	Como parte da solução mais básica, preenche a tabela com soluções que podem ser desnecessárias

Existem quatro passos fundamentais para resolvermos problemas com essa metodologia, a saber:

4) Analisando o Desempenho

- Entendidas as características e vantagens de cada abordagem, vamos testar o desempenho de cada uma delas na resolução da série de Fibonacci.
- Verificou-se o tempo de execução de cada uma das implementações de acordo com o código no slide a seguir:

Analisando o desempenho: top-down x bottom-up

```python
import time
for n in range(35, 40):
    start = time.time()
    result = fibonacci(n)
    finish = time.time()
    print("===========================================================================")
    print("Fibonacci(", n, ")")
    print("Resultado - Fibonacci 1 (Original): ", result)
    print("Tempo Total de Execução - Fibonacci 1: ", round(finish - start, 2), "segundos")
    memo = [None] * (n+1)
    start = time.time()
    result = fibonacciTopDown(n, memo)
    finish = time.time()
    print("Resultado - Fibonacci 2 (Top-Down): ", result)
    print("Tempo Total de Execução - Fibonacci 2: ", round(finish - start, 20), "segundos")
    memo = [None] * (n+1)
    start = time.time()
    result = fibonacciBottomUp(n, memo)
    finish = time.time()
    print("Resultado - Fibonacci 3 (Bottom-Up): ", result)
    print("Tempo Total de Execução - Fibonacci 3: ", round(finish - start, 20), "segundos")
    print("===========================================================================")
```

Analisando o desempenho: top-down x bottom-up

```
=================================================================
Fibonacci( 35 )
Resultado - Fibonacci 1 (Original):  9227465
Tempo Total de Execução - Fibonacci 1:  4.36 segundos
Resultado - Fibonacci 2 (Top-Down):  9227465
Tempo Total de Execução - Fibonacci 2:  6.568013668060303 segundos
Resultado - Fibonacci 3 (Bottom-Up):  9227465
Tempo Total de Execução - Fibonacci 3:  1.549720764160156e-05 segundos
=================================================================
=================================================================
Fibonacci( 36 )
Resultado - Fibonacci 1 (Original):  14930352
Tempo Total de Execução - Fibonacci 1:  6.96 segundos
Resultado - Fibonacci 2 (Top-Down):  14930352
Tempo Total de Execução - Fibonacci 2:  9.895121097564697 segundos
Resultado - Fibonacci 3 (Bottom-Up):  14930352
Tempo Total de Execução - Fibonacci 3:  9.05990600585938e-06 segundos
=================================================================
=================================================================
Fibonacci( 37 )
Resultado - Fibonacci 1 (Original):  24157817
Tempo Total de Execução - Fibonacci 1:  13.97 segundos
Resultado - Fibonacci 2 (Top-Down):  24157817
Tempo Total de Execução - Fibonacci 2:  14.980348348617554 segundos
Resultado - Fibonacci 3 (Bottom-Up):  24157817
Tempo Total de Execução - Fibonacci 3:  1.406669616699219e-05 segundos
=================================================================
```

Analisando o desempenho: top-down x bottom-up

```
===================================================================
Fibonacci( 38 )
Resultado - Fibonacci 1 (Original):  39088169
Tempo Total de Execução - Fibonacci 1:  20.79 segundos
Resultado - Fibonacci 2 (Top-Down):  39088169
Tempo Total de Execução - Fibonacci 2:  24.70142889022827 segundos
Resultado - Fibonacci 3 (Bottom-Up):  39088169
Tempo Total de Execução - Fibonacci 3:  1.573562622070312e-05 segundos
===================================================================
===================================================================
Fibonacci( 39 )
Resultado - Fibonacci 1 (Original):  63245986
Tempo Total de Execução - Fibonacci 1:  32.15 segundos
Resultado - Fibonacci 2 (Top-Down):  63245986
Tempo Total de Execução - Fibonacci 2:  40.089319944381714 segundos
Resultado - Fibonacci 3 (Bottom-Up):  63245986
Tempo Total de Execução - Fibonacci 3:  1.668930053710938e-05 segundos
===================================================================
```

Analisando o desempenho: top-down x bottom-up

- Escolheu-se o cálculo de termos maiores (35 a 39) para mostrar como o código, usando apenas a recursividade, começa a se tornar extremamente lento à medida que n cresce.
- Por outro lado, com programação dinâmica, tanto a abordagem top-down quanto bottom-up consegue um desempenho muito superior como pôde ser visto.
- Observe também que, nesses casos, **a abordagem bottom-up mostrou-se superior**.

- O método bottom-up utiliza uma abordagem iterativa para calcular os números de Fibonacci, evitando a sobrecarga associada à recursão presente nos métodos Fibonacci 1 (Original) e Fibonacci 2 (Top-Down).

Vimos o quão lento a resolução pode se tornar com o uso apenas da recursividade e a grande eficiência alcançada no que tange ao tempo de execução dos códigos provindos do uso da metodologia referente à programação dinâmica.

Aprender programação dinâmica pode abrir um mundo de possibilidades no desenvolvimento de algoritmos eficientes para resolver problemas complexos.

Estudos de casos

- Caminho mais curto em um grafo
- Multiplicação de matrizes encadeadas
- Pêndulo invertido
- Problema da Mochila
- Problema do Troco: https://panda.ime.usp.br/panda/static/pythonds_pt/04-Recursao/11-programacaoDinamica.html
- Aplicações em Economia para apreçamento de ativos usando Equações de Euler

Faça, usando programação dinâmica:

3ª. Lista de Exercícios:

- Beecrowd 1029 - Fibonacci, Quantas Chamadas?
- Beecrowd 1166 - Torre de Hanoi, Novamente!

Referências

- CORMEN, Thomas H. et al. **Introduction to algorithms**. MIT press, 2009.
- Geeks for Geeks: https://www.geeksforgeeks.org/dynamic-programming/#concepts