Exercícios 3

Davi Juliano Ferreira Alves - RA: 133595
Projeto e Análise de Algoritmo

26 de julho de 2021

Exercício 1 - Considere o problema de retornar n centavos de troco com o número mínimo de moedas.

a) Seja T o valor a ser retornado e o seguinte $D = \{25, 10, 5, 1\}$ o conjunto de diferentes valores de moedas disponíveis. Para esse conjunto de possíveis moedas, é possível projetar um algoritmo guloso que encontre a quantidade mínima de moedas para retornar o troco T? Justifique. Caso possível, forneça um algoritmo guloso que encontre a solução ótima.

Resolução: Suponha que o troco T a ser retornado pode ser reescrito pela base de inteiros dada por $D = \{25, 10, 5, 1\}$, ou seja, $T = 25.d + 10.c + 5.b + 1.a$, para $a, b, c, d \in \mathbb{Z}$. Queremos provar que a quantidade mínima de moedas é sempre retornada pelo algoritmo guloso que é descrito como:

- Ordenar a base inteira de moedas.
- Inicializar o conjunto solução como vazio.
- Encontrar o maior valor da base que seja menor que o valor solicitado.
- Adicionar o valor da base escolhido no conjunto solução.
- Subtrair do valor solicitado a quantidade escolhida da base.
- Repetir os três itens anteriores até zerar o valor socilitado, e printar o conjunto solução com todos os valores particionados na base de inteiros.

Analisando a base de inteiros $D = \{25, 10, 5, 1\}$, sabemos que essa base de inteiros é uma base canônica de inteiros, de acordo com [1], quando pode ser dividida em duas bases de inteiros que também são canônicas, e essa divisão se dá por $C]_{c_k} = \{c_1, ..., c_k\}$ e ${}_{c_k}[C = \{c_k/c_k = 1, c_{k+1}/c_k, ..., c_n/c_k\}$. Uma base de inteiros é definida canônica se, para qualquer $x \in \mathbb{Z}$ e para os valores $(a_1, ..., a_n)$ da base de inteiros, os valores de $(a_1, ..., a_n)$ é o mesmo para o algoritmo guloso e para a solução ótima.

Definimos (a, b, c, d) na decomposição de T, e usaremos a divisão da base de inteiros para provar que D é canônico. Podemos dividir D como $D]_5 = \{1, 5\}$ e ${}_5[D = \{1, 2, 5\}$ e ambos são canônicos, pois, em ${}_5[D$, podemos escrever $\{1 = (1, 0, 0), 2 = (0, 1, 0), 3 = (1, 1, 0), 4 = (0, 2, 0), 5 = (0, 0, 1), 6 = (1, 0, 1), 7 = (0, 1, 1), 8 = (1, 1, 1), 9 = (0, 2, 1), 10 = (0, 0, 2), ...\}$, e, se repete sucessivamente, e em $D]_5$, podemos escrever $\{1 = (1, 0), 2 = (2, 0), 3 = (3, 0), 4 = (4, 0), 5 = (0, 1), 6 = (1, 1), 7 = (2, 1), 8 = (3, 1), 9 = (4, 1), 10 = (0, 2), ...\}$, e assim sucessivamente. Pelo teorema dado por [1], temos que D é canônico, e portanto, o algoritmo guloso sempre retorna a solução ótima nessa base de inteiros.

b) Forneça um outro conjunto D para o qual o algoritmo guloso não encontra a solução ótima. Mostre um caso para o qual este algoritmo falha.

Resolução: Queremos uma B que não seja canônica. Tome uma base de inteiros $B = \{1, 7, 10\}$, e reescreva o troco $T = c.10 + b.7 + 1.a$. Para o valor 15, temos que, no algoritmo guloso, a trinca que a representa é dada por $(5, 0, 1) = 1.10 + 5.1$ (6 moedas), enquanto a trinca que representa o valor ótimo é dado por $(0, 2, 1) = 2.7 + 1.1$ (3 moedas). Portanto, eis o contra-exemplo no qual o algoritmo guloso falha em uma base não canônica.

c) Projete um algoritmo por programação dinâmica que encontra a quantidade mínima de moedas necessárias para devolver n centavos de troco para qualquer conjunto D que inclua a moeda de 1 centavo.

Resolução: Sabemos, por definição dada em aula, que a Programação Dinâmica é:

Definição 1. *(Programação Dinâmica) A Programação Dinâmica é um paradigma de algoritmos aplicável a uma ampla gama de problemas que pode ser subdividido em:*

- *Identificação de uma coleção de subproblemas (que são classificados como logo abaixo) do problema maior;*
  - *Subestrutura ótima: as soluções ótimas do problema incluem soluções ótimas de subcasos;*
  - *Sobreposição de subproblemas: O cálculo da solução por recursão implica no recálculo de subproblemas;*
- *Resolução dos menores primeiro;*
- *Uso das respostas aos menores problemas para responder problemas maiores (Memorização);*
- *Duas formas de se aplicar a Programação Dinâmica (Top-down recursivo + memorização e Bottom-up);*

Dada a definição como nas notas de aula, teremos de definir como resolver o problema do troco usando a Programação Dinâmica.

Para resolver, será dada a base D de inteiros formada por n números que formarão o T troco. Para isso, será criado uma matriz de tamanho $T + 1$ e n, ou seja, ele terá índices $(0, 1, 2, ..., T, T+1)$. Portanto, essa matriz será preenchida da quantidade de valores que cada uma das moedas conseguem preencher, utilizando da memorização dos valores passados, tendo a primeira moeda como base. Portanto, é dado o pseudo-código abaixo (Note que zeraremos a primeira coluna, pois será a coluna de troco 0 da linguagem):

Algorithm 1: Psudo-código do Problema do Troco em Programação Dinâmica

Input : Base D de inteiros e um troco T
Output: Quantidade mínima de moedas
Inicializacao.
Criação da matriz $M_{n\times(T+1)}$.
Zerar a primeira coluna de $M_{n\times(T+1)}$
Completar a primeira linha (da *moeda inicial*) com os valores fundamentais.

```
for Percorrer as i linhas de M do
    for Percorrer as j colunas de M do
        if Moeda i > j then
            O a_ij = a_(i-1)j.
        end
        else
            O a_ij = min{a_(i-1)j, 1 + a_(i(j-Moeda i))}
        end
    end
end
```

Após o algoritmo do problema do troco, a ultima linha e ultima coluna da matriz $M_{n\times(T+1)}$ é a quantidade mínima das moedas do problema.

Exercício 2 - Um palíndromo é uma palavra, frase ou qualquer sequência de caracteres que pode ser lida tanto da esquerda para a direita quanto da direita para a esquerda. Por exemplo, "arara"é um palíndromo, enquanto "araras"não é. Escreva um algoritmo de programação dinâmica para encontrar o palíndromo mais longo que é uma subsequência de uma dada sequência de caracteres. Assim, dada a entrada "character", seu algoritmo deve retornar "carac". Faça uma análise do consumo de tempo de sua solução.

Resolução: Como já foi definido o que é a programação dinâmica no exercício passado, passaremos a resolver o problema do palíndromo. Para resolver o problema do palíndromo, iremos fazer uma matriz $M_{n\times n}$ de resultados booleanos de *True* ou *False*, e, para resultados verdadeiros, a substring é um palíndromo se o resultado é verdadeiro.

As linhas e colunas de M são dadas pela string dada. Portanto, construiremos a matriz que será preenchida de uma maneira *bottom-up*. Ou seja, a_{ij} só é verdadeiro se $a_{(i+1)(j-1)}$ é verdadeiro e se a string na posição i é a mesma da string na posição j, e não será verdadeiro, se as condições acima não forem verdadeiros.

Como o menor palíndromo é gerado pela substring de tamanho 1, toda a diagonal principal de M é retida de 1, portanto, precisamos inicializá-la com a diagonal principal = 1. Como o nosso critério segue o valor de $a_{(i+1)(j-1)}$ ser verdadeiro, temos que para o tamanho igual a 2 da substring, temos que, para $i = 2$ e $j = 3$, $i + 1, j - 1$ não falam nada sobre i e j, ou seja, deve-se feito na mão, ou seja, comparação 2 a 2 nas strings, para o caso de quando a substring é de tamanho 2 no preenchimento da matriz.

Para os demais valores da tabela, utilizaremos o critério definido primordialmente. Portanto, temos o seguinte pseudo-código:

Algorithm 2: Psudo-código do Problema do Palíndromo em Programação Dinâmica

Input : Uma string S de tamanho n
Output: Menor palíndromo possível de S
Inicializacao.
Criação da matriz $M_{n \times n}$.
Preencher M na diagonal principal com *TRUE*.
for $i < n - 1$ **do**
  if $S(i) = S(i+1)$ **then**
    $a_{i(j+1)} = TRUE$
  end
end
for *Percorrer as k substrings maiores iguais a 3 até n* **do**
  for *Percorrer os i índices até $n - (k+1)$* **do**
    $j = i + k - 1$
    if $a_{(i+1)(j-1)} = TRUE$ *e* $S(i) = S(j)$ **then**
      $a_{ij} = TRUE.$
    end
  end
end
Verificar na matriz preenchida qual é a maior substring marcada por *TRUE*.

Analisando o código, temos que o algoritmo possui dois laços de repetição encadeados, e dois outros laços de tamanho n que inicializam a matriz booleana, ou seja, a função que aproxima a complexidade do algoritmo é dada por:

$$T(n) = n^2 + 2.n \leq n^2 + 2.n^2 = 3.n^2$$
$$\Rightarrow T(n) = O(n^2)$$

Exercício 3 - Você precisa dirigir um carro de uma cidade A até uma cidade B. Um tanque cheio do carro possui autonomia para viajar m quilômetros e você sabe as distâncias, a partir de A, dos postos de combustível existentes no caminho. Sejam $d_1 < d_2 < ... < d_n$ as distâncias dos n postos do caminho. Você deve encontrar onde você deve parar para abastecer o carro de forma a fazer o menor número de paradas para chegar até B.

a) Forneça um algoritmo guloso para resolver o problema.

Resolução: Antes de definir o algoritmo guloso, definiremos o problema das distâncias dado.

Seja A e B pontos representando a cidade. Suponha que existem n postos de gasolina na distância entre A e B. O tanque do carro pode viajar uma quantidade m de quilômetros antes de reabastecer m quilômetros em um posto.

A tática gulosa consiste em buscar um mínimo local na primeira iteração, ou seja, vai ser buscado a distância d, que consiste em, seja $d_1, d_2, ..., d_j < m$, $d = \max\{d_1, d_2, ..., d_j\}$. Após encontrar o d, será retornado o posto associado a d. Portanto, a técnica do algoritmo guloso executa as seguintes operações:

1. Verifica o mais distante posto que m pode percorrer de A;

2. Abastece o tanque no posto mais distante p_j;
3. Restabelece o posto p_j como um novo ponto A
4. Refaz as operações acima

Portanto, o pseudo-código acima define a lógica do algoritmo guloso que resolve o problema dado.

b) Mostre que o algoritmo encontra a solução ótima corretamente.

Resolução: Para mostrarmos que o algoritmo encontra a solução ótima corretamente, deve-se supor uma rota ótima. Seja p_d o posto associado a maior distância no algoritmo guloso e seja p_{o1} o posto 1 da rota ótima. Se $p_d = p_{o1}$, temos que o algoritmo guloso é da rota ótima.

Suponha agora que p_{o1} é mais próximo de A que p_d. Para essa análise, precisamos tomar o p_{o2} que é o posto 2 da rota ótima. Entretanto, p_d é mais próximo de A do que de p_{o2}, ou seja, o reabastecimento será no p_d. Mas p_d é mais próximo de p_{o2} do que mais próximo de p_{o1}, e isso significa que p_d está na rota ótima.

No segundo caso, teremos que p_{o2} é mais próximo de A do que de p_d. Nesse caso, p_d está na rota ótima, pois queremos minimizar a quantidade de postos, entretanto há 2 postos na rota ótima, sendo que podemos escolher p_d, uma contradição.

Considerando os dois casos, temos que, no primeiro caso, se p_d está mais próximo de A do que de p_{o2}, podemos reabastecer em p_d. No caso contrário, podemos deixar de reabastecer em p_{o1}, o que contradiz a rota mínima estabelecida. Portanto, estabelecemos o caminho ótimo por meio dos postos p_d do algoritmo guloso.

Referências

[1] Xuan Cai. Canonical coin systems for change-making problems. In *2009 Ninth International Conference on Hybrid Intelligent Systems*, volume 1, pages 499–504. IEEE, 2009.