

Paradigmas de Resolução de Problemas

Programação Dinâmica: Problema do Troco

Prof. Edson Alves – UnB/FGA

1. Definição
2. Bases canônicas

Definição

Problema do Troco

Seja $C = \{c_1, c_2, \dots, c_N\}$ uma sequência ordenada de N inteiros positivos distintos e M um inteiro positivo. O problema do troco consiste em determinar um vetor de inteiros não-negativos $x = \{x_1, x_2, \dots, x_N\}$ tal que

$$M = \sum_{i=1}^N x_i c_i$$

e que a soma

$$S = \sum_{i=1}^N x_i$$

seja mínima.

Características do problema do troco

- Os elementos do conjunto C são denominados *moedas*
- M é o *troco*
- O problema pode ser definido informalmente como: *Qual é o menor número de moedas necessárias para dar o troco M ?*
- Se $c_1 = 1$, há solução para qualquer M
- Se C é o conjunto de moedas utilizadas no sistema financeiro da maioria dos países, o problema do troco pode ser resolvido por meio de um algoritmo guloso

Algoritmo guloso para o problema do troco

- O algoritmo guloso para o problema do troco escolhe, dentre as moedas, a maior delas que é menor ou igual a M (assuma que esta seja a moeda c_k)
- Em seguida, ele atribui a x_k o valor M/c_k e subtrai de M o valor $x_k c_k$
- O algoritmo então prossegue até que M se torne igual a zero
- Para todos os valores x_i não atribuídos durante o algoritmo, vale que $x_i = 0$

Visualização do algoritmo guloso para o problema do troco

i	c_i	x_i
6	50	
5	25	
4	10	
3	5	
2	2	
1	1	

$$M = 73$$

Visualização do algoritmo guloso para o problema do troco

i	c_i	x_i
6	50	1
5	25	
4	10	
3	5	
2	2	
1	1	

$$M = 23$$

Visualização do algoritmo guloso para o problema do troco

i	c_i	x_i
6	50	1
5	25	0
4	10	
3	5	
2	2	
1	1	

$$M = 23$$

Visualização do algoritmo guloso para o problema do troco

i	c_i	x_i
6	50	1
5	25	0
4	10	2
3	5	
2	2	
1	1	

$$M = 3$$

Visualização do algoritmo guloso para o problema do troco

i	c_i	x_i
6	50	1
5	25	0
4	10	2
3	5	0
2	2	
1	1	

$$M = 3$$

Visualização do algoritmo guloso para o problema do troco

i	c_i	x_i
6	50	1
5	25	0
4	10	2
3	5	0
2	2	1
1	1	

$$M = 1$$

Visualização do algoritmo guloso para o problema do troco

i	c_i	x_i
6	50	1
5	25	0
4	10	2
3	5	0
2	2	1
1	1	1

$$M = 0$$

Implementação do algoritmo guloso para o problema do troco

```
1 #include <bits/stdc++.h>
2
3 using namespace std;
4
5 vector<int> coin_change(int M, const vector<int>& cs)
6 {
7     int N = (int) cs.size();
8     vector<int> xs(N);
9
10    for (int i = N - 1; i >= 0; --i)
11    {
12        xs[i] = M / cs[i];
13        M -= (xs[i] * cs[i]);
14    }
15
16    return xs;
17 }
```

Implementação do algoritmo guloso para o problema do troco

```
19 int main()
20 {
21     vector<int> cs { 1, 2, 5, 10, 25, 50 };
22     int M;
23
24     cin >> M;
25
26     auto xs = coin_change(M, cs);
27
28     for (size_t i = 0; i < cs.size(); ++i)
29         cout << cs[i] << ": " << xs[i] << '\n';
30
31     cout << accumulate(xs.begin(), xs.end(), 0) << " moedas\n";
32
33     return 0;
34 }
```

Incorretude do algoritmo guloso

- O algoritmo guloso para o problema do troco, porém, não produz a solução correta para todas as entradas possíveis
- Por exemplo, se $C = \{1, 4, 5\}$ e $M = 8$, o algoritmo guloso retornaria 4 moedas: uma de 5 e três de 1
- Contudo, é possível dar um troco de 8 com apenas duas moedas de 4
- Assim, para obter a solução correta para qualquer troco e qualquer conjunto de moedas, é preciso utilizar um algoritmo baseado em outro paradigma que não o guloso

Algoritmo de programação dinâmica

- A programação dinâmica pode ser utilizada para desenvolver um algoritmo correto para o problema do troco
- Seja $c(m)$ o mínimo de moedas necessárias para um troco igual a m
- O caso base acontece quando $m = 0$: neste caso, $c(0) = 0$
- As transições acontecem para cada uma das moedas $c_k \leq m$:

$$c(m) = \min\{c(m - c_{k_1}), c(m - c_{k_2}), \dots, c(m - c_{k_r})\} + 1$$

- Isto corresponde a escolher uma moeda (o termo $+1$) que seja menor ou igual ao troco e computar o troco mínimo para o restante
- São $O(M)$ estados distintos e as transições são feitas em $O(N)$
- Portanto a complexidade do algoritmo é $O(MN)$

Algoritmo de programação dinâmica para o problema do troco

```
1 #include <bits/stdc++.h>
2
3 using namespace std;
4
5 const int MAX { 1'010'000 }, oo { 1'000'000'010 };
6 int st[MAX];
7
8 int coin_change(int m, const vector<int>& cs)
9 {
10     if (m == 0)
11         return 0;
12
13     if (st[m] != -1)
14         return st[m];
15
16     auto res = oo;
17
18     for (auto c : cs)
19         if (c <= m)
20             res = min(res, coin_change(m - c, cs) + 1);
```

Algoritmo de programação dinâmica para o problema do troco

```
22     st[m] = res;
23     return res;
24 }
25
26 int main()
27 {
28     memset(st, -1, sizeof st);
29
30     int N, M;
31     cin >> N >> M;
32
33     vector<int> cs(N);
34
35     for (int i = 0; i < N; ++i)
36         cin >> cs[i];
37
38     cout << coin_change(M, cs) << '\n';
39
40     return 0;
41 }
```

Implementação *bottom-up*

- O estado e as transições apresentadas anteriormente permite uma implementação *bottom-up* do algoritmo de programação dinâmica para o problema do troco
- O natural seria um laço externo, com o troco m variando de 0 a M , e um laço interno, avaliando as N moedas
- Embora a complexidade permaneça a mesma da implementação *top-down*, esta ordem não é favorável à *cache*, por conta dos diferentes saltos associados às moedas
- É possível melhorar a performance em tempo de execução invertendo os laços: para cada moeda, deve-se avaliar todos os trocos possíveis

Implementação *bottom-up* para o problema do troco

```
1 #include <bits/stdc++.h>
2
3 using namespace std;
4
5 const int MAX { 1'010'000 }, oo { 1'000'000'010 };
6 int st[MAX];
7
8 int coin_change(int M, const vector<int>& cs)
9 {
10     for (int m = 1; m <= M; ++m)
11         st[m] = oo;
12
13     st[0] = 0;
14
15     for (auto c : cs)
16         for (int m = c; m <= M; ++m)
17             st[m] = min(st[m], st[m - c] + 1);
18
19     return st[M];
20 }
```

Implementação *bottom-up* para o problema do troco

```
22 int main()
23 {
24     int N, M;
25     cin >> N >> M;
26
27     vector<int> cs(N);
28
29     for (int i = 0; i < N; ++i)
30         cin >> cs[i];
31
32     cout << coin_change(M, cs) << '\n';
33
34     return 0;
35 }
```

Bases canônicas

Definição

- Embora não seja um correto, o algoritmo guloso produz o resultado correto para todos os trocos possíveis para certas bases de moedas C
- Seja $G(m)$ e $D(m)$ o mínimo de moedas para o troco m computados pelo algoritmo guloso e pelo algoritmo de programação dinâmica, respectivamente
- Uma base C é dita canônica se $G(m) = D(m)$ para todos os trocos inteiros não-negativos m
- Qualquer base $C = \{1, c_2\}$ é canônica
- Se C é canônica, o ganho de performance obtido em utilizar o algoritmo guloso é notável ($O(N)$, ao contrário da complexidade $O(NM)$ do algoritmo de programação dinâmica)

- Considere uma base $C = \{1, c_2, c_3, \dots, c_N\}$ não-canônica
- Um contraexemplo m é um inteiro positivo tal que $G(m) > D(m)$
- Xuan Cai apresenta vários resultados relativos à bases não-canônicas e contraexemplos em seu artigo "*Canonical Coin Systems for Change-Making Problems*", de 2009
- Dentre eles há um teorema que reduz os possíveis contraexemplos ao intervalo $(c_3 + 1, c_{N-1} + c_N)$
- Outro resultado provado no artigo é que $C = \{1, c_2, c_3\}$ é não-canônica se, somente se, $0 < r < c_2 - q$, onde $c_3 = qc_2 + r$, com $r \in [0, c_2 - 1]$

Verificação de canonicidade para $N \leq 5$

- Um teorema importante associa as bases com $N = 3$ com todas as demais: se $C = \{1, c_2, c_3\}$ é não-canônica, então a base $C = \{1, c_2, c_3, \dots, c_N\}$, com $N \geq 4$, também será não-canônica
- É provado também que as bases $C = \{1, c_2, c_3, c_4\}$ são não-canônicas se elas satisfazem exatamente uma das condições abaixo:
 1. $C = \{1, c_2, c_3\}$ é não-canônica
 2. $G((k+1)c_3) > k+1$, onde $kc_3 < c_4 < (k+1)c_3$
- Bases $C = \{1, c_2, c_3, c_4, c_5\}$ serão não-canônicas se, e somente se, satisfazem exatamente uma das condições abaixo:
 1. $C = \{1, c_2, c_3\}$ é não-canônica
 2. $C \neq \{1, 2, c_3, c_3 + 1, 2c_3\}$
 3. $G((k+1)c_4) > k+1$, com $kc_4 < c_5 < (k+1)c_4$

Algoritmo $O(N^3)$ para verificação de canonicidade

- David Pearson, em seu artigo “*A Polynomial-time Algorithm for the Change-Making Problem*”, de 1994, apresentou um algoritmo $O(N^3)$ para a identificação do menor contraexemplo, se existir
- O algoritmo elenca $O(N^2)$ possíveis candidatos à menor contraexemplo, por meio da observação de uma relação não-trivial entre a solução gulosa $G(c_i - 1)$ e o menor contraexemplo possível
- Cada candidato pode ser verificado em $O(N)$, de modo que a solução tem complexidade $O(N^3)$

Algoritmo $O(N^3)$ para verificação de canonicidade

- Seja $\mu = (m_1, m_2, \dots, m_N)$ a solução ótima para o menor contraexemplo μ e (x_1, x_2, \dots, x_N) a solução gulosa para $(c_i - 1)$
- Então existe um j tal que $m_k = x_k$, se $k \in [1, j - 1]$, $m_j = x_j + 1$ e $m_r = 0$, se $r > j$
- Assim, para cada moeda c_i , há N candidatos a j
- A partir da solução gulosa, se constrói a possível solução ótima do menor representante
- Daí se assume que $\mu = \sum_k m_k c_k$
- Se $G(c_i - 1) > \sum_k m_k$, então μ será um contraexemplo para a base não-canônica $C = \{c_1, c_2, \dots, c_N\}$, com $c_1 > c_2 > \dots > c_N = 1$

Implementação do algoritmo de verificação de canonicidade

```
1 #include <bits/stdc++.h>
2
3 using namespace std;
4
5 const int oo { 2'000'000'007 };
6
7 vector<int> greedy(int x, int N, const vector<int>& xs)
8 {
9     vector<int> res(N, 0);
10
11     for (int i = 0; i < N; ++i)
12     {
13         auto q = x / xs[i];
14         x -= q*xs[i];
15
16         res[i] = q;
17     }
18
19     return res;
20 }
```

Implementação do algoritmo de verificação de canonicidade

```
22 int value(const vector<int>& M, int N, const vector<int>& xs)
23 {
24     int res = 0;
25
26     for (int i = 0; i < N; ++i)
27         res += M[i]*xs[i];
28
29     return res;
30 }
31
32 int min_counterexample(int N, const vector<int>& xs)
33 {
34     if (N <= 2)
35         return -1;
36
37     int ans = oo;
38
39     for (int i = N - 2; i >= 0; --i) {
40         auto g = greedy(xs[i] - 1, N, xs);
41         vector<int> M(N, 0);
```

Implementação do algoritmo de verificação de canonicidade

```
43     for (int j = 0; j < N; ++j)
44     {
45         M[j] = g[j] + 1;
46         auto w = value(M, N, xs);
47         auto G = greedy(w, N, xs);
48
49         auto x = accumulate(M.begin(), M.end(), 0);
50         auto y = accumulate(G.begin(), G.end(), 0);
51
52         if (x < y)
53             ans = min(ans, w);
54
55         M[j]--;
56     }
57 }
58
59 return ans == oo ? -1 : ans;
60 }
```

1. **CAI**, Xuna. *Canonical Coin Systems for Change-Making Problems*, 2009.
2. **CORMEN**, Thomas H.; **LEISERSON**, Charles E.; **RIVEST**, Ronald; **STEIN**, Clifford. *Introduction to Algorithms*, 3rd Edition, MIT Press, 2009.
3. **HALIM**, Steve; **HALIM**, Felix. *Competitive Programming 3*, Lulu, 2013.
4. **LAARKSONEN**, Antti. *Competitive Programmer's Handbook*, 2017.
5. **PEARSON**, David. *A Polynomial-time Algorithm for the Change-Making Problem*, 1994.