Programação Dinâmica

Fabio Lubacheski
fabio.lubacheski@mackenzie.br

Desafio

Escreva um algoritmo que calcula o caminho, 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.

```
        7

      3   8

    8   1   0

  2   7   4   4

4   5   2   6   5
```

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

Desafio – Algoritmo guloso

Para resolver o problema temos algumas possibilidades:

- Utilizar uma **algoritmo guloso** (ganancioso):

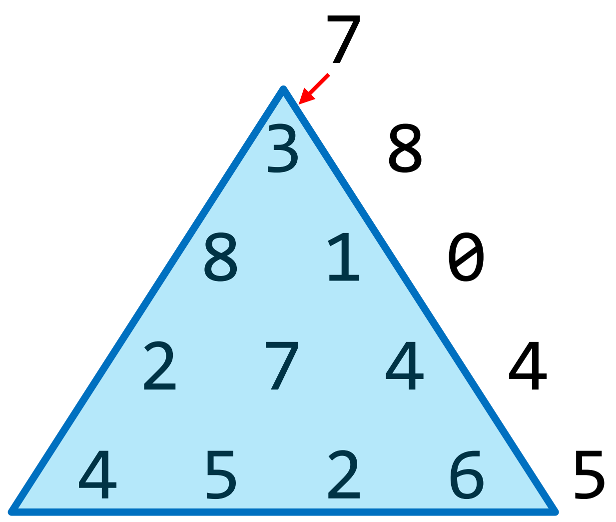
  Ideia: sempre escolher o maior número entre os dois “descendentes” da esquerda e direita.

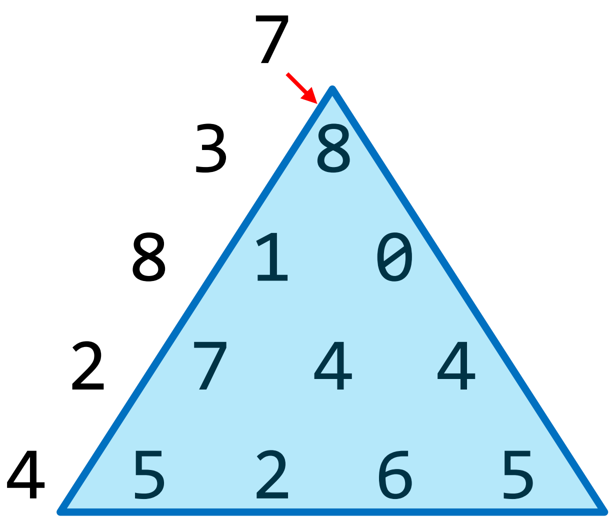
7 3 **8** 8 **1** 0 2 **7** 4 4 4 5 **2** 6 5	**7** **3** 8 **8** 1 0 2 **7** 4 4 4 **5** 2 6 5

Essa estratégia teríamos como resultado o valor **28**, mas o valor ótimo seria **30**.

Desafio – Algoritmo força-bruta

Poderíamos implementar um algoritmo utilizando força-bruta, o algoritmo em cada linha toma dois caminhos, à esquerda e depois à direita, e no final verifica a maior soma nas subpirâmides.





Desafio – Algoritmo força-bruta

Antes de implementar o algoritmo, precisamos definir como representar a pirâmide em um programa ?

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

Desafio – Algoritmo força-bruta

Antes de implementar o algoritmo, precisamos definir como representar a pirâmide em um programa ?

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

	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

Desafio – Algoritmo força-bruta

```
Algoritmo Pirâmide(P, i, j)
    Entrada: Uma matriz P[1..N,1..N] representando pirâmide, onde N <= 100, linha
      i e coluna j do elemento da pirâmide avaliado.
Saída: A maior soma do caminho do topo da pirâmide até a base.
início
    se i = N então
      retornar P[i,j]
    senão
      retornar máximo(Pirâmide(P,i+1, j), Pirâmide(P,i+1,j+1))+P[i,j]
    fim-se

fim.
```

No início da execução chamamos o algoritmo assim:

```
Pirâmide(P,1,1)
```

Complexidade do Algoritmo força-bruta

Como em cada linha o algoritmo toma **duas decisões**: esquerda ou direita.

- Seja **n** é a altura da pirâmide, um caminho terá $\mathbf{n-1}$ decisões.

- Se temos sempre dois caminhos distintos, então existem então $\mathbf{2^{n-1}}$ caminhos diferentes.

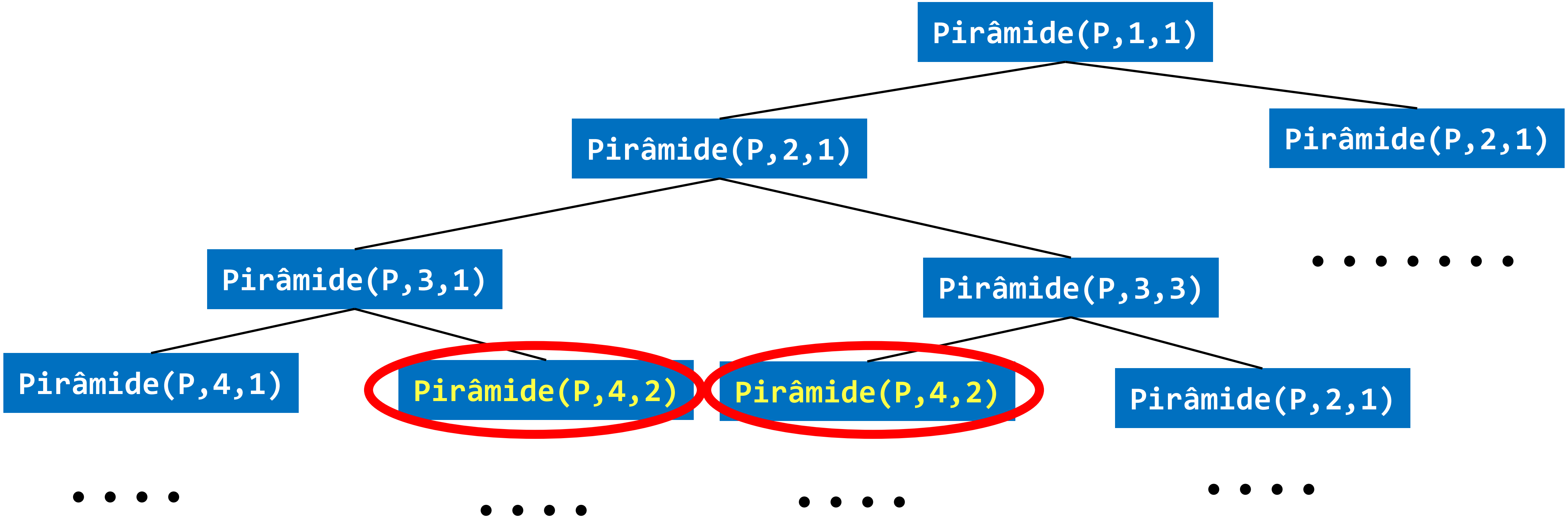
- Um programa para calcular todos os caminhos tem portanto complexidade $O(2^n)$: exponencial!

- Para a pirâmide com `n=100`, temos:

$$\mathbf{2^{99} \approx 6.34 \times 10^{29}} = 633825300114114700748351602688$$

Complexidade do Algoritmo força-bruta

O problema do algoritmo força-bruta é que o cálculo da maior soma avalia o mesmo subproblema várias vezes.



Uma outra solução

- Teria como **reaproveitar** o que já foi calculado, só calcular o mesmo problema **somente uma vez** ?
- Para tanto podemos guardar os cálculos de cada subproblema em uma tabela (Matriz Solução $S[i,j]$).
- Precisamos também definir uma ordem em que vamos preencher a tabela, de modo que quando precisarmos de um valor, este já tenha sido calculado.

$$\mathrm{S}[i,j] = \begin{cases} \mathrm{P}[i,j] & \textbf{se } i = \mathrm{N} \\ \mathrm{P}[i,j] + \mathrm{MAX}(\mathrm{S}[i+1,j], \mathrm{S}[i+1,j+1]) & \textbf{se } j \leq i < \mathrm{N} \end{cases}$$

Considere que i começa com o valor N ($i \leftarrow \mathrm{N}$) e decrementa até 1.

Agora podemos preencher a tabela $S[i,j]$.

Desafio – Algoritmo implementando a recorrência

```
Algoritmo Pirâmide(P)
    Entrada: Uma matriz P[1..N,1..N] representando pirâmide, onde N <= 100.
    Saída: A maior soma do caminho do topo da pirâmide até a base.
    Estrutura Auxiliar: tabela S[1..N,1..N] para guardar os cálculos de cada
                        subproblema
início
   para i = N até N faça
      para j = 1 até i faça
          se i == N então
             S[i,j] = P[i,j]
          senão
             S[i,j] = máximo(Pirâmide(P,i+1, j), Pirâmide(P,i+1,j+1))+P[i,j]
          fim-se
      fim-para
   fim-para
   retorne S[1,1]
fim.
```

Complexidade do Algoritmo usando a Programação Dinâmica

Para resolver o problema da pirâmide de números usamos a técnica de **Programação Dinâmica**.

- O resultado da solução fica na posição `S[1,1]` da tabela.

- Agora o tempo necessário para resolver o problema só cresce polinomialmente, como temos dois laços aninhados a complexidade do algoritmo é $O(N^2)$:

Programação Dinâmica

- A palavra **programação** na expressão **programação dinâmica** não tem relação direta com **programação de computadores**. Ela significa **planejamento** e refere-se à construção da tabela que armazena as soluções das sub-instâncias.

- A **programação dinâmica** é aplicada a problemas de **otimização combinatória**, como os **algoritmos gulosos**, também tem uma estratégia parecida com **divisão e conquista**, ou seja, combinando subproblemas.

- A programação dinâmica tem com resultado um algoritmo que é versão iterativa inteligente de um algoritmo recursivo (backtracking), ou seja, uma **recorrência** com apoio de **uma tabela**. **Clássica troca de espaço por tempo.**

Programação dinâmica versus divisão e conquista.

Divisão e conquista:

- Combina as soluções de subproblemas de forma independente; e
- Pode resolver várias vezes o mesmo problema.

Programação dinâmica:

- Se aplica quando os subproblemas podem ser usados na solução de outros subproblemas; e
- Ao contrário da **divisão e conquista**, a **programação dinâmica** resolve um subproblema apenas uma vez e armazena a solução em uma tabela.

Programação dinâmica versus método guloso

Método guloso:

- Abocanha a alternativa mais promissora (sem explorar as outras);
- É muito rápido; e
- Nunca se arrepende de uma decisão já tomada.

Programação dinâmica:

- Explora todas as alternativa (mas faz isso de maneira eficiente);
- É um pouco mais lenta que método guloso, mas muito mais rápida que o **backtracking**; e
- A cada iteração pode se arrepender de decisões tomadas anteriormente (ou seja, pode rever o ótimo local).

Maior subsequência comum – LCS

- O problema ***Longest Common Subsequence* – LCS** consiste em encontrar a maior subsequência comum dado duas sequências X e Y, a solução deste problema pode ser aplicada em problemas de compactação de arquivos e bioinformática.
- Só para lembrarmos, uma **sequência** é caracterizada pelo ordem de seus elementos, e pode ser definida por `a[1..n]`.
- Uma **subsequência** é o que sobra quando alguns termos de uma **sequência** são apagados. Assim uma subsequência de uma sequência `a[1..n]` é qualquer sequência da forma `s[1..k]` tal que:

$$s[1]=a[i_1],\ s[2]=a[i_2],\ldots,s[k]=a[i_k]$$

Desde que os índices `i` sigam a seguinte restrição $1 \leq i_1 < i_2 < \ldots\ i_k \leq n$

Maior subsequência comum – LCS – usando Força Bruta

- Dado duas sequências `X[1..m]` e `Y[1..n]`:

  `X[1..m] = {A, B, C, B, D, A, B} m = 7`

  `Y[1..n]: = {B, D, C, A, B, A}    n = 6`

  A partir da sequência X podemos ter $2^m - 1$ subsequências distintas, `{A}, {B}, {A,B}, {A,C}, {A,B,C}, .....` , ou seja, removendo alguns elementos mas não alterando a ordem.

  Para resolver o problema podemos implementar um algoritmo **força bruta** para gerar todas as subsequências de `X[1..m]`, para cada subsequência testar se a subsequência também uma subsequência de `Y[1..n]`, guardando o tamanho das subsequências comuns.

  Complexidade: Como há aproximadamente 2^m subsequências em X para serem verificas e temos Y com tamanho n, podemos dizer que a complexidade desse algoritmo é $O(n2^m)$ que é exponencial.

Maior subsequência comum – LCS – Programação Dinâmica

- Para diminuir a complexidade precisamos encontrar uma maneira de decompor o problema em subproblemas, de tal forma que a **solução ótima de todo o problema**, dependa da **solução ótima de subproblemas menores**. Essa propriedade é chamada de **subestrutura ótima**.

- A programação dinâmica armazena a **solução ótima** dos subproblemas em uma **tabela**.

- O **consumo de tempo** de um algoritmo usando **programação dinâmica** é, em geral, **proporcional ao tamanho da tabela**.

Recorrência para preencher a tabela

$$S[i,j] = \begin{cases} 0 & \textbf{se } i \textbf{ ou } j = 0 \\ S[i-1,j-1]+1 & \textbf{se } i \textbf{ e } j > 0 \; e \; X[j] = Y[i] \\ \mathrm{MAX}(S[i,j-1], S[i-1,j] & \textbf{se } i \textbf{ e } j > 0 \textbf{ e } X[j] \neq Y[i] \end{cases}$$

Onde $S[i,j]$ é uma tabela com a solução ótima dos subproblemas. Note que a primeira linha ($i = 0$) e primeira coluna ($j = 0$) de $S[i,j]$ tem valores igual a zero.

Agora é só preencher a tabela, a algoritmo que preencher a tabela gasta $O(nm)$, e para **apresentar a maior subsequência** o algoritmo gasta $O(m+n)$, bem menos que a complexidade usando força bruta.,,

Exercícios

1) Implemente, na linguagem C, o algoritmo que resolve o problema da Pirâmide usando força-bruta.

2) Implemente, na linguagem C, o algoritmo que resolve o problema da Pirâmide usando a recorrência apresentada para solução com programação dinâmica.

3) Implemente uma função, na linguagem C, que a partir da tabela tabela $S[i, j]$ a função apresenta o caminha da maior soma do topo da pirâmide até a base.

4) Execute o LCS (preencha a tabela $S[i, j]$) para as seguintes sequências X e Y

```
X[1..m]={G, A, C, C, T, G }
Y[1..n]={A, G, T, A, A, C, G, C, T, A}
```

Exercícios

5) Implemente, na linguagem C, o algoritmo que resolve o problema da LCS usando força-bruta.

6) Implemente, na linguagem C, o algoritmo LCS (programação dinâmica), que constrói a tabela $S[i,j]$.

7) Escreva uma função recursiva, na linguagem C, que imprime a maior subsequência comum (LCS) de X e Y utilizando a tabela $S[i,j]$ já previamente preenchida.

8) Apresente uma versão iterativa para a função que imprime a LCS.

9) Apresente uma função que imprime todas as subsequência comum (LCS) de X e Y utilizando a tabela $S[i,j]$ já previamente preenchida.

Fim

Fim