Programação Dinâmica

Programação Dinâmica

Projetistas de algoritmos e programadores: escrever programas que achem a melhor solução para todas as instâncias dos problemas.

Programação Dinâmica é técnica algorítmica, normalmente usada em problemas de otimização, que é baseada em guardar os resultados de subproblemas em vez de os recalcular.

Chaves Programação Dinâmica

- Obter uma sequência de decisões
- Minimizar o custo total em um número de estágios
- Compromisso entre custo imediato e futuro

Clássica Troca de Espaço por tempo!!!

Técnica algoríıtmica: método geral para resolver problemas que tem algumas características em comum.

Problema de Otimização:: encontrar a ”melhor” solução entre todas as soluções possíveis, segundo um determinado critério (função objectivo). Geralmente descobrir um máximo ou mínimo.

Resolução de Problemas

Características da Programação Dinâmica

Características que um problema deve apresentar para poder ser resolvido com Programação Dinâmica

Subestrutura Ótima

Sub Problemas Coincidentes

Subestrutura Ótima

- Quando a solução ótima de um problema contém nela própria soluções ótimas para os seus subproblemas do mesmo tipo
- Problema ”clássico” das Olimpíadas Internacionais de Informática de 1994
- Calcular 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
```

Exemplo:

No problema da pirâmides de números, a solução ótima contém nela própria os melhores percursos de sub pirâmides, ou seja, soluções ótimas de subproblemas

Subestrutura Ótima
Problema Pirâmide de Números

- **Restrições:** todos os números são inteiros entre 0 e 99 e o número de linhas do triângulo é no máximo 100.
- Dois possíveis caminhos:

Subestrutura Ótima
Problema Pirâmide de Números

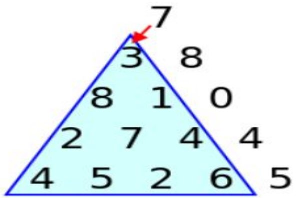
- Como resolver o problema?
- Pesquisa Exaustiva (aka ”Força Bruta”)
- Avaliar todos os caminhos possíveis e ver qual o melhor.
- Mas quanto tempo demora isto? Quantos caminhos existem?

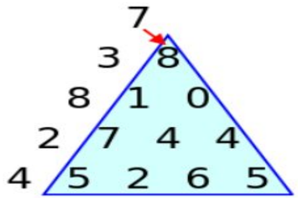
Análise da complexidade temporal:

- Em cada linha podemos tomar duas decisões: esquerda ou direita
- Seja n a altura da pirâmide. Um caminho são... n − 1 decisões!
- Existem então 2 n−1 caminhos diferentes
- Um programa para calcular todos os caminhos tem portantocomplexidade O(2 n): exponencial!
- 2 99 ~ 6.34 × 10 29 (633825300114114700748351602688)

Subestrutura Ótima
Problema Pirâmide de Números

- Quando estamos no topo da pirâmide, temos duas decisões possíveis (esquerda ou direita):
- Em cada um dos casos temos de ter em conta todas os caminhos das respectivas sub pirâmides.





Mas o que nos interessa saber sobre estas sub pirâmides?

- Apenas interessa o valor da sua melhor rota interna (que é um instância mais pequena do mesmo problema)!
- Para o exemplo, a solução é 7 mais o máximo entre o valor do melhor caminho de cada uma das sub pirâmides

Subestrutura Ótima
Problema Pirâmide de Números

- Então este problema pode ser resolvido 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

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

Subestrutura Ótima
Problema Pirâmide de Números

Definição Recursiva:

Max(i, j):
Se i = n então

retorna P[i][j]

Senão

retorna P[i][j] + máximo (Max(i + 1, j), Max(i + 1, j + 1))

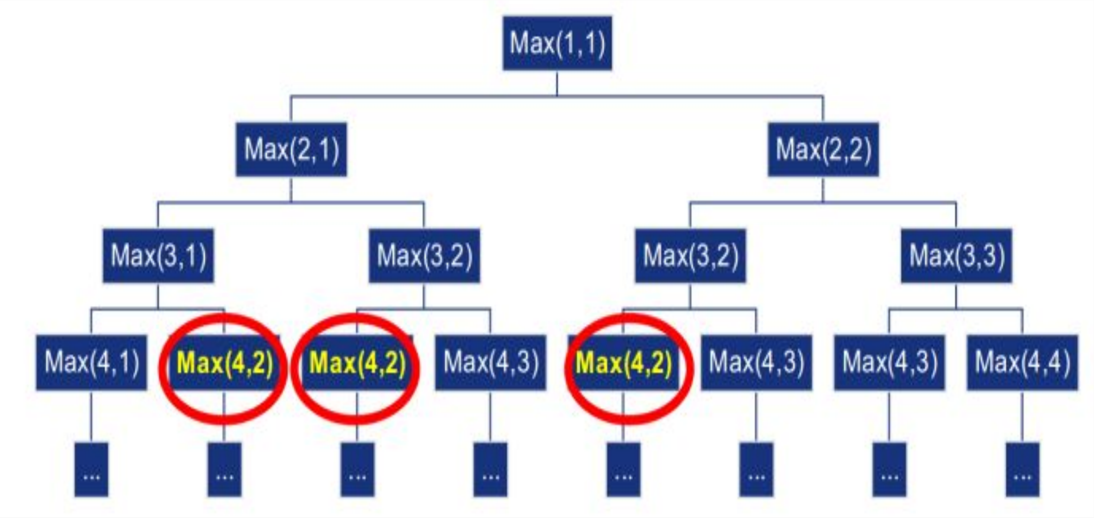
Para resolver o problema basta chamar... Max(1,1)

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

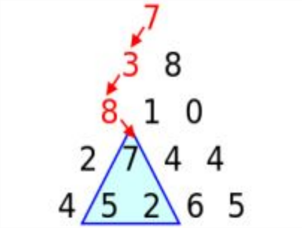
	1	2	3	4	5
1	7				
2	3	8			
3	8	1	0		
4	2	7	4	4	
5	4	5	2	6	5

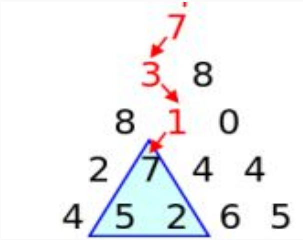
Subestrutura Ótima
Problema Pirâmide de Números

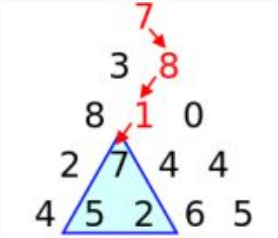
Continuamos com crescimento exponencial!



Estamos a avaliar o mesmo subproblema várias vezes..

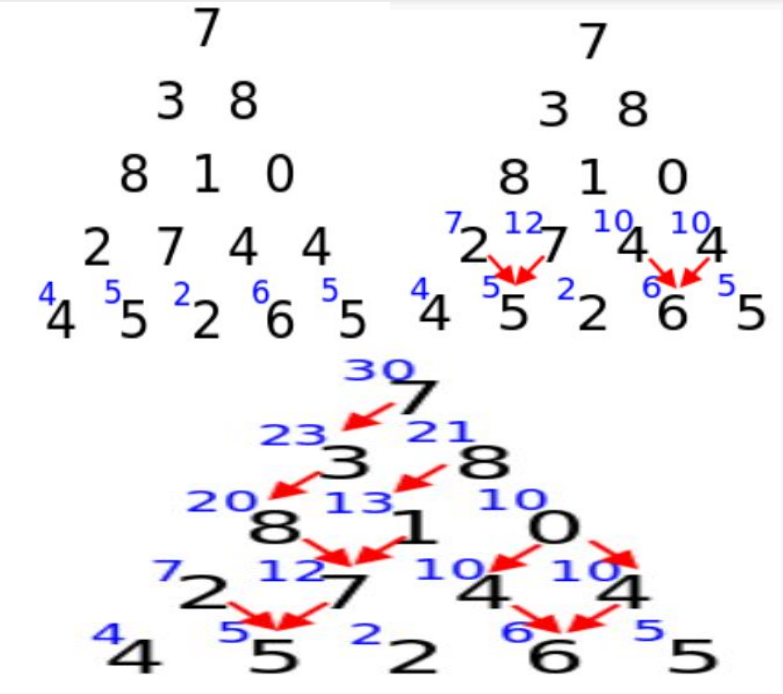






Subestrutura Ótima
Problema Pirâmide de Números

- Temos de reaproveitar o que já calculamos ->Só calcular uma vez o mesmo subproblema
  Criar uma tabela com o valor obtido para cada subproblema (Matriz M[i][j])
- Será que existe uma ordem para preencher a tabela de modo a que quando precisamos de um valor já o temos?
- Começar a partir do fim! (base da pirâmide)



Subestrutura Ótima
Problema Pirâmide de Números

- Tendo em conta a maneira como preenchemos a tabela, até podemos aproveitar P[i][j]:

- Com isto a solução fica em... P[1][1]

- Agora o tempo necessário para resolver o problema já só cresce polinomialmente (O(n 2)) e já temos uma solução admissível para o problema (99 2 = 9801)

Solução Polinominal :

Calcular():
Para i ← n − 1 até 1 fazer
Para j ← 1 até i fazer

P[i][j] ← P[i][j] + máximo (P[i + 1][j], P[i + 1][j + 1])

- E se fosse necessário saber a constituição do melhor caminho?
Basta usar a tabela já calculada!

Sub Problemas 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 MergeSort, cada chamada recursiva é feita a um subproblema novo, diferente de todos os outros.

*Dividir: Calcula o ponto médio do sub-arranjo, o que demora um tempo constante

*Conquistar: Recursivamente resolve dois subproblemas, cada um de tamanho n/2, o que contribui com 2 T (n / 2) para o tempo de execução;

*Combinar: Unir os sub-arranjos em um único conjunto ordenado, que leva o tempo.

Metodologia

Metodologia

- Se um problema apresenta estas duas características, temos uma boa pista de que a Programação Dinâmica se pode aplicar.

Guia para resolver com PD

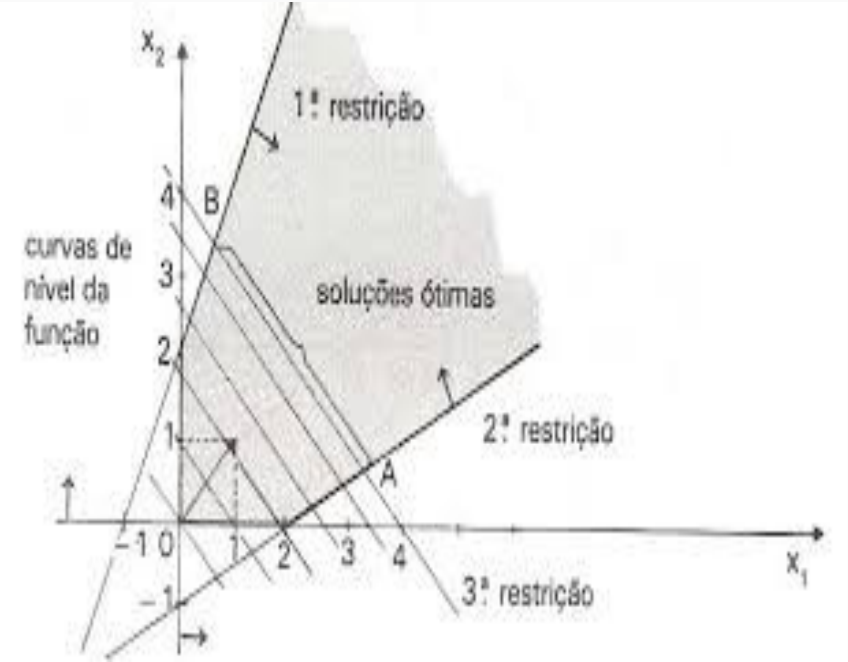
1. **Caracterizar** a solução óptima do problema
2. **Definir recursivamente** a solução óptima, em função de soluções óptimas de subproblemas
3. **Calcular** as soluções de todos os subproblemas: "de trás para a frente" ou com "memoization"
4. **Reconstruir** a solução ótima, baseada nos cálculos efectuados (opcional - apenas se for necessário)

Metodologia - Caracterizar a solução óptima do problema

- 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 correcta)
- Procurar dividir o problema em subproblemas do mesmo tipo
- Verificar se o problema obedece ao princípio de optimalidade
- Verificar se existem sub problemas coincidentes

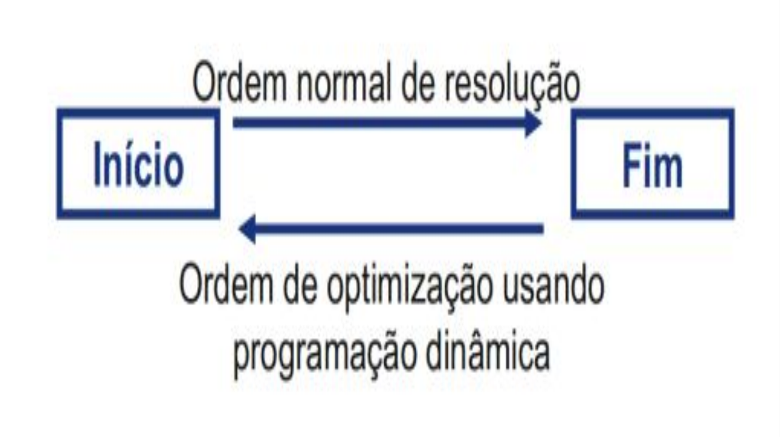
Metodologia - Definir recursivamente a solução óptima, em função de soluções óptimas de subproblemas

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



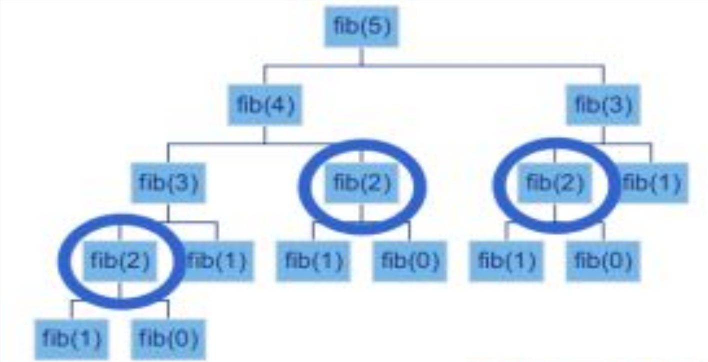
Metodologia - Calcular as soluções de todos os subproblemas: ”de trás para a frente”

- 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 - Calcular as soluções de todos os subproblemas: ”de trás para a frente”

- Existe uma técnica, conhecida como ”memoization”, que permite resolver o problema pela ordem normal (”top-down”).
- Usar a função recursiva obtida directamente a partir da 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 o resultado já calculado.



Números de Fibonacci

$F(0) = 0$

$F(1) = 1$

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

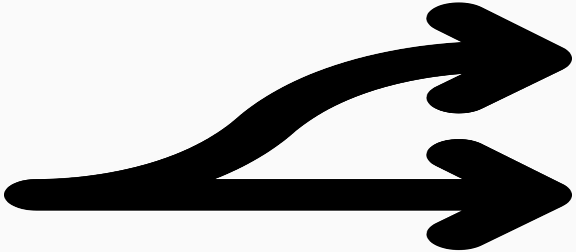
i \ j	0	1 A	2 F	3 O	4 G	5 A
0	0	1	2	3	4	5
1 G	1	1	2	3	3	4
2 O	2	2	2	2	3	4
3 T	3	3	3	3	3	4
4 A	4	3	4	4	4	3
5 S	5	4	4	5	5	4

Metodologia - Reconstruir a solução ótima, baseada nos cálculos efectuados

- Pode (ou não) ser requisito do problema, assim Duas alternativas:

1. Directamente a partir da tabela dos sub-problemas
2. Nova tabela que guarda as decisões em cada etapa

- Não necessitando de saber qual a melhor solução, podemos por vezes poupar espaço



Aplicação da Metodologia

Exemplo - Subsequência Crescente

- Dada uma sequência de números:

  7, 6, 10, 3, 4, 1, 8, 9, 5, 2

- Descobrir qual a maior **subsequência crescente** (não necessariamente contígua)

  7, 6, 10, 3, 4, 1, 8, 9, 5, 2 (Tamanho 2)
  7, 6, 10, 3, 4, 1, 8, 9, 5, 2 (Tamanho 3)
  7, 6, 10, 3, 4, 1, 8, 9, 5, 2 (Tamanho 4)

Exemplo - Subsequência Crescente

1) Caracterizar a solução óptima do problema

- Seja **n** o tamanho da sequência e **num[i]** o i-ésimo número
- "Força bruta", quantas opções? **Exponencial!** *(binomial theorem)*
- Generalizar e resolver com subproblemas iguais:
  - Seja **best(i)** o valor da melhor subsequência a partir da i-ésima posição
  - **Caso base:** a melhor subsequência a começar da última posição tem tamanho... 1!
  - **Caso geral:** para um dado i, podemos seguir para todos os números entre $i + 1$ e n, desde que sejam... maiores
    - Para esses números, basta-nos saber o melhor a partir daí! **(princípio da otimalidade)**
    - O melhor a partir de uma posição é necessário para calcular todas as posições de índice inferior! **(subproblemas coincidentes)**

Exemplo - Subsequência Crescente

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

- **n** - tamanho da sequência
- **num[i]** - número na posição i
- **best(i)** - melhor subsequência a partir da posição i

Solução recursiva para Subsequência Crescente

$$\text{best}(n) = 1$$

$$\text{best}(i) = 1 + \textbf{máximo}\{\text{best}(j) \colon i < j \leq n, num[j] > num[i]\}$$

$$\text{para } 1 \leq i < n$$

Exemplo - Subsequência Crescente

Subsequência Crescente

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

- Seja **best[]** a tabela para guardar os valores de best()

Subsequência crescente (solução polinomial)

Calcular():
  $best[n] \leftarrow 1$
  Para $i \leftarrow n - 1$ **até** 1 **fazer**
    $best[i] \leftarrow 1$
    Para $j \leftarrow i + 1$ **até** n **fazer**
      Se $num[j] > num[i]$ **e** $1 + best[j] > best[i]$ **então**
        $best[i] \leftarrow 1 + best[j]$

i	1	2	3	4	5	6	7	8	9	10
num[i]	7	6	10	3	4	1	8	9	5	2
best[i]	3	3	1	4	3	3	2	1	1	1

Exemplo - Subsequência Crescente

4) Reconstruir a solução ótima

- Vamos exemplificar com uma tabela auxiliar que guarda as decisões
- Seja **next[i]** uma próxima posição para obter o melhor a partir da posição *i* ('X' se é a última posição).

i	1	2	3	4	5	6	7	8	9	10
num[i]	7	6	10	3	4	1	8	9	5	2
best[i]	**3**	**3**	**1**	**4**	**3**	**3**	**2**	**1**	**1**	**1**
next[i]	**7**	**7**	**X**	**5**	**7**	**7**	**8**	**X**	**X**	**X**

Perguntas?

Referências

- http://wiki.icmc.usp.br/images/c/cb/SCC211Cap11.pdf
- http://www.ufjf.br/epd015/files/2010/06/ProgramacaoDinamica.pdf
- https://www.ime.usp.br/~maratona/aulas/programacao-dinamica
- https://pt.slideshare.net/OnOSJunior/programao-dinmica
- https://maratonapcauece.wordpress.com/tag/programacao-dinamica/
- https://www.pdffiller.com/jsfiller-desk5/?projectId=233666249&expId=4080&expBranch=3#cdc7f023a5eb4be19ce6d0f8e4debad3
- http://www.dca.fee.unicamp.br/~gomide/courses/IA718/transp/IA718IntroducaoProgramacaoDinamica_2.pdf