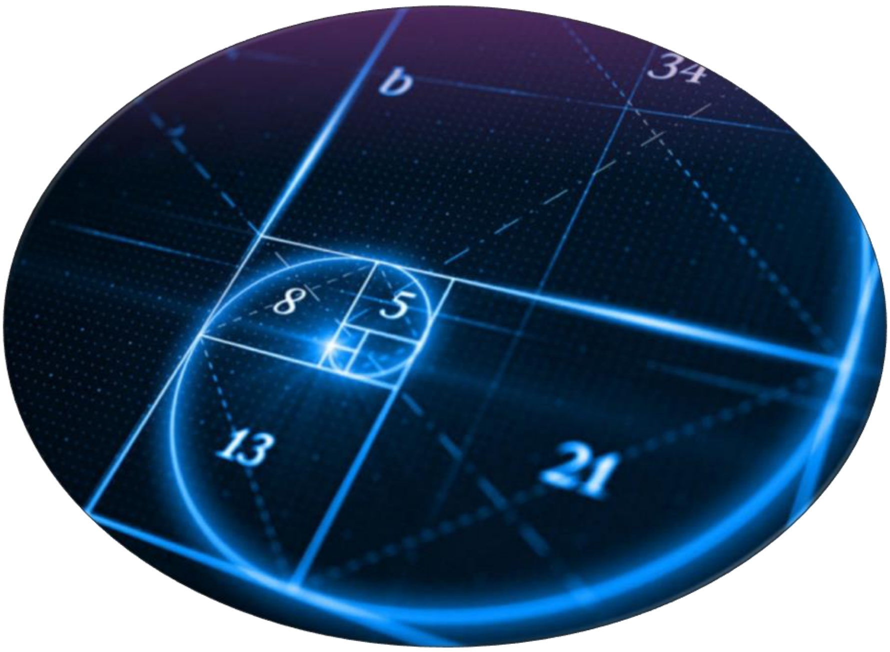
Instituto Federal de Educação, Ciência e Tecnologia de São Paulo

Trabalho Análise de Algoritmos

Docente: **Bento Rafael Siqueira**

Discentes:
Rodrigo Luis Tavano Bosso – PC3005623

Programação Dinâmica



0-Resumo:

O seguinte trabalho tem como objetivo explicar o funcionamento dos métodos de Programação Dinâmica passando por seus conceitos, análise de desempenho e apresentação de alguns exemplos.

Para que se possa realizar tal discussão, estaremos fundamentados em uma série de pesquisas em links e livros que explicam o funcionamento geral da Programação Dinâmica. Consoante a tais fatores partiremos para o texto propriamente dito.

Palavras-chave: Algoritmos, Programação , Dinâmica, Fibonacci.

1- Introdução:

1.1- O que é Programação Dinâmica?

Programação dinâmica é método de programação de computador que busca a melhoria de algoritmos com base em princípios matemáticos e computacionais.

Richard Bellman desenvolveu o primeiro método na década de 1950 e encontrou aplicações as quais foi empregadas vários campos, de economia e engenharia aeroespacial.

Se temos problemas que podem ser aninhados recursivamente dentro de problemas maiores, e existir uma relação entre o valor do problema maior e os valores dos subproblemas. Então esses são passíveis de serem tratados de forma dinâmica.

A medida do avanço desse texto, iremos encontrar exemplos que ilustram o uso e o funcionamento de recursos da programação dinâmica.

2- Conceitos Básicos:

2.1- Requisitos:

Para um problema ser resolvido de forma dinâmica deve-se preencher determinados requisitos, são eles:

1. *Problema passível de ser resolvido de forma recursiva.*
2. *Problema não ser cíclico e/ou dependente.*
3. *A resposta do problema principal ser um conjunto combinatório dos subproblemas relacionados*

Com esses requisitos atendidos pode-se começar a pensar em implementação de programação dinâmica.

2.2 – Abordagem Top Down:

A abordagem Top - Down aproxima o problema de forma recursiva comum. Assim, analisa-se quais subproblemas são necessários resolver até que se chegue em um subproblema com resolução trivial.

Também se usa o sistema de memória, em que ao longo dos cálculos os resultados são armazenados para que sejam reutilizados. Dessa forma, o algoritmo observa primeiramente na tabela se a solução ótima do subproblema já foi computada, assim retornando um cálculo que seria refeito com uma resposta já computada com complexidade O(U). Caso positivo, simplesmente extrai o valor. Caso negativo, resolve e salva o resultado na tabela.

2.3 – Abordagem Bottom Up:

A abordagem Bottom Up, vamos calculando os subproblemas menores e aumentando a complexidade com o decorrer da execução, diferente da anterior, a solução ótima começa a ser calculada a partir do subproblema mais trivial.

2.4 – Aplicação no exemplo de Fibonacci:

Um dos exemplos mais usados para explicar as abordagens de programação dinâmica é o exemplo de Fibonacci.

Esse é um exemplo de algoritmo Up-Down aplicado ao problema de computação de Fibonacci:

```
def fibonacciTopDown(n, table = {}):

    if n == 1 or n == 0:

        return n    (casos mais básicos sendo tratados com o retorno de N)

    try:

        return table[n]

    except:

        table[n] = fibonacciTopDown(n-1) + fibonacciTopDown(n-2)        (realização da sequência
de forma recursiva )

        return table[n]
```

Mas também pode-se realizar o algoritmo Bottom – Up , em que se realiza o código baseado na mesma lógica do seguinte algoritmo:

```
def fibonacciBottomUp(n, table = {}): {

  table[0] = 0

  table[1] = 1              ( casos básicos nos primeiras posições)

  for cont in range(2, n + 1):

    table[cont] = table[cont - 1] +  table[cont - 2]  (realização da sequência de forma recursiva )

  return table[n]
```

3- Desempenho:

Analisar o desempenho, depende claramente do problema inicial. Mas geralmente ocorre uma redução de complexidade de $O\ (2^n)$ para $O(N)$.

Antes de partir para os exemplos diversificados, segue a análise de desempenho entre Abordagem Top Down e Abordagem Botton Up :

Pontos de Análise	Top-Down	Bottom-UP
Propriedade de problema	É mais fácil de pensar no problema	Tendência de formulação complexa
Código	Codificação e regras simplificadas	Inviável se há muitas condições
Resolução de Subproblemas	Resolve apenas os subproblemas que são necessários	Resolve todos os subproblemas do problema original
Parâmetros de Tabela	A tabela é preenchida apenas com as soluções necessárias	Coo parte da solução mais básica, preenche a tabela com soluções que podem ser desnecessárias

Com as características e vantagens de cada abordagem apresentadas acima, vamos testar o desempenho de cada uma delas na resolução da série de Fibonacci. Para isso, verificou-se o tempo de execução de cada uma das implementações de acordo com os códigos abaixo:

```
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")

    start = time.time()

    result = fibonacciTopDown(n)

    finish = time.time()

    print("Resultado - Fibonacci 2 (Top-Down): ", result)

    print("Tempo Total de Execução - Fibonacci 2: ", round(finish - start, 20), "segundos")

    start = time.time()

    result = fibonacciBottomUp(n)

    finish = time.time()

    print("Resultado - Fibonacci 3 (Bottom-Up): ", result)

    print("Tempo Total de Execução - Fibonacci 3: ", round(finish - start, 20), "segundos")

print("=================================================================")
```

(5)

Apresentando a saída com base nesse código:

Fibonacci(35)

Resultado - Fibonacci 1 (Original): 9227465

Tempo Total de Execução - Fibonacci 1: 6.13 segundos

Resultado - Fibonacci 2 (Top-Down): 9227465

Tempo Total de Execução - Fibonacci 2: 3.57627868652344e-06 segundos

Resultado - Fibonacci 3 (Bottom-Up): 9227465

Tempo Total de Execução - Fibonacci 3: 1.7404556274414062e-05 segundos

===

Fibonacci(36)

Resultado - Fibonacci 1 (Original): 14930352

Tempo Total de Execução - Fibonacci 1: 9.84 segundos

Resultado - Fibonacci 2 (Top-Down): 14930352

Tempo Total de Execução - Fibonacci 2: 3.337860107421875e-06 segundos

Resultado - Fibonacci 3 (Bottom-Up): 14930352

Tempo Total de Execução - Fibonacci 3: 1.4543533325195312e-05 segundos

===

Fibonacci(37)

Resultado - Fibonacci 1 (Original): 24157817

Tempo Total de Execução - Fibonacci 1: 15.9 segundos

Resultado - Fibonacci 2 (Top-Down): 24157817

Tempo Total de Execução - Fibonacci 2: 5.4836273193359375e-06 segundos

Resultado - Fibonacci 3 (Bottom-Up): 24157817

Tempo Total de Execução - Fibonacci 3: 3.7431716918945312e-05 segundos

===

Fibonacci(38)

Resultado - Fibonacci 1 (Original): 39088169

Tempo Total de Execução - Fibonacci 1: 26.14 segundos

Resultado - Fibonacci 2 (Top-Down): 39088169

Tempo Total de Execução - Fibonacci 2: 3.337860107421875e-06 segundos

Resultado - Fibonacci 3 (Bottom-Up): 39088169

Tempo Total de Execução - Fibonacci 3: 2.574920654296875e-05 segundos

===

Fibonacci(39)

Resultado - Fibonacci 1 (Original): 63245986

Tempo Total de Execução - Fibonacci 1: 49.29 segundos

Resultado - Fibonacci 2 (Top-Down): 63245986

Tempo Total de Execução - Fibonacci 2: 7.152557373046880e-06 segundos

Resultado - Fibonacci 3 (Bottom-Up): 63245986

Tempo Total de Execução - Fibonacci 3: 2.7179718017578120e-05 segundos

==

(5)

4- Exemplos:

4.1- Mochila booleana

Se p[1..n] e v[1..n] são vetores numéricos e X é um subconjunto do intervalo 1..n, denotaremos por p(X) e v(X) as somas $\sum_{i \in X} p[i]$ e $\sum_{i \in X} v[i]$ respectivamente.

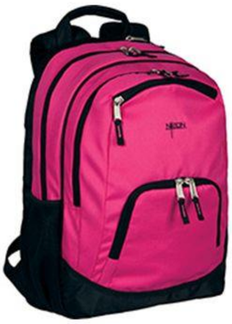
Problema da Mochila Booleana: Dados vetores p[1..n] e v[1..n] de números naturais e um número natural c, encontrar um subconjunto X do intervalo 1..n que maximize v(X) sob a restrição p(X) ≤ c. Geralmente ilustrado com uma mochila comum capacidade de peso máximo p(X) e com objetos ou recursos com valores v(X).

Nosso problema é encontrar uma mochila de valor máximo, sem que extrapole o peso máximo definido.

O problema da mochila booleana tem estrutura recursiva: qualquer solução de uma instância é composta por soluções de seus subproblemas.

Suponha que X é uma solução da instância (p, v, n, c) do problema. Se n ∉ X então X é solução do subproblema (p, v, n−1, c)   e se n ∈ X então X − {n} é solução do subproblema (subpágina) (p, v, n−1, c−p[n]).

Reciprocamente, se Y é solução da instância (p, v, n−1, c) e Y′ é solução da instância (p, v, n−1, c−p[n]) então Y ou Y′ ∪ {n} é solução da instância (p, v, n, O algoritmo consome Ω(2n) unidades de tempo e portanto só é útil para valores muito pequenos de n.



Mochila: 10 kg

Item	Peso	Valor
1	6	30
2	3	14
3	4	16
4	2	9

Ideia

Problema a resolver K(10,4)

Começa no último item : 4
  Seu peso é ≤ 10 ?
    Se Sim:
      Tenho duas opções:
        1. Coloco o item na mochila :
          subproblema a resolver K(10-2, 3)
        2. Não coloco o item na mochila
          subproblema a resolver K(10,3)
      Resultado = max{K(10-2,3) + 9, K(10,3) }
  Se Não:
      subproblema a resolver K(10,3)
      Resultado = resultado de K(10,3)

Algoritmo:

```
public class Mochila {
  /* abordagem bottom-up       *começamos pelo caso base: zero itens com zero valor
  *e começamos a encher a mochila      */
 public static int calcula (int capacidade, int[] pesos, int[] valores) {
  int n = pesos.length;
  int [][] k = new int[n + 1][capacidade+1];
  for(int i = 0; i <= n; i++) {
   for(int j = 0; j <= capacidade; j++) {
    if(( i == 0) || (j ==0))// condição inicial
     k[i][j] = 0;
    else
     //ainda da para tentar inserir o item na mochila
    if(pesos[i-1] <= j)
     // 2 condições: ainda tem espaço ou tentamos retirar um item
     k[i][j] = Math.max(valores[i-1] + k[i-1][j-pesos[i-1]], k[i-1][j]);
    else
     //mochila já está cheia
     k[i][j] = k[i-1][j];
   }
    }
```

```java
//imprime matriz
  for(int i = 0; i <= n; i++) {
   for(int j = 0; j <= capacidade; j++)
    System.out.printf("%3d ", k[i][j]);
   System.out.println();
 }
 return k[n][capacidade];
}
public static void main(String []args) {
 int[] valores = {60, 100, 120};
 int[] pesos = {10,20,30};
 int capacidade = 50;
 System.out.printf("Valor maximo conseguido na mochila - %d\n", calcula(capacidade, pesos, valores));
 }
}
```

5- Referências Bibliográficas:

1. https://en.wikipedia.org/wiki/Dynamic_programming#Computer_programming
2. https://mat.gsia.cmu.edu/classes/dynamic/dynamic.html
3. http://www.spoj.com/problems/ACODE/
4. http://www.ime.usp.br/~maratona/
5. https://lamfo-unb.github.io/2019/05/30/Programacao-Dinamica/
6. https://www.ime.usp.br/~pf/analise_de_algoritmos/aulas/dynamic-programming.html
7. Cormen, Thomas (2009). Algoritmos: Teoria e Prática 3 ed. [S.l.]: Campus
8. http://wiki.icmc.usp.br/images/1/1a/PD1.pdf

Instituto Federal de Educação, Ciência e Tecnologia de São Paulo

Trabalho Análise de Algoritmos

Docente: Bento Rafael Siqueira

Discentes:

Rodrigo Luis Tavano Bosso – PC3005623

Programação Dinâmica