Paradigmas de Resolução de Problemas

Programação Dinâmica: Definição

Prof. Edson Alves - UnB/FGA

2020

Sumário

Definição

Programação Dinâmica

- A programação dinâmica é um paradigma de solução de problemas que combina características dos outros paradigmas
- Assim como o paradigma guloso, ela é aplicável em problemas que possuem subestrutura ótima
- Ela também resolve o problema através da combinação das soluções dos subproblemas, o que se assemelha à etapa de fusão da divisão e conquista
- De forma semelhante a busca completa, ela avalia todas as alternativas disponíveis igualmente
- Ela, porém, difere dos demais paradigmas porque evita recalcular um subproblema múltiplas vezes por meio da técnica da memorização, e por optar por uma ou mais alternativas apenas após avaliar todas elas

Características da programação dinâmica

- A programação dinâmica é aplicável em problemas que possuem duas características:
    1. subestrutura ótima (a solução do problema pode ser formada a partir das soluções ótimas dos subproblemas); e
    2. subproblemas repetidos (subproblemas compartilham subproblemas em comum).
- Caso a segunda característica não esteja presente, não há necessidade de memorização e o algoritmo será equivalente a uma busca completa
- Como a solução do problema será formada a partir da solução dos subproblemas, esta solução pode ser descrita por meio de uma relação de recorrência
- Os subproblemas que podem não podem mais serem subdivididos e que são necessários para a solução dos demais constituem os casos-base do problema

Características da programação dinâmica

- Tanto o problema quanto os subproblemas são caracterizados por estados, os quais correspondem ao conjunto de variáveis (e seus respectivos valores) que identificam unicamente uma instância do problema
- Uma transição corresponde a relação entre uma instância do problema e os subproblemas necessários à sua solução
- A complexidade dos algoritmos de programação dinâmica, em geral, é dada pelo produto do número total de estados pelo custo das transições de cada estado
- A dificuldade de aplicação da programação dinâmica, em geral, reside em se determinar a relação de recorrência que caracteriza a solução

Exemplo de aplicação de programação dinâmica: Números de Fibonacci

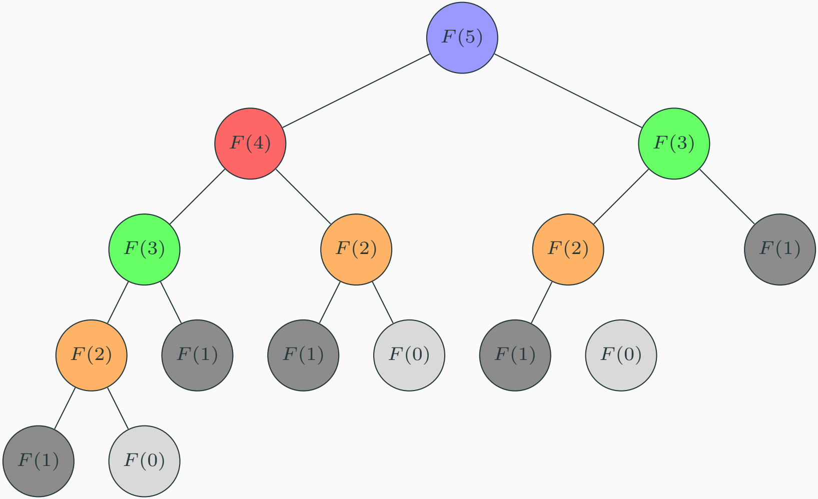
- Considere o problema de se determinar o n-ésimo número de Fibonacci
- Os números de Fibonacci são definido como

$$F(n) = \begin{cases} 0, & \text{se } n = 0 \\ 1, & \text{se } n = 1 \\ F(n-1) + F(n-2), & \text{caso contrário} \end{cases}$$

- Esta definição permite uma implementação direta de busca completa, com complexidade $O(2^n)$:

```
int F(int n)
{
    return n <= 1 ? n : F(n - 1) + F(n - 2);
}
```

Visualização da chamada $F(5)$



Números de Fibonacci e Programação Dinâmica

- A figura anterior ilustra a complexidade exponencial da implementação apresentada
- A medida que a recursão avança, alguns estados são computados repetidas vezes
- Observe que o problema tem subestrutura ótima: o n-ésimo número de Fibonacci pode ser computado a partir de números de Fibonacci anteriores
- A repetição de estados com subestrutura ótima o torna um candidato natural para um algoritmo de programação dinâmica
- Com a adição da memorização, a complexidade muda para $O(N)$, um ganho significativo de performance

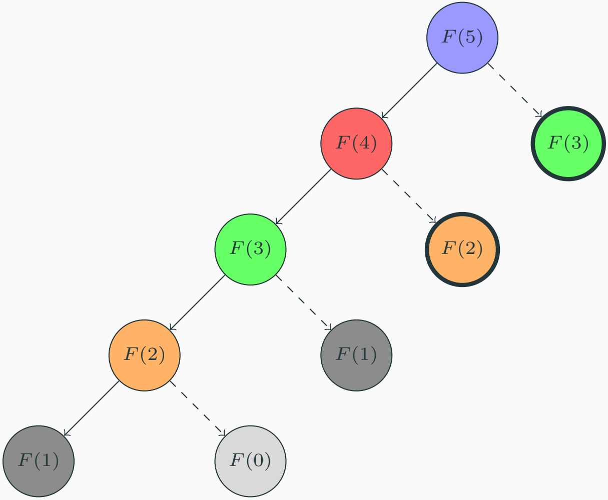
Implementação dos números de Fibonacci com PD

```cpp
#include <bits/stdc++.h>

using namespace std;

const int MAX { 200010 };

int st[MAX];

int F(int N)
{
    if (N == 0 or N == 1)
        return N;

    if (st[N] != -1)
        return st[N];

    auto res = F(N - 1) + F(N - 2);

    st[N] = res;
    return res;
}
```

Implementação dos números de Fibonacci com PD

```cpp

int main()
{
    ios::sync_with_stdio(false);
    memset(st, -1, sizeof st);

    int N;
    cin >> N;

    cout << F(N) << endl;

    return 0;
}
```

Visualização da chamada $F(5)$ com PD



Números de Fibonacci e Programação Dinâmica

- A implementação utilizando programação dinâmica visita cada estado, no máximo, duas vezes
- Na primeira visita (setas contínuas) o estado é computado recursivamente, utilizando a mesma recorrência da solução de busca completa
- Na segunda visita (setas pontilhadas) o estado já foi computado, e o valor armazenado na tabela é retornado imediatamente
- Assim, a complexidade é $O(N)$
- Observe que, exceto pela memorização e inicialização da tabela, o código é idêntico à implementação de busca completa
- Este tipo de implementação é denominada *top-down*
- Há uma segunda forma de implementação de algoritmos de programação dinâmica, denominada *bottom-up*

Implementação *top-down*

Implementação *top-down*

- Uma implementação *top-down* de um algoritmo de programação dinâmica parte da relação de recorrência para produzir uma função recursiva
- Após a verificação dos casos-base, a tabela é consultada para se determinar se o estado já foi computado ou não
- Em caso afirmativo, o valor armazenado na tabela é retornado
- Caso contrário, o subproblema associado ao estado é solucionado por meio de recursão
- Em seguida, a solução obtida é armazenada na tabela e então retornado

Tabela de memorização

- A tabela de memorização deve ter tamanho suficiente para armazenar todos os estados possíveis (dentro dos limites das variáveis que compõem o estado)
- Além disso, esta tabela deve ser iniciada com um valor que sinalize que o estado associado não foi computado ainda
- Este valor sentinela deve corresponder a um valor que não pode ser uma solução de nenhum estado (-1, ∞, etc)
- Assim, a complexidade em memória do algoritmo será, no mínimo, $O(S)$, onde S é o total de estados possíveis

Características da implementação *top-down*

- A principal vantagem da implementação *top-down* é a simplicidade: em geral, é uma tradução literal da relação de recorrência que representa a solução do problema
- Outra vantagem é que apenas os estados necessários à solução são computados
- Por outro lado, a tabela deve ter dimensões que comportem todos os estados possíveis
- Também há um custo de execução associado às chamadas recursivas
- A ordem em que os subproblemas associados aos estados são resolvidos corresponde a uma DFS no grafo cujos vértices são os estados e as arestas as transições

Exemplo: Coeficientes Binomiais

- O coeficiente binomial $\binom{n}{m}$ é dado
$$\binom{n}{m} = \frac{n!}{(n-m)!m!},$$
se $n \geq m$, e $\binom{n}{m} = 0$, caso contrário
- Os coeficientes $\binom{i}{j}$, com $0 \leq j \leq i$, formam a i-ésima linha do Triângulo de Pascal:

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

Exemplo: Coeficientes Binomiais

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

$$\binom{n}{m} = \binom{n-1}{m} + \binom{n-1}{m-1}$$

  , se $0 < m < n$
- Se $m = 0$ ou $m = n$, então $\binom{n}{m} = 1$
- Estas relação caracterizam a recorrência e os casos bases que permitem uma implementação de programação dinâmica para os coeficientes binomiais

Implementação dos coeficientes binomiais com DP

```cpp
#include <bits/stdc++.h>

using namespace std;
const int MAX { 1010 };

long long st[MAX][MAX];

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

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

    if (st[n][m] != -1)
        return st[n][m];

    auto res = binom(n - 1, m) + binom(n - 1, m - 1);

    st[n][m] = res;
    return res;
}
```

Melhoria e correção da implementação

- A implementação anterior pode ser melhorada em um ponto, e também corrigida, uma vez que não produz a saída correta para todas as entradas
- Cada linha do Triângulo de Pascal é simétrica: assim

$$\binom{n}{m} = \binom{n}{n-m}$$

- Este fato permite computa apenas a metade das colunas, e este espaço extra pode ser repassado para as linhas
- Os valores dos coeficientes crescem muito rapidamente, de modo que estouram a capacidade de armazenamento de um `long long`
- Uma forma de tratar isso é computar os restos destes coeficientes para um módulo M dado (em competições, é comum o valor $M = 10^9 + 7$)

Implementação dos coeficientes binomiais melhorada

```
#include <bits/stdc++.h>

using namespace std;

const int MAX { 1010 };
const long long MOD { 1000000007 };

long long st[2*MAX][MAX];

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

    if (m > n - m)
        m = n - m;

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

```

Implementação dos coeficientes binomiais melhorada

```
    if (st[n][m] != -1)
        return st[n][m];

    auto res = (binom(n - 1, m) + binom(n - 1, m - 1)) % MOD;

    st[n][m] = res;
    return res;
}

int main()
{
    int n, m;
    cin >> n >> m;

    memset(st, -1, sizeof st);

    cout << "binom(" << n << "," << m << ") = " << binom(n, m) << endl;

    return 0;
}
```

Implementação *bottom-up*

Implementação *bottom-up*

- Assim como nas implementações *top-down*, uma implementação *bottom-up* também se baseia na relação de recorrência e nos casos-base do problema
- A primeira diferença é que na implementação *bottom-up* todos os estados intermediários, necessários ou não, são computados
- Inicialmente os casos-base são preenchidos
- Em seguida, todos os estados que dependem apenas dos casos-base
- Após eles, os estados que podem ser computados a partir dos estados já computados
- A ordem de preenchimento dos estados correspondem à uma ordenação topológica do grafo cujos vértices são os estados e as arestas são as transições

Características da implementação *bottom-up*

- Em geral, esta ordem corresponde ao preenchimento das linhas, uma por vez
- Em outros problemas, porém, a ordem pode não ser óbvia à primeira vista
- Esta forma de preenchimento da tabela de memorização dispensa uma inicialização prévia
- Nos casos onde os elementos de uma linha dependem apenas da linha anterior, a complexidade de memória pode ser reduzida, armazenando-se apenas duas linhas por vezes ($O(N)$ ao invés de $O(NM)$ da tabela completa da implementação *top-down*)
- Por fim, implementações *bottom-up* não utilizam de recursão, em geral sendo baseadas em laços aninhados

Implementação *bottom-up* dos números de Fibonacci

```cpp
#include <bits/stdc++.h>

using namespace std;

const int MAX { 200010 };

long long fib[MAX];

void precomp()
{
    fib[0] = 0;
    fib[1] = 1;

    for (int i = 2; i < MAX; ++i)
        fib[i] = fib[i - 1] + fib[i - 2];
}

long long F(int N)
{
    return fib[N];
}
```

Redução de memória

- Conforme comentado anteriormente, quando a relação de recorrência para valores da linha seguinte dependem apenas da linha anterior, e não é preciso manter o registro do caminho ótimo, é possível reduzir a complexidade em memória de implementações *bottom-up*
- Basta manter duas referências para dois vetores, uma para a linha atual (`next`) e outra para a linha anterior (`prev`)
- Os valores de `prev` são usados para computar os valores de `next`
- Em seguida, as referências são trocadas, e o processamento continua
- Também é comum usar uma matriz bidimensional, e trocar entre as linhas por meio do valor associado à segunda dimensão

Implementação *bottom-up* dos coeficientes binomiais

```cpp
#include <bits/stdc++.h>

using namespace std;

const int MAX { 2010 };
const long long MOD { 1000000007 };

long long a[MAX], b[MAX];

long long binom(int n, int m)
{
    long long *prev = a, *next = b;

    prev[0] = 1;

    for (int i = 1; i <= n; ++i)
    {
        next[0] = next[i] = 1;

```

Implementação *bottom-up* dos coeficientes binomiais

```
        for (int j = 1; j < n; ++j)
            next[j] = (prev[j] + prev[j - 1]) % MOD;

        swap(prev, next);
    }

    return prev[m];
}

int main()
{
    int n, m;
    cin >> n >> m;

    cout << "binom(" << n << "," << m << ") = " << binom(n, m) << endl;

    return 0;
}
```

Referências

1. **CORMEN**, Thomas H.; **LEISERSON**, Charles E.; **RIVEST**, Ronald; **STEIN**, Clifford. *Introduction to Algorithms*, 3rd Edition, MIT Press, 2009.
2. **LAARKSONEN**, Antti. *Competitive Programmer's Handbook*, 2017.
3. **HALIM**, Steve; **HALIM**, Felix. *Competitive Programming 3*, Lulu, 2013.