Técnicas e Análise de Algoritmos

Programação Dinâmica - Parte 02

Professor: **Jeremias Moreira Gomes**

E-mail: jeremias.gomes@idp.edu.br

Introdução

Introdução

- Ao resolver problemas computacionais, existem diversos paradigmas que auxiliam na criação eficientes de soluções:
  - **Busca Completa**
  - Algoritmos Gulosos (visto indiretamente (MST))
  - Divisão e Conquista (visto indiretamente (mergesort))
  - **Programação Dinâmica**
  - Busca Aleatória
  - Recursividade (visto indiretamente (DFS))
  - Algoritmos Aproximados

Programação Dinâmica - Características

- A programação dinâmica é aplicável em problemas que possuem duas características:
  1. Subestrutura ótima - solução do problema pode ser formada a partir das soluções ótimas de seus subproblemas
  2. Sobreposição de subproblemas - problemas compartilham subproblemas em comum (necessita resolver um subproblema múltiplas vezes)

Programação Dinâmica - Abordagens

Característica	Top-Down	Bottom-Up
Abordagem	Recursiva	Iterativa
Execução	Chama subproblemas sob demanda	Resolve todos subproblemas
Armazenamento	Cache	Tabela preenchida progressivamente
Benefício	Mais intuitivo, fácil de implementar	Melhor controle de performance

Exemplo do Coeficiente Binomial

Coeficiente Binomial

- O coeficiente binomial $\binom{n}{m}$ é dado por $\binom{n}{m} = \frac{n!}{(n-m)!m!}$
  - Caso m > n, o coeficiente é 0
  - Caso m == 0, o coeficiente é 1
  - Caso m == n, o coeficiente é 1

Coeficiente Binomial

- Os coeficientes $\binom{i}{j}$ formam a i-ésima linha do Triângulo de Pascal, onde $0 \leq j \leq i$

```
                    1
                1       1
            1       2       1
        1       3       3       1
    1       4       6       4       1
1       5       10      10      5       1
```

Coeficiente Binomial

- O Triângulo de Pascal permite uma visualização importante em relação aos coeficientes binomiais:
  - Se m > 0 e m < n, então o coeficiente $\binom{n}{m}$ é dado pela soma de dois coeficientes da linha anterior, sendo o imediatamente acima e o seu antecessor

Coeficiente Binomial

$$\binom{n}{m} = \binom{n-1}{m} + \binom{n-1}{m-1}, \qquad \text{se } 0 < m < n$$

					1					
				1		1				
			1		2		1			
		1		3		3		1		
	1		4		6		4		1	
1		5		10		10		5		1

Se $m = 0$ ou $m = n$, então $\binom{n}{m} = 1$

Coeficiente Binomial

$$\boxed{\binom{n}{m}} = \boxed{\binom{n-1}{m}} + \boxed{\binom{n-1}{m-1}}, \qquad \text{se } 0 < m < n$$

1

1 1

1 2 1

1 3 3 1

1 4 6 4 1

1 5 10 10 5 1

Se $m = 0$ ou $m = n$, então $\binom{n}{m} = 1$

Coeficiente Binomial

$$\boxed{\binom{n}{m}} = \boxed{\binom{n-1}{m}} + \boxed{\binom{n-1}{m-1}}, \qquad \text{se } 0 < m < n$$

1

1 1

1 2 1

1 3 3 1

1 4 6 4 1

1 5 10 10 5 1

Se $m = 0$ ou $m = n$, então $\binom{n}{m} = 1$

Coeficiente Binomial

- Essas relações apresentadas caracterizam a recorrência e os casos base que permitem uma implementação que usa programação dinâmica para os coeficientes binomiais

$$T(n,m) = \begin{cases} 0, & \text{se } m > n \\ 1, & \text{se } m = 0 \text{ ou } m = n \\ T(n-1,m) + T(n-1,m-1), & \text{caso contrário} \end{cases}$$

Coeficiente Binomial

- Para criar uma solução de programação dinâmica deste código, a abordagem top-down é bem próxima da relação de recorrência escrita
  - Assim, pode-se começar escrevendo uma versão recursiva
  - Após a versão recursiva, aplica-se o passo de memorização (que é o que caracteriza uma solução de programação dinâmica)

Coeficiente Binomial

```
long long binom(long long n, long long m)
{
    if (m > n)  {
        return 0;
    }
    if (m == 0 || m == n) {
        return 1;
    }

    return binom(n - 1, m) + binom(n - 1, m - 1);
}
```

Versão recursiva (busca completa)

Coeficiente Binomial

```
map<pair<long long, long long>, long long> pd;

long long binom(long long n, long long m)
{
    if (m > n)  return 0;

    if (m == 0 || m == n) return 1;

    if (pd.count({n, m}) > 0) return pd[{n, m}];

    pd[{n, m}] = binom(n - 1, m) + binom(n - 1, m - 1);

    return pd[{n, m}];
}
```

Versão top-down

Coeficiente Binomial

- É possível otimizar mais a abordagem top-down, uma vez que binômios possuem uma propriedade de simetria $\binom{n}{m} = \binom{n}{n-m}$
- Deve-se tomar cuidado pois os limites (long long) são atingidos bem rápido
  - É comum problemas trabalharem com o resto do cálculo desses coeficientes
    - Deve calcular o resto em todas as operações da função

```
#define MOD 1000000007

map<pair<long long, long long>, long long> pd;

long long binom(long long n, long long m)
{
    if (m > n) return 0;

    if (m == 0 || m == n) return 1;

    // Calcular somente o menor entre m e n - m,
    // pois os coeficientes são iguais
    if (m > m - n) {
        m = n - m;
    }

    if (pd.count({n, m}) > 0) return pd[{n, m}];

    pd[{n, m}] = (binom(n - 1, m) + binom(n - 1, m - 1)) % MOD;

    return pd[{n, m}];
}
```

Versão top-down otimizada

Coeficiente Binomial

- Para a abordagem bottom-up, deve-se:
    - Criar a tabela de memória para o cálculo dos valores
    - Preencher os casos bases
    - Calcular os casos de maneira iterativa, baseado naqueles que já foram calculados

```
#define MOD 1000000007

int pd[100][100];

long long binom(long long n, long long m)
{
    for (long long i = 0; i <= n; i++) {
        for (long long j = 0; j <= m; j++) {
            if (j > i) {
                pd[i][j] = 0;
            } else if (j == 0 || j == i) {
                pd[i][j] = 1;
            } else {
                pd[i][j] = (pd[i - 1][j] + pd[i - 1][j - 1]) % MOD;
            }
        }
    }

    return pd[n][m];
}
```

Versão bottom-up

Problema da Mochila Binária

Problema da Mochila Binária

- É um problema clássico em computação
- Pode ser motivado, de maneira informal pelo seguinte:
  - Digamos que você é um ladrão que invadiu um museu
  - Nesse museu, cada peça possui um **valor** e um **peso**
  - Você tem uma mochila que contém uma **capacidade máxima**
  - Seu objetivo é escolher dentro os objetos, um conjunto que caiba na mochila cujo o valor total **seja o máximo possível**

Problema da Mochila Binária

- De uma maneira formal, ele pode ser escrito pelo seguinte:
  - Considere uma conjunto C = {c1 , c2 , . . . , cN }, onde ci = (wi , vi), e seja M um inteiro positivo.
  - O problema da mochila binária consiste em determinar um subjconjunto S ⊂ {1, 2, . . . , N } de índices de C tal que

$$W = \sum_{j \in S} w_j \leq M$$ e que

$$V = \sum_{j \in S} v_j$$ seja máxima

O termo “mochila binária” remete ao fato de que, para cada um dos objetos, há duas opções: escolhê-lo ou não

Problema da Mochila Binária

- Um primeiro pensamento de solução para esse problema, seria o de ordenar os objetos pelo valor e ir pegando aqueles de maior valor possível, porém imagine o seguinte caso de teste

Capacidade: 20	Item 1	Item 2	Item 3
Peso	10	10	20
Valor	15	20	30

Problema da Mochila Binária

- Um primeiro pensamento de solução para esse problema, seria o de ordenar os objetos pelo valor e ir pegando aqueles de maior valor possível, porém imagine o seguinte caso de teste

Capacidade: 20	Item 1	Item 2	Item 3
Peso	10	10	20
Valor	15	20	30

Nesse caso, a abordagem mandaria escolher o item 3, porém, pegar os itens 1 e 2 produzem uma resposta melhor

Problema da Mochila Binária - Solução

- Como $|C| = N$, há 2^N subconjuntos de índices a serem avaliados
- Como cada subconjunto pode ser avaliado em $O(N)$, uma solução de busca completa tem complexidade $O(N\,2^N)$
- É possível, entretanto, reduzir esta complexidade por meio de um algoritmo de programação dinâmica

Problema da Mochila Binária - Solução

- Seja v(i, m) a soma máxima dos valores que pode ser obtida a partir dos primeiros i elementos de C e uma mochila com capacidade m
- São dois casos-base: o primeiro deles acontece quando não há nenhum elemento a ser considerado
- Neste casos, temos v(0, m) = 0

Problema da Mochila Binária - Solução

- O segundo caso-base acontece quando não há espaço disponível na mochila: $v(i, 0) = 0$
- São duas as transições possíveis:
  a. Ignorar o i-ésimo elemento e considerar apenas os $i - 1$ primeiros; ou
  b. Caso possa ser transportado, pegar o i-ésimo elemento e colocá-lo na mochila

Problema da Mochila Binária - Solução

- A primeira transição não modifica o estado da mochila e nem o total dos valores transportados
- Caso a mochila não consiga transportar o i-ésimo elemento, esta será a única transição possível
- Assim,

$$v(i, m) = v(i - 1, m), \text{ se } w_i > m$$

Problema da Mochila Binária - Solução

- A segunda transição só é possível se $w_i \leq m$
- Caso esta condição seja atendida, a capacidade da mochila é reduzida em w_i unidades e o total dos valores transportados é acrescido em v_i
- Assim, deve-se optar pela transição que produz o maior valor:

$$v(i, m) = \max\{ v(i-1, m), v(i-1, m-w_i) + v_i \}, \text{ se } w_i \leq m$$

Problema da Mochila Binária - Solução

- A solução do problema será dada por v(N, M)
- O número de estados distintos é O(NM) e cada transição é feita em O(1)
- Portanto a solução de programação dinâmica para o problema do troco tem complexidade O(NM)

Problema da Mochila Binária - Busca Completa

```
int solver(int indice, int capacidade)
{
    if (indice < 0) {
        return 0;
    }

    if (capacidade < pesos[indice]) {
        return solver(indice - 1, capacidade);
    }

    int caso1 = solver(indice - 1, capacidade);
    int caso2 = solver(indice - 1, capacidade - pesos[indice]) + valores[indice];

    return max(caso1, caso2);
}
```

Problema da Mochila Binária - Top-down

```
int pd[100][100];
int solver(int indice, int capacidade)
{
    if (indice < 0) { return 0; }
    if (pd[indice][capacidade] != -1) {
        return pd[indice][capacidade];
    }
    if (capacidade < pesos[indice]) {
        return solver(indice - 1, capacidade);
    }
    int caso1 = solver(indice - 1, capacidade);
    int caso2 = solver(indice - 1, capacidade - pesos[indice]) + valores[indice];
    pd[indice][capacidade] = max(caso1, caso2);
    return pd[indice][capacidade];
}
```

Problema da Mochila Binária - Bottom-up

```
int solver(int indice, int capacidade) {
    for (int i = 0; i <= indice; i++) {
        for (int j = 0; j <= capacidade; j++) {
            if (i == 0 || j == 0) {
                pd[i][j] = 0;
            } else if (pesos[i - 1] <= j) {
                pd[i][j] = max(pd[i - 1][j], pd[i - 1][j - pesos[i - 1]] + valores[i - 1]);
            } else {
                pd[i][j] = pd[i - 1][j];
            }
        }
    }
    return pd[indice][capacidade];
}
```

Conclusão