Programação Dinâmica: Introdução

Prof. Daniel Kikuti

Universidade Estadual de Maringá

Um problema inicial

Dada uma escada com n degraus, de quantas maneiras distintas podemos subir a escada se a cada passo é possível subir um ou dois degraus?

Exemplo

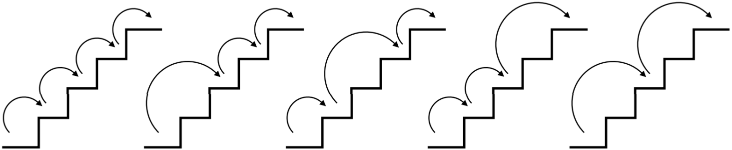


Figure 1 All five different ways to climb a four-step staircase using single and double steps.

Fonte: `http://ms.appliedprobability.org/data/files/Articles47/47-1-4.pdf`

Solução para o problema

- Seja $s(n)$ o número de maneiras distintas para subir uma escada com n degraus.
- Uma escada sem degraus pode ser subida de um único modo. Assim como uma escada com um único degrau.
- Supondo que você encontra-se no n-ésimo degrau. Para alcançá-lo, você poderia ter vindo do degrau (n-1)-ésimo ou do degrau (n-2)-ésimo.
- Podemos definir recursivamente o problema como:

$$s(n) = \begin{cases} 1 & \text{se } n = 0 \text{ ou } n = 1 \\ s(n-1) + s(n-2) & \text{se } n > 1. \end{cases}$$

Recorrência parecida com Fibonacci

Definição dos números de Fibonacci

$$f(n) = \begin{cases} 1 & \text{se } n = 1 \text{ ou } n = 2 \\ f(n-1) + f(n-2) & \text{se } n > 2. \end{cases}$$

A série

n	1	2	3	4	5	6	7	$\cdots$
$f(n)$	1	1	2	3	5	8	13	$\cdots$

Usando a recorrêcia para o problema da escada, podemos definir $f(n)$ como:

$$f(n) = s(n-1)$$

Algoritmo recursivo para o problema da escada

$\textsc{Stairs}(n)$

```
if n ≤ 1 then
    ans ← 1
else
    ans ← Stairs(n − 1) + Stairs(n − 2)
return ans
```

Complexidade?

- O: $T(n) \leq 2T(n-1) + \Theta(1) \implies T(n) =$
- Ω: $T(n) \geq 2T(n-2) + \Theta(1) \implies T(n) =$
- Θ: $T(n) = T(n-1) + T(n-2) + \Theta(1) \implies T(n) =$

Algoritmo recursivo para o problema da escada

$\textsc{Stairs}(n)$

```
if n ≤ 1 then
    ans ← 1
else
    ans ← Stairs(n − 1) + Stairs(n − 2)
return ans
```

Complexidade?

- O: $T(n) \leq 2T(n-1) + \Theta(1) \implies T(n) = O(2^n)$.
- Ω: $T(n) \geq 2T(n-2) + \Theta(1) \implies T(n) = \Omega(2^{n/2})$.
- Θ: $T(n) = T(n-1) + T(n-2) + \Theta(1) \implies T(n) = \Theta(\varphi^n)$.
  Proporção áurea: $\varphi = \frac{1+\sqrt{5}}{2} \approx 1.618$.

Algoritmo recursivo para o problema da escada

$\textsc{Stairs}(n)$

```
if n ≤ 1 then
    ans ← 1
else
    ans ← Stairs(n − 1) + Stairs(n − 2)
return ans
```

Complexidade?

- O: $T(n) \leq 2T(n-1) + \Theta(1) \implies T(n) = O(2^n)$.
- Ω: $T(n) \geq 2T(n-2) + \Theta(1) \implies T(n) = \Omega(2^{n/2})$.
- Θ: $T(n) = T(n-1) + T(n-2) + \Theta(1) \implies T(n) = \Theta(\varphi^n)$.
  Proporção áurea: $\varphi = \frac{1+\sqrt{5}}{2} \approx 1.618$.

Conclusão

Consumo de tempo **exponencial** = **péssimo**.

Introdução

- Técnica para desenvolvimento de algoritmos.
- Vários problemas aparentemente exponenciais possuem uma solução polinomial via Programação Dinâmica.
- Bastante usada em problemas de otimização (máximo/mínimo).
- Programação Dinâmica $\approx$ Força Bruta cuidadosa (exata).
- Programação Dinâmica $\approx$ Recursão $+$ Reuso.

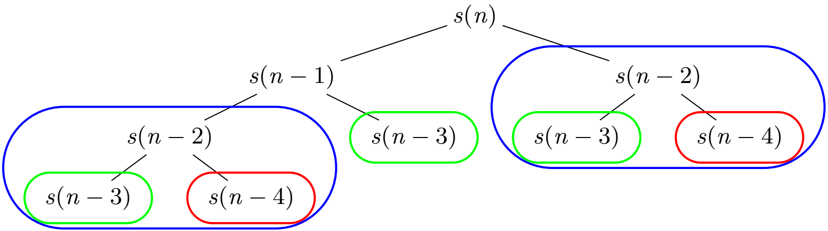
Introdução

Um resumo do que é a técnica

Programação Dinâmica (PD) é um nome “chique” para **divisão e conquista com tabela**. Ao invés de resolver subproblemas recursivamente, resolva-os sequencialmente e armazene suas soluções em uma tabela. O truque é resolvê-los na ordem certa, tal que quando uma solução para um subproblema é necessária, ela já está disponível na tabela. PD é particularmente útil em problemas nos quais Divisão e Conquista aparenta levar a um número exponencial de subproblemas, mas existe de fato um número pequeno de subproblemas repetidos em frequencia exponencial. Neste caso, faz sentido computar cada solução a primeira vez e armazená-la em uma tabela para usos posteriores, ao invés de recomputá-la recursivamente cada vez que for necessária.

PARBERY, Ian. **Problems on Algorithms**. Prentice Hall, 1995.

Subproblemas e sobreposições



Observações

- O algoritmo ingênuo faz chamadas recursivas a subproblemas repetidos, ou seja, há **sobreposição de problemas**.
- O que pode ser feito? Salvar soluções para subproblemas, ou seja, **gastar memória para salvar tempo**.

Algoritmo memoizado (Programação Dinâmica top-down)

$memo$ é uma estrutura de dados do tipo dicionário, previamente inicializada, global ou passada por como parâmetro na chamada recursiva.

```
STAIRS-MEMO(n)
if memo[n] = −1 then
    if n ≤ 1 then
        ans ← 1
    else
        ans ← STAIRS-MEMO(n − 1) + STAIRS-MEMO(n − 2)
    memo[n] ← ans
return memo[n]
```

Algoritmo memoizado

Complexidade

- Para todo k, $\textsc{Stairs-memo}(k)$ é recursivo somente na primeira chamada.
- Serão feitas n chamadas recursivas (não memoizadas).
- Custo por chamada memoizada é $\Theta(1)$ (ignora a recursão).

A complexidade de um algoritmo memoizado será

$$\begin{array}{c}\text{Complexidade}\\\text{de Tempo}\end{array} = \begin{array}{c}\text{Número de}\\\text{subproblemas}\end{array} \times \begin{array}{c}\text{Tempo gasto por}\\\text{subproblemas}\end{array}$$

Algoritmo memoizado

Complexidade

- Para todo k, $\textsc{Stairs-memo}(k)$ é recursivo somente na primeira chamada.
- Serão feitas n chamadas recursivas (não memoizadas).
- Custo por chamada memoizada é $\Theta(1)$ (ignora a recursão).

A complexidade de um algoritmo memoizado será

$$\begin{array}{c} \text{Complexidade} \\ \text{de Tempo} \end{array} = \begin{array}{c} \text{Número de} \\ \text{subproblemas} \end{array} \times \begin{array}{c} \text{Tempo gasto por} \\ \text{subproblemas} \end{array}$$

Complexidade de $\textsc{Stairs-memo}$ é $\Theta(n)$.

Programação Dinâmica (Algoritmo bottom-up)

Stairs-dp(n)

for $i \leftarrow 0$ **to** n **do**
    if $i \leq 1$ **then**
        $ans \leftarrow 1$
    else
        $ans \leftarrow memo[i-1] + memo[i-2]$
    $memo[i] \leftarrow ans$
return $memo[n]$

Complexidade de tempo da versão *bottom-up* é $\Theta(n)$.

Considerações sobre a versão *bottom-up*

- Faz exatamente as mesmas computações que a versão memoizada (recursão "desenrolada").
- Mais rápido na prática (sem custo de chamadas recursivas).
- Análise da complexidade de tempo mais fácil.
- Pode usar menos espaço (poderíamos reescrever o código de STAIRS-DP para lembrar somente dos dois valores anteriores).

Outro problema

Problema de Escalonamento de Intervalos Ponderados

Kleinberg & Tardos. **Algorithm Design**. Cap. 16.

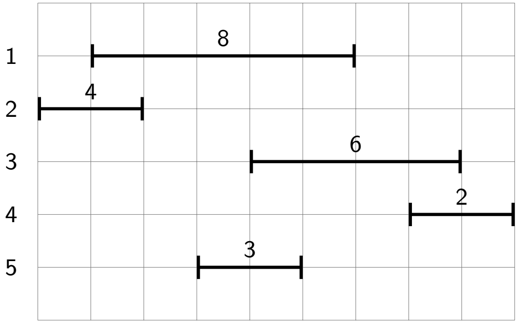
Definições

- Um **intervalo** i é um conjunto de números naturais consecutivos denotado por $[s_i, f_i]$, onde s_i e f_i são respectivamente o início e término do intervalo i ($s_i \leq f_i$).
- Um intervalo i é **anterior** a um intervalo j se $f_i < s_j$. Analogamente, i é **posterior** a j se $s_i > f_j$.
- Dois intervalos i e j são **disjuntos** se e somente se i é anterior ou posterior a j. Uma coleção de intervalos é disjunta se os intervalos da coleção são disjuntos dois a dois.
- Cada intervalo i está associado a um valor (ou peso) v_i.

O problema

Dada uma coleção A de intervalos ponderados ($v_i \geq 0$), encontrar uma subcoleção de A disjunta que tenha valor máximo.

Exemplo 1



- O intervalo 2 é anterior ao 5 (5 é posterior a 2).
- 2 e 5 são disjuntos; 1 e 2 não são disjuntos.
- O conjunto {2,4,5} é disjunto e possui valor total 9.
- **Objetivo:** selecionar um subconjunto disjunto $S \subseteq \{1, \ldots, n\}$, tal que $\sum_{i \in S} v_i$ é máximo.
- $S_1 = \{2, 3\}$ e $S_2 = \{1, 4\}$ são ótimos.

Como resolver este problema?

Força bruta descuidado

1. Gere todos os subconjuntos de intervalos possíveis.
2. Elimine aqueles que não são disjuntos.
3. Pegue aquele cuja soma dos valores é máxima.

Custo desta solução?

1. Quantos subconjuntos existem?
2. Verificar se um conjunto é disjunto.
3. Calcular a soma dos pesos.

Como resolver este problema?

Força bruta descuidado

1. Gere todos os subconjuntos de intervalos possíveis.
2. Elimine aqueles que não são disjuntos.
3. Pegue aquele cuja soma dos valores é máxima.

Custo desta solução?

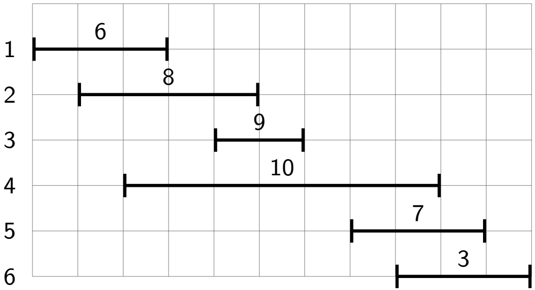
1. Quantos subconjuntos existem? $\Theta(2^n)$
2. Verificar se um conjunto é disjunto. $O(n^2)$
3. Calcular a soma dos pesos. $\Theta(n)$

Pensando em uma solução mais eficiente para o problema

Pré-processamento

- Suponha que os intervalos estão ordenados de forma não decrescente pelo tempo de término: $f_1 \leq f_2 \leq \ldots \leq f_n$.
- Seja $p(j)$, para o intervalo j, o maior índice $i < j$ tal que os intervalos i e j são disjuntos.

Exemplo 2



$p(1) = 0$
$p(2) = 0$
$p(3) = 1$
$p(4) = 0$
$p(5) = 3$
$p(6) = 3$

Definindo a solução ótima S^* em termos de subproblemas

Propriedade de **subestrutura ótima**

Um problema possui subestrutura ótima se a solução ótima para o problema contém dentro dela soluções ótimas para os subproblemas.

Partindo do fato óbvio

O intervalo n (o último) pertence ou não a S^*.

- Se $n \in S^*$: nenhum intervalo indexado entre $p(n)$ e n pode pertencer a S^*, pois pela definição de $p(n)$, sabemos que os intervalos $p(n) + 1, p(n) + 2, \ldots, n - 1$ sobrepõem n. Além disto, se $n \in S^*$, então S^* deve incluir a solução ótima S_1^* para o subproblema consistindo dos intervalos $\{1, 2, \ldots, p(n)\}$, pois caso não incluísse, poderíamos substituir S_1^* por uma melhor sem o perigo de sobrepor n.
- Se $n \notin S^*$: então S^* é igual a solução ótima para os intervalos $\{1, \ldots, n - 1\}$ (raciocínio análogo ao anterior).

Solução recursiva para o problema

- Seja $opt(n)$ o valor da solução ótima para $S = \{1, 2, \ldots, n\}$.
- $opt(0) = 0$ (valor ótimo para o conjunto vazio).

$$opt(n) = \begin{cases} v_n + opt(p(n)) & \text{se } n \in S_n^* \\ opt(n-1) & \text{se } n \notin S_n^*. \end{cases}$$

Portanto, o subconjunto disjunto de valor máximo terá valor total dado por:

$$opt(n) = \max\left(v_n + opt(p(n)), opt(n-1)\right).$$

Exercícios

1. Escreva um algoritmo recursivo (sem usar PD) para resolver o problema dos intervalos ponderados. Mostre que o algoritmo está correto.
2. Faça a árvore de recursão para o Exemplo 2. Quantos problemas distintos precisam ser resolvidos?
3. Qual a complexidade deste algoritmo no pior caso (apresente um exemplo de quando ocorre).
4. Apresente um algoritmo memoizado e outro *bottom-up*.