Programação Dinâmica

Prof. Leandro C. Fernandes

Baseado no material de Pedro Ribeiro (Universidade Porto) e no livro "Algoritmos: Teoria e Prática" - Cormen, Leiserson, Rivest, C. Stein

Sumário

- Motivação e conceitos base
  - Exemplos iniciais e cálculos repetidos
- Programação Dinâmica
  - Definição
  - Características de um problema de PD
  - Passos para chegar a uma solução

Sequência Fibonacci

- Sequência de números definida por Leonardo Fibonacci: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...
- Definida por:

  F(0) = 0

  F(1) = 1

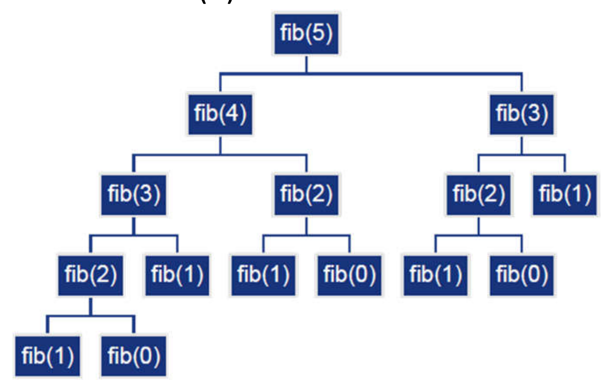
  F(n) = F(n-1) + F(n-2)

- Implementação direta:

```
fib(n):
 se n=0 ou n=1 então
    retornar n
 senão
    retornar fib(n-1) + fib(n-2)
```

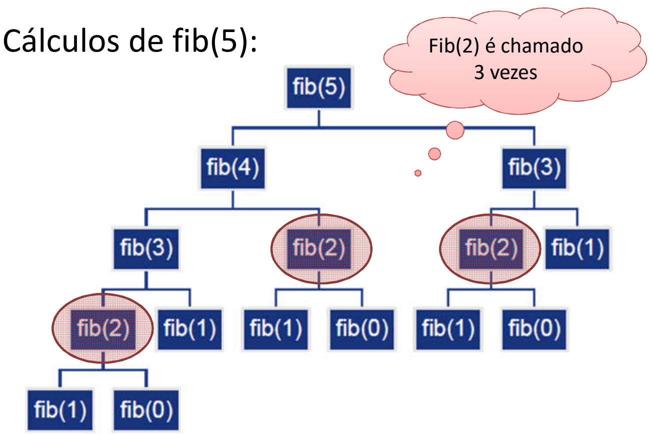
Sequência Fibonacci

- Cálculos de fib(5):



Sequência Fibonacci

- Cálculos de fib(5):



Sequência Fibonacci

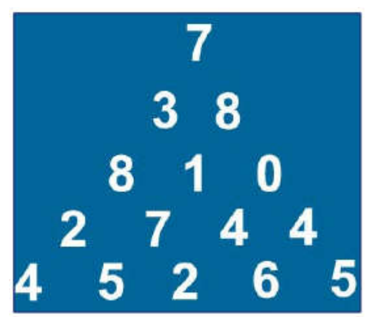
- *Como melhorar nossa implementação inicial?*
- Uma boa sugestão é: começar do zero e ir mantendo sempre em memória os **dois últimos números** da sequência.

```
fib(n):
 se n=0 ou n=1 então
   retornar n
 senão
   f1 = 1
   f2 = 0
   para i:2 até n faça
      f = f1 + f2
      f2 = f1
      f1 = f
  retornar f
```

O que observamos ...

- Aspectos importantes a serem lembrados:
  - Divisão de um problema em **subproblemas do mesmo tipo**
  - Calcular o mesmo subproblema **apenas uma vez**
- Será que tais ideias podem ser aplicadas em outros problemas mais complicados?

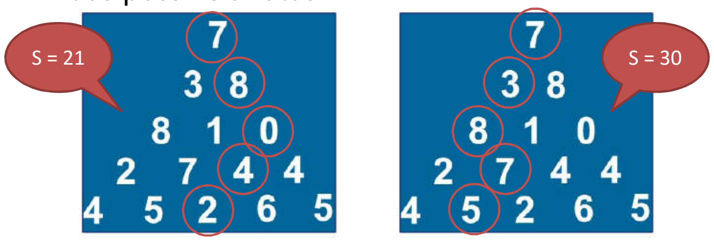
Pirâmide de Números



Calcular a rota, que começa no topo da pirâmide e acaba na base, com maior soma . Em cada passo podemos ir diagonalmente para baixo e para a esquerda ou para baixo e para a direita.

Pirâmide de Números

- Duas possíveis rotas



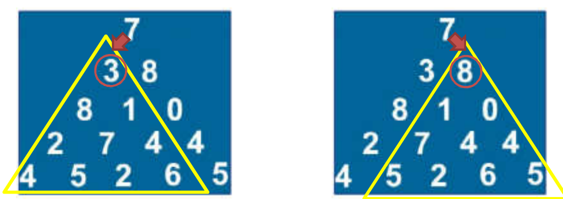
- **Limites:** todos os números da pirâmide são inteiros entre 0 e 99 e o número de linhas do triângulo é no máximo 100.

Como resolver o problema?

- Ideia: **Força Bruta!**
  - Avaliar todos os caminhos possíveis e ver qual o melhor.
- Mas quanto tempo demora isto?
  - Analise da complexidade:
    - Em cada linha podemos tomar **duas** decisões diferentes: esquerda ou direita. Seja *n* a altura da pirâmide. Uma rota é constituída por *n* - 1 decisões diferentes, então existem 2^{n-1} caminhos diferentes.
    - Um programa que calculasse todas rotas teria complexidade $O(2^n)$: crescimento exponencial!
  - Note que $\mathbf{2^{99} \approx 6{,}34 \times 10^{29}}$, que é um número absurdamente grande! o.O

(Re)Analisando o problema

- Quando estamos no topo da pirâmide, temos duas decisões possíveis (esquerda ou direita):



- Em cada um dos casos, temos de ter em conta todas as rotas das respectivas subpirâmides assinaladas a amarelo.

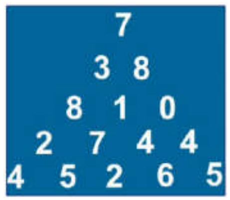
(Re)Analisando o problema

- *O que nos interessa nestas subpiramides?*
- **O valor da sua melhor rota interna. Observe que isto é um instância menor do mesmo problema!**
- Para o exemplo, a solução do problema será: 7 + o máximo entre os valores (resultante da melhor rota) de cada uma das subpiramides.

Uma solução recursiva

- Resolvendo recursivamente:
  - Seja **P[i][j]** o *j*-ésimo número da *i*-ésima linha
  - Seja **Max(i,j)** o melhor que conseguimos a partir da posição i,j

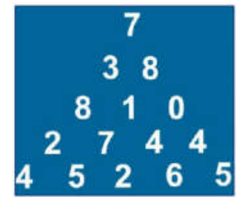
	1	2	3	4	5
1	7				
2	3	8			
3	8	1	0		
4	2	7	4	4	
5	4	5	2	6	5



Uma solução recursiva

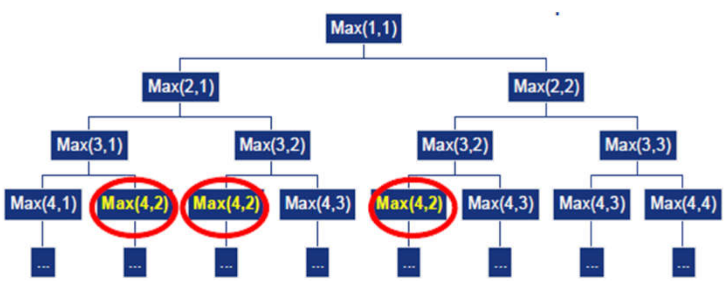
```
Max(i,j):
 se i = n então
   Max(i,j) = P[i][j]
 senão
   Max(i,j) = P[i][j] +
              maior(Max(i+1,j), Max(i+1,j+1))
```

	1	2	3	4	5
1	7				
2	3	8			
3	8	1	0		
4	2	7	4	4	
5	4	5	2	6	5



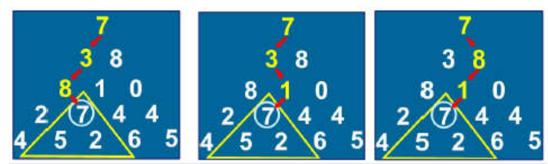
Analise da solução recursiva

- Continuamos com crescimento exponencial!



Analise da solução recursiva

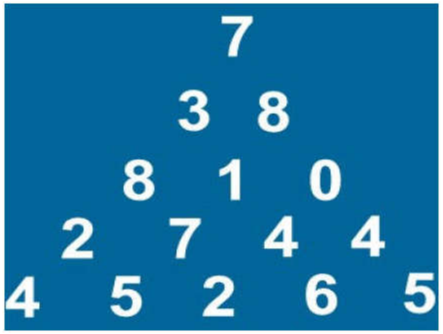
- Estamos avaliando o mesmo subproblema várias vezes!



  - Temos de **reaproveitar** o que já foi calculado, ou seja, calcular apenas uma vez o mesmo subproblema!
- Ideia:
  - Criar uma **tabela** com o valor obtido para cada subproblema: M[i][j]
  - Existe uma **ordem para preencher a tabela** de modo a que quando precisamos de um valor já o temos?

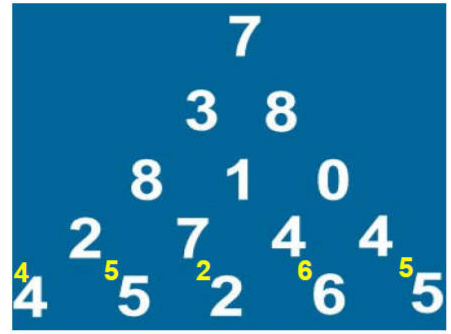
Montando a tabela M

- Começar a partir do fim!



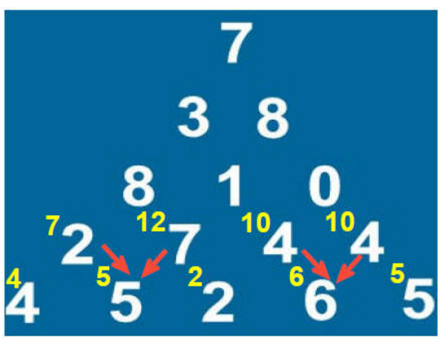
Montando a tabela M

- Começar a partir do fim!



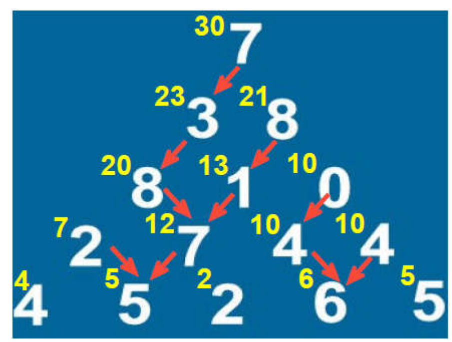
Montando a tabela M

- Começar a partir do fim!



Montando a tabela M

- Começar a partir do fim!



Solução para a Pirâmide

- Tendo em conta a maneira como preenchemos a tabela, podemos até aproveitar **P[][]** !

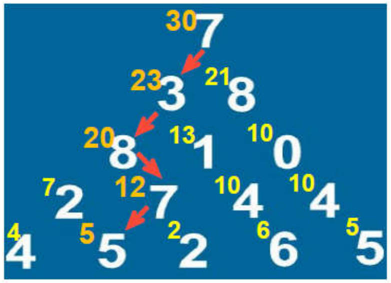
```
Calcular():
  Para i: n-1 ate 1 faça
    Para j: 1 ate i faça
      P[i][j] = P[i][j] + máximo(P[i+1][j], P[i+1][j+1])
```

  - Com isto solução fica em **P[1][1]**!

- Agora, o tempo necessário para resolver o problema cresce **polinomialmente**, i.e. **O(n^2)**, e temos uma solução admissível para o problema **99^2 = 9801** B-)

O caminho dentro da Pirâmide

- Se fosse necessário saber o caminho que leva a melhor solução?
  - Basta usar a tabela já calculada!



O que fizemos!?

- Para resolver o problema da pirâmide de números usamos…

Programação Dinâmica (PD)

Programação Dinâmica

Definição (adaptado de NIST-DADSP*):

- técnica algorítmica, normalmente usada em problemas de otimização, que é baseada em guardar os resultados de subproblemas, em vez de recalculá-los.
  - **Técnica algorítmica:** método geral para resolver problemas que têm algumas características em comum
  - **Problema de optimização:** quando se pretende encontrar a "melhor" solução entre todas as soluções admissíveis, mediante um determinado critério (*função objetivo*).

* **NIST** – National Institute of Standarts and Technology
DADSP – Dictionary of Algorithms, Data Structures, and Problems

Programação Dinâmica

- Quais são as características que um problema deve apresentar para poder ser resolvido com Programação Dinâmica?
  - Subestrutura ótima
  - Subproblemas coincidentes

Características do problema

- **Subestrutura ótima (*optimal substructure*):**
  - Quando a solução óptima de um problema contém nela própria soluções óptimas para subproblemas do mesmo tipo.
    - **Exemplo:** *No problema das pirâmides de números, a solução óptima contém nela própria os melhores percursos de subpirâmides, ou seja, soluções óptimas de subproblemas.*
  - Quando um problema apresenta esta característica diz-se que ele respeita o **princípio de optimalidade** (*optimality principle*).

Características do problema

- **Subproblemas coincidentes:**
  - Quando um espaço de subproblemas é pequeno, isto é, não são muitos os subproblemas a resolver pois muitos deles são exatamente iguais uns aos outros.
    - **Exemplo:** *no problema das pirâmides, para um determinada instância do problema, existem apenas n`+ (n-1) + … + 1 < n^2 subproblemas (crescem polinomialmente) pois, como já vimos, muitos subproblemas que aparecem são coincidentes.*
  - Também esta característica nem sempre acontece, quer porque mesmo com subproblemas coincidentes são muitos subproblemas a resolver, quer porque não existem subproblemas coincidentes.
    - **Exemplo:** *no quicksort, cada chamada recursiva é feita a um subproblema novo, diferente de todos os outros.*

Programação Dinâmica

Etapas:

1) Caracterizar a solução ótima do problema
2) Definir recursivamente a solução ótima, em função de soluções ótimas de subproblemas
3) Calcular as soluções de todos os subproblemas: “de trás para a frente” (abordagem *botton-up*) ou com “memoization”
4) Reconstruir a solução ótima, baseada nos cálculos efetuados (opcional)

(**nota:** estes passos representam apenas um guia de resolução)

Metodologia

1) Caracterização da solução ótima

– Compreender bem o problema
– Verificar se um algoritmo que verifique todas as soluções à força bruta não é suficiente
– Tentar generalizar o problema (é preciso prática para perceber como generalizar da maneira correta)
– Procurar dividir o problema em subproblemas do mesmo tipo
– Verificar se o problema obedece ao princípio de optimalidade
– Verificar se existem subproblemas coincidentes

Metodologia

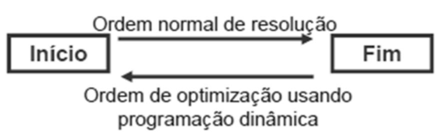
2) Definir recursivamente a solução óptima, em função de soluções óptimas de subproblemas.

– Definir recursivamente o valor da solução óptima, com rigor e exatidão, a partir de subproblemas mais pequenos do mesmo tipo
– Imaginar sempre que os valores das soluções ótimas já estão disponíveis quando precisamos deles
– Não é necessário codificar. Basta definir matematicamente a recursão.

Metodologia

3) Calcular as soluções de todos os subproblemas: "de trás para a frente"

– Descobrir a ordem em que os subproblemas são precisos, a partir dos subproblemas mais pequenos até chegar ao problema global (*bottom-up*) e codificar, usando uma tabela.
– Normalmente, esta ordem é inversa à ordem normal da função recursiva que resolve o problema



Metodologia

3) Calcular as soluções de todos os subproblemas: "*memoization*"

– Existe uma técnica, chamada **"*memoization*",** que permite resolver o problema pela ordem normal (*top-down*)
– Usar a função recursiva obtida diretamente a partir definição da solução e ir mantendo uma tabela com os resultados dos subproblemas.
– Quando queremos aceder a um valor pela primeira vez temos de calculá-lo e a partir daí basta ver qual é.
– **Exemplo:** *Linha de produção*

Metodologia

4) Reconstruir a solução ótima, baseada nos cálculos efetuados

- Pode ou não ser requisito do problema
- Duas alternativas:
  - **Diretamente** a partir da tabela dos sub-problemas.
  - **Nova tabela** que guarda as decisões em cada etapa.
- Não necessitando de saber qual a melhor solução, podemos por vezes poupar espaço.

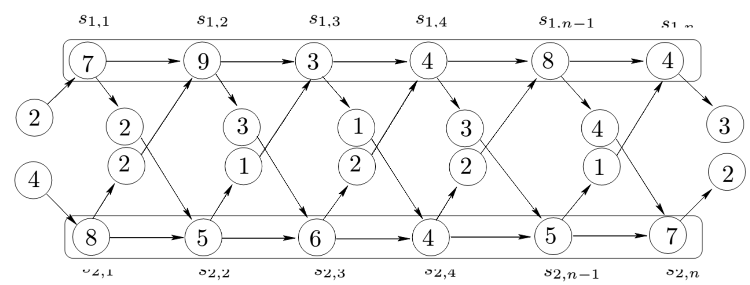
Considerações finais

- Nem sempre a PD representa a **melhor solução** para um problema, mas no entanto apresenta normalmente ganhos muito significativos sobre algoritmos exponenciais de força-bruta;
- A PD e uma **técnica** ativamente usada na vida real, tanto no meio empresarial, como no meio acadêmico;
- A **ideia base** da PD e muito simples, mas nem sempre e fácil chegar a sua solução.
  - Alguns dizem que a parte mais difícil e generalizar o problema da maneira correta, de modo que seja possível escrever a solução em função de soluções ótimas de subproblemas;
- Só existe uma maneira de dominar a PD: **hard work and practice**!

PROBLEMAS

Escalonamento de Linha de Montagem

- Um fábrica possui 2 linhas de montagem. Escolher caminho que minimiza o tempo de se produzir um carro.



Caracterizando a solução ótima

- Vamos supor que o caminho mais rápido até $S_{1,j}$ passa por $S_{1,j-1}$
  Observação chave: devemos usar o caminho mais rápido para chegar até $S_{1,j-1}$. Se houver outro caminho mais rápido até $S_{1,j-1}$ o caminho escolhido até $S_{1,j}$ não seria o mais rápido.
- Suponha que o caminho mais rápido passa por $S_{2,j-1}$ Similarmente devermos escolher o caminho mais rápido até $S_{2,j-1}$
- De uma forma geral, a solução ótima que passa por $S_{i,j}$ contém dentro dela soluções ótimas para $S_{1,j-1}$ ou $S_{2,j-1}$

Esta propriedade é chamada *Subestrutura Ótima*

Definindo a solução recursiva

Definições:

- $f_i[j]$:= tempo mínimo para chegar até $S_{i,j}$
- $f* := \min\{ f_1[n] + x_1 , f_2[n] + x_2 \}$ i.e., a Solução!
- $f_1[1] := e_1 + a_{1,1}$
- $f_2[1] := e_2 + a_{2,1}$
- Para j ≥ 2 temos:
  - $f_1[j] := \min\{ f_1[j-1] + a_{1,j} , f_2[j-1] + t_{2,j-1} + a_{1,j} \}$
  - $f_2[j] := \min\{ f_2[j-1] + a_{2,j} , f_1[j-1] + t_{1,j-1} + a_{2,j} \}$

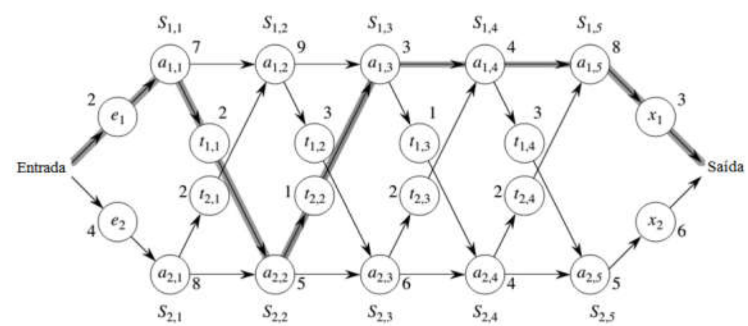
$f_i[j]$ provê o valor da solução ótima para chegar até $S_{i,j}$

Construindo o caminho da solução

Definições:

- $l_i[j]$:= linha (1 ou 2) cuja estação j−1 é usada no caminho mais rápido até $S_{i,j}$
  - Em outras palavras, temos que $S_{l_i[j],j-1}$ precede $S_{1,j}$ para j ≥ 2
- l^* := linha (1 ou 2) cuja estação n é usada.
- Combinando temos a equação recursiva:

$f_{1,1} = e_1 + a_{1,1}$, se j = 1
$f_{1,j} = \min\{ f_1[j-1] + a_{1,j} , f_2[j-1] + t_{2,j-1} + a_{1,j} \}$, se j ≥ 2
$f_{2,1} = e_2 + a_{2,1}$, se j = 1
$f_{2,1} = \min\{ f_2[j-1] + a_{2,j} , f_1[j-1] + t_{1,j-1} + a_{2,j} \}$, se j ≥ 2



j	1	2	3	4	5
$f_1[j]$	9	18	20	24	32
$f_2[j]$	12	16	22	25	30

$f^* = 35$

j	2	3	4	5
$l_1[j]$	1	2	1	1
$l_2[j]$	1	2	1	2

$l^* = 1$