Desafios de Programação
Programação Dinâmica

Wladimir Araújo Tavares [1]

[1]Universidade Federal do Ceará - Campus de Quixadá

30 de março de 2017

O que é programação dinâmica?

- Wikipédia: Método para resolver problemas complexos dividindo-os em subproblemas mais simples.

Passos para resolver problemas de PD

- Definir os subproblemas
- Escrever a relação de recorrência que relaciona os subproblemas
- Reconhecer e resolver os casos bases

Coin Change

- Problema: Dado n, encontre o número de diferentes maneiras de escrever n como soma de 1, 3 e 4
- Exemplo: para n=5, a resposta é 6

$$
\begin{aligned}
5 &= 1+1+1+1+1 \\
&= 1+1+3 \\
&= 1+3+1 \\
&= 3+1+1 \\
&= 1+4 \\
&= 4+1
\end{aligned}
$$

Coin Change

- Definir subproblemas
  - Seja D_n o número de maneira de escrever n como a soma de 1, 3 e 4
- Encontre a recorrência
  - Considere um possível solução $n = x_1 + \ldots + x_m$
  - Se $x_m = 1$ então a soma do resto dos termos é n-1.
  - Assim, todas as somas $(n = x_1 + \ldots + x_m)$ terminadas com $x_m = 1$ é igual a D_{n-1}

Coin Change

- Encontre a recorrência
  - $D_n = D_{n-1} + D_{n-3} + D_{n-4}$
- Resolver os casos bases
  - $D_0 = 1$
  - $D_1 = D_2 = 1$
  - $D_3 = 2$

Implementação

```
D[0] = D[1] = D[2] = 1;
D[3] = 2;
for(int i = 4; i <= n; i++){
 D[i] = D[i-1] + D[i-3] + D[i-4];
}
```

LIS

- Problema: Dado uma sequência $a[0 \dots n-1]$, encontre a maior subsequência crescente de a.
- Exemplo: a[] = $\{2,5,3,8,4,6\}$.

Subsequência crescente de a
2
2,5
2,5,8
2,5,6
2,3,8
2,3,4
2,3,4,6
5
5,8
5,6
$\vdots$

LIS

- Definir subproblemas
  - Seja C_i o tamanho da maior subsequência crescente de $a[0 \ldots i]$ que contém a_i como último elemento.
- Encontre a recorrência
  - $C_i = max\{C_j + 1 | a_j < a_i, 0 \leq j \leq i\}$
- Resolva os casos bases
  - $C_0 = 1$

Implementação

```
C[0] = 1;
for(int i = 1; i < n; i++){
    C[i]=1;
    for(int j = 0; j < i; j++){
        if( a[j] < a[i])
            C[i] = C[i] > C[j]+1 ? C[i] : C[j]+1;
    }
}
ans = 0;
for(int i = 0; i < n; i++) ans = ans > C[i] ? ans : C[i];
```

Implementação

```
set <int> st;
set <int >::iterator it;

st.clear();
for(int i = 0; i < n; i++){
    st.insert(a[i]);
    it = st.find(a[i]);
    it++;
    if( it != st.end() )
        st.erase(it);
}
ans = st.size();
```

LCS

- Problema: Dado duas strings x e y, encontre o tamanho da maior subsequência comum (LCS)
- Exemplo:
  - x: A**BC**BD**AB**
  - y: **B**D**CAB**C
  - BCAB é a maior subsequência encontrada e o seu tamanho é 4

LCS

- Defina os subproblemas
  - Seja D_{ij} o comprimento da LCS de $x[1 \dots i]$ e $y[1 \dots j]$
- Encontre a recorrência
  - Se $x_i = y_j$, então o caractere está na LCS
    - $D_{ij} = D_{i-1,j-1} + 1$
  - Caso contrário, x_i ou y_j não contribuem para o LCS, então ele pode ser removido
    - $D_{ij} = max(D_{i-1,j}, D_{i,j-1})$
- Resolva os casos bases:
  - $D_{i0} = D_{0j} = 0$

Implementação

```
int D[1001][1001];

n = strlen(x);
m = strlen(y);

for(int i = 0; i <= n; i++) D[i][0] = 0;
for(int j = 0; j <= m; j++) D[0][j] = 0;

for(int i = 1; i <= n; i++)
    for(int j = 1; j <= m; j++)
        if( x[i-1] == y[j-1] )
            D[i][j] = D[i-1][j-1] + 1;
        else
            D[i][j] = max(D[i-1][j], D[i][j-1]);
```

LCS

	ε	B	D	C	A	B	C
ε	0	0	0	0	0	0	0
A	0	0	0	0	1	1	1
B	0	1	1	1	1	2	2
C	0	1	1	2	2	2	3
B	0	1	1	2	2	3	3
D	0	1	2	2	2	3	3
A	0	1	2	2	3	3	3
B	0	1	2	2	3	4	4

Reduzindo os requisitos de memória

```
int D[2][1001];
int ans;
n = strlen(x);
m = strlen(y);

for(int j = 0; j <= m; j++) D[0][j] = 0;

for(int i = 1; i <= n; i++){
    int ii = i&1;
    D[ii][0] = 0;
    for(int j = 1; j <= m; j++){
        if( x[i-1] == y[j-1] )
            D[ii][j] = D[1-ii][j-1] + 1;
        else
            D[ii][j] = max(D[1-ii][j], D[ii][j-1]);
    }
}
ans = D[n&1][m];
```

Variações

1. Resolva o problema LIS usando LCS.
2. Dado duas strings x e y encontre o comprimento da menor string z tal que x e y são subsequência de z.
3. Dado duas strings s1 e s2 e as operações INSERÇÃO, REMOÇÃO e SUBSTITUIÇÃO pode ser executada na string s1. Encontre o número mínimo de operações necessárias para converter s1 em s2.

Subset sum

- Problema: Dado um conjunto de n números a_i cuja a soma é L e $K \leq L$. Existe um subconjunto de números a_i cuja soma é K?
- Exemplo: a[] = $\{1,3,4,7\}$

K	RESPOSTA	PROVA
8	SIM	$\{1,7\}$
9	NÃO	
10	SIM	$\{7,3\}$

Subset Sum

- Defina os subproblemas
  - Seja $M_{ij} = \begin{cases} 1, & \text{se existe um subconjunto } a[1\dots i] \text{ cuja soma é j} \\ 0, & \text{caso contrário} \end{cases}$
- Encontre a recorrência
  - Seja $M_{ij} = M_{i-1,j} \quad || \quad M_{i-1,j-a_i}$
- Resolva os casos bases:
  - $M_{i0} = 1, 0 \leq i \leq n$
  - $M_{0j} = 0, 1 \leq j \leq K$

Implementação

```
for(int i=0; i <= n; i++) M[i][0] = 1;
for(int j=1; j <= K; j++) M[0][j] = 0;

for(int i = 1; i <= n; i++){
    for(int j = 1; j <= K; j++){
        if( j < a[i-1])
            M[i][j] = M[i-1][j];
        else
            M[i][j] = M[i-1][j] || M[i-1][j-a[i-1]];
    }
}

ans = M[n][K];
```

Tabela de subproblemas

	0	1	2	3	4	5	6	7	8	9
0	1	0	0	0	0	0	0	0	0	0
1	1	1	0	0	0	0	0	0	0	0
2	1	1	0	1	1	0	0	0	0	0
3	1	1	0	1	1	1	0	1	1	0
4	1	1	0	1	1	1	0	1	1	0

Implementação

```
for(int i=0; i <= K; i++) m[i] = 0;
m[0] = 1;
for(int i = 0; i < n; i++){
    for(int j = K; j >= a[i]; j--){
        m[j] = m[j] | m[j-a[i]];
    }
}
ans = m[K];
```

Variações

1. Suponha que a_i represente o número de doces de caixa. Você quer dividir o mais justo possível entre duas crianças.
2. Cada a_i pode ser usado mais de uma vez para alcançar um valor K.
3. Suponha que a_i represente moedas, você quer minimizar o número de moedas para dar o troco de valor K.
4. Você quer dividir os doces de maneira mais justa possível entre três crianças. [Dica: Resolva o seguinte subproblema m[b][c] indica se podemos dividir os doces tal que a primeira criança recebe b e a segunda recebe c].
5. Problema da mochila com repetição de objetos.

Maior substring palindrome

- Problema: Dado uma string, encontre o tamanho da maior substring que é palindrome.
- Por exemplo, se a string é "desproggorpdes", a maior string palindrome é "proggorp".

Maior substring palindrome

- Defina os subproblemas
  - Seja $D_{ij} = \begin{cases} 1, & \text{se s[i ... j] é palindrome} \\ 0, & \text{caso contrário} \end{cases}$
- Recorrência
  - Seja $D_{ij} = D_{i-1,j-1} \&\& s[i] == s[j]$
- Resolva os casos bases:
  - $D_{ii} = 1$
  - $D_{i,i+1} = s[i] == s[i+1]$
- A tabela é preenchida diagonalmente.

Execução d = 0

	a	b	b	a	a	b
a	1					
b		1				
b			1			
a				1		
a					1	
b						1

Execução d = 1

	a	b	b	a	a	b
a	1	0				
b		1	1			
b			1	0		
a				1	1	
a					1	0
b						1

Execução d = 2

	a	b	b	a	a	b
a	1	0	0			
b		1	1	0		
b			1	0	0	
a				1	1	0
a					1	0
b						1

Execução d = 3

	a	b	b	a	a	b
a	1	0	0	1		
b		1	1	0	0	
b			1	0	0	1
a				1	1	0
a					1	0
b						1

Execução d = 4

	a	b	b	a	a	b
a	1	0	0	1	0	
b		1	1	0	0	0
b			1	0	0	1
a				1	1	0
a					1	0
b						1

Execução d = 5

	a	b	b	a	a	b
a	1	0	0	1	0	0
b		1	1	0	0	0
b			1	0	0	1
a				1	1	0
a					1	0
b						1

Implementação

```
int maior_palindrome_substring(char * s){
    int n =  strlen(s);
    vector <vector <char> > D;
   D.resize(n);
    for(int i = 0; i < n; i++) D[i].assign(n, 0);
    int maxLenght = 1;
    for(int i = 0; i < n; i++) D[i][i] = 1;
    for(int i = 0; i < n-1; i++){
        if( s[i] == s[i+1] ){
            D[i][i+1] = 1;
            maxLenght = 2;
        }
    }
    for(int d = 3; d <= n; d++){
        for(int i = 0; i < n-d+1; i++){
            int j = i + d -1;
            if( D[i+1][j-1] && s[i] == s[j]){
                D[i][j] = 1;
                maxLenght = d > maxLenght ? d : maxLenght;
            }
        }
    }
    return maxLenght;
}
```

Palindrome

- Problema: Dado uma string x, encontre o número mínimo de caracteres que precisa ser inserido para que x torne-se palindrome.
- Exemplo:
  - x = "Ab3bd"
  - Inserindo dois caracteres podemos obter "dAb3bAd"ou "Adb3bdA"

1. Resolva usando programação dinâmica em intervalos.
2. Resolva usando LCS.

Multiplicações de cadeia de matrizes

- Problema: Dado uma sequência de matrizes, encontre uma maneira eficiente de multiplicar essas matrizes. O problema não é executar os multiplicações, mas apenas decidir a ordem em que as multiplicações serão executadas.
- Exemplo: ABCD
    1. (A(BC))D
    2. ((AB)C)D
    3. (AB)(CD)
    4. (A(BC)D)
    5. A(B(CD))

Multiplicação de Matrizes

- Defina os subproblemas
  - $M_{i,j}$ = número mínimo de operações de multiplicações necessárias para computar $A_i A_{i+1} \ldots A_j$ sendo que as dimensões da matriz A_i é dado por $p_{i-1} \times p_i$.
- Defina a recorrência
  - $M_{ij} = min_{\{i \leq k \leq j-1\}} M_{ik} + M_{k+1,j} + p_{i-1} p_k p_j$
- Resolva os casos bases
  - $M_{ii} = 0$

Implementação

```
#include<stdio.h>
#include<limits.h>

int MatrixChainOrder(int p[], int n){
    int m[n][n];
    for (int i=1; i<n; i++) m[i][i] = 0;
    for (int L=2; L<n; L++){
        for (int i=1; i<n-L+1; i++){
            int j = i+L-1;
            m[i][j] = INT_MAX;
            for (int k=i; k<=j-1; k++){
                int q = m[i][k] + m[k+1][j] + p[i-1]*p[k]*p[j];
                if (q < m[i][j])
                    m[i][j] = q;
            }
        }
    }
    return m[1][n-1];
}
```

Implementação

```c
#include<stdio.h>
#include<limits.h>
int NaiveMatrixChainOrder(int p[], int i, int j){
    if(i == j) return 0;
    int min = INT_MAX;
    for (int k = i; k <j; k++){
        int q = NaiveMatrixChainOrder(p, i, k) +
                NaiveMatrixChainOrder(p, k+1, j) +
                p[i-1]*p[k]*p[j];

        if ( q < min) min = q;
    }
    return min;
}

int main(){
    int arr[] = {40, 20, 30, 10, 30};
    int n = sizeof(arr)/sizeof(int);
    printf("Minimum number of multiplications is %d ",
    NaiveMatrixChainOrder(arr, 1, n-1));
    return 0;
}
```

Implementação

```
#include<stdio.h>
#include<limits.h>
#include <vector>
using namespace std;
vector < vector <int> > M;
int MemoMatrixChainOrder(int p[], int i, int j)
{
    if(i == j) return 0;
    if( M[i][j] != -1 ) return M[i][j];
    int min = INT_MAX;
    for (int k = i; k <j; k++){
        int q = MemoMatrixChainOrder(p, i, k) +
        MemoMatrixChainOrder(p, k+1, j) +p[i-1]*p[k]*p[j];
        if ( q < min) min = q;
    }
    return M[i][j] = min;
}
int main(){
    int arr[] = {40, 20, 30, 10, 30};
    int n = sizeof(arr)/sizeof(int); M.resize(n);
    for(int i = 0; i < n; i++) M[i].assign(n, -1);
    printf("Minimum number of multiplications is %d ",
                        MemoMatrixChainOrder(arr, 1, n-1));
}
```

Conjunto Independente Máximo

- Problema: dado uma árvore, encontre o maior conjunto independente nela.
- Subproblemas
  - Decida arbitrariamente a raiz da árvore.
  - B_v: solução ótima para a subárvore tendo v como raiz e incluindo v no conjunto independente.
  - W_v: solução ótima para a subárvore tendo v como raiz e não incluindo v no conjunto independente.
  - Resposta é $max\{B_r, W_r\}$

Conjunto Independente Máximo

- Recorrência
  - Se v é escolhido, então seus filhos não podem ser escolhidos

$$B_v = 1 + \sum_{u \in netos(v)} B_u \tag{1}$$

  - Se v não é escolhido, então seus filhos podem ou não serem escolhidos.

$$B_v = 1 + \sum_{u \in filhos(v)} max\{B_u, W_u\} \tag{2}$$

- Casos Bases: Folhas.

Problema do caixeiro viajante

- Problema: Dado um grafo ponderado com n vértices, encontre o menor caminho que visita todos os nós exatamente uma vez começando e terminando no vértice 0.
- Esse problema não é NP-difícil?
  - Sim, mas pode ser resolvido em $O(n2^n)$
  - O algoritmo de força bruta roda em $O(n!)$

Problema do caixeiro viajante

- Subproblema
  - $D_{S,v}$ comprimento do caminho mínimo que visita todos os vértices do conjunto S exatamente uma única vez e termina em v.
  - Existem aproximadamente $n2^n$ subproblemas
  - A resposta é $D_{V-\{0\},0}$ onde V é o conjunto de vértices.
- Recorrência:

$$D_{S,v} = min_{u \in S}(D_{S-\{u\},u} + dist[u][v]) \tag{3}$$

- Caso Base:

$$D_{\emptyset,v} = dist[0][v] \tag{4}$$

Implementação

```
#include <stdio.h>
#include<limits.h>
int n = 4;
int dist[][4] = { {0,12,11,16}, {15,0,15,10} ,
                  {8,14,0,18}, {9,11,17,0} };
int dp[1<<4][4];
int tsp(int mask, int v){
    if( mask == 0 ) return dist[0][v];
    int min = INT_MAX;
    for(int i = 0; i < n; i++){
        if( (mask & (1<<i)) != 0 ){
            int q = tsp(mask ^ (1<<i), i) + dist[i][v];
            if( q < min ) min = q;
        }
    }
    return min;
}
int main(){
    int mask = 0;
    for(int i = 0; i < n; i++) mask |= 1<<i;
    for(int i=0; i<(1<<n); i++)
        for(int j = 0; j < n; j++) dp[i][j] = -1;
    int q = tsp( mask ^ (1<<0) , 0);
}
```

Implementação

```
int tsp(int mask, int v){

    if( mask == 0 ) return dist[0][v];
    if( dp[mask][v] != -1) return dp[mask][v];
    int min = INT_MAX;
    for(int i = 0; i < n; i++)
    {
        if( (mask & (1<<i)) != 0 ){
            int q = tsp(mask ^ (1<<i), i) + dist[i][v];
            if( q < min ) min = q;
        }

    }
    dp[mask][v] =min;
    return dp[mask][v];
}

int main(){
    int mask = 0;
    for(int i = 0; i < n; i++) mask |= 1<<i;
    for(int i=0; i<(1<<n); i++)
        for(int j = 0; j < n; j++) dp[i][j] = -1;
    int q = tsp( mask ^ (1<<0) , 0);
}
```