PARADIGMAS DE PROJETO DE ALGORITMOS

DCE529 - Algoritmos e Estruturas de Dados III

Atualizado em: 18 de abril de 2023

Iago Carvalho

Departamento de Ciência da Computação

PROJETO DE ALGORITMOS

Projeto de algoritmos é um método específico para criar um processo matemático na resolução de problemas.

- Quando aplicado, da-se origem a engenharia de algoritmos

Existem diversas formas de se realizar o projeto de algoritmos

- Cada *forma de pensamento* constitui um diferente paradigma
    - Recursividade
    - Força bruta
    - Guloso
    - **Divisão e conquista**
    - **Programação dinâmica**

DIVISÃO E CONQUISTA

Um algoritmo de divisão e conquista é baseado em três passos básicos

1. Dividir o problema em subproblemas menores (**Divisão**)
2. Resolver cada subproblema separadamente (**Conquista**)
3. Recombinar as soluções dos subproblemas em uma solução global

Na grande maioria das vezes, algoritmos de divisão e conquista são baseados em recursão

- Se a entrada é muito grande, divide-se em partes menores e as resolve de forma recursiva
- Se a entrada já é pequena o suficiente, então ela é resolvida

DIVISÃO E CONQUISTA - PROBLEMAS

Existem alguns problemas que podem ser resolvidos utilizando este paradigma

Ordenação de números

- Mergesort
- Quicksort

Computação da sequência de Fibonacci

- Implementação recursiva

Busca binária

Encontrar a maior subsequência de elementos em um vetor

QUANDO UTILIZAR DIVISÃO E CONQUISTA

Existem quatro condições para a eficiente utilização deste paradigma

1. Deve-se ser possível dividir o problema em subproblemas menores
2. A combinação dos resultados deve ser eficiente
3. Subproblemas devem ter tamanhos parecidos
    - Sempre que estiverem no mesmo nível de recursão
4. A solução dos subproblemas deve ser simples
    - Operações repetidas ou correlacionadas

PSEUDO-CÓDIGO

```
D&CGenerico(x)
Entrada: (sub)problema x
se x é o caso base então
    retorna resolve(x);
senão
    Divida x em n subproblemas x_0, x_1, ..., x_{n-1};
    para i ← 0 até n − 1 faça
        y_i ← D&CGenerico(x_i);
    fim
    Combine y_0, y_1, ..., y_{n-1} em y;
    retorna y;
fim
```

COMPLEXIDADE

A complexidade de algoritmos construídos utilizando este paradigma deve ser analisada como equações de recorrência

Mergesort e Quicksort:

$$T(1) = 1$$
$$T(n) = 2T\left(\frac{n}{2}\right) + n, \forall n > 1;$$

Busca em árvore binária:

$$T(n) = T\left(\frac{n}{2}\right) + 1$$

DIVISÃO E CONQUISTA - VANTAGENS

Problemas muito grandes, quando divididos, tornam-se exponencialmente mais simples

Tendência a complexidade logaritmica

Algoritmos altamente paralelizáveis na *conquista*

Maior precisão numérica

- Caso seja necessário realizar calculos matemáticos extremamente precisos (muitas casas decimais) é mais interessante realizar diversos calculos em separado e depois agrupa-los

DIVISÃO E CONQUISTA - DESVANTAGENS

Número de chamadas recursivas pode ser incoveniente

- Pilha de execução muito grande
- Grande uso de memória

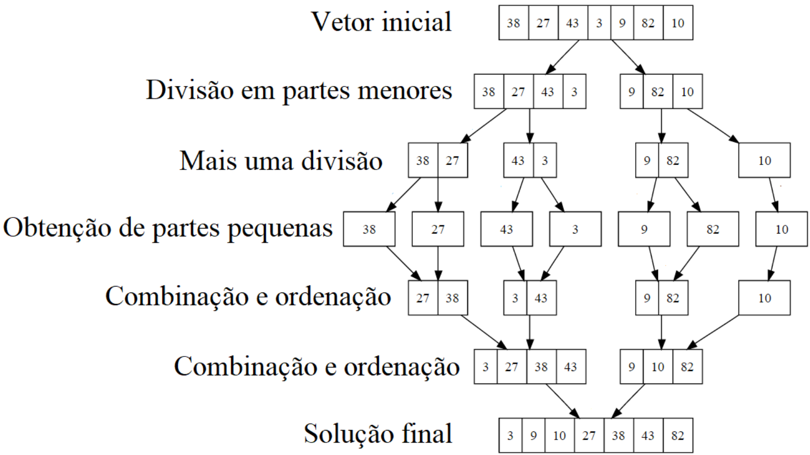
Eventual dificuldade para definir o caso base

- Quando eu paro de dividir e começo a conquistar?

Rendundância na resolução de subproblemas repetidos

- Semelhante ao que acontece ao resolvermos a sequência de Fibonacci

MERGESORT


Vetor inicial
38 27 43 3 9 82 10
Divisão em partes menores
38 27 43 3
9 82 10
Mais uma divisão
38 27
43 3
9 82
10
Obtenção de partes pequenas
38
27
43
3
9
82
10
Combinação e ordenação
27 38
3 43
9 82
10
Combinação e ordenação
3 27 38 43
9 10 82
Solução final
3 9 10 27 38 43 82

PROGRAMAÇÃO DINÂMICA

PROGRAMAÇÃO DINÂMICA

É um nome *chique* para algoritmos recursivos utilizando uma tabelinha

- Cada subproblema recursivo é resolvido e guardado em uma tabela
- Subproblemas são resolvidos sequencialmente
  - Em ordem crescente de tamanho
- Subproblemas salvos são utilizados na resolução de problemas maiores

Quando um algoritmo recursivo tem complexidade exponencial, a programação dinâmica pode levar a um algoritmo mais eficiente

Caso a soma dos tamanhos dos subproblemas seja igual a $O(n)$, é possível obter algoritmos eficientes

Caso sejam obtidos n subproblemas de tamanho $n-1$, a tendência é gerar um algoritmo exponencial

RECURSIVIDADE PARA COMPUTO DE TABELA VERDADE

A	B	C	D	F
0	0	0	0	1
0	0	0	1	1
0	0	1	0	0
0	0	1	1	0
0	1	0	0	1
0	1	0	1	1
0	1	1	0	0
0	1	1	1	0
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	0
1	1	0	0	1
1	1	0	1	0
1	1	1	0	1
1	1	1	1	0

A = 0 ⇨

A	B	C	D	F
0	0	0	0	1
0	0	0	1	1
0	0	1	0	0
0	0	1	1	0
0	1	0	0	1
0	1	0	1	1
0	1	1	0	0
0	1	1	1	0

A = 1 ⇨

A	B	C	D	F
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	0
1	1	0	0	1
1	1	0	1	0
1	1	1	0	1
1	1	1	1	0

PROGRAMAÇÃO DINÂMICA

Algoritmos por Programação Dinâmica são quase sempre recursivos

- De certa forma, lembram algoritmos de divisão e conquista

Um possível defeito na busca recursiva é a computação redundante de subproblemas, ou a exploração redundante do espaço de busca

- Para contornar esta situação, podemos armazenar os resultados dos subproblemas já resolvidos
  - Para cada subproblema inédito, o resolvemos e armazenamos o resultado
  - Para os subproblemas repetidos, apenas consultamos o resultado

CARACTERIZAÇÃO DOS PROBLEMAS

Programação dinâmica é aplicável a problemas que possuem as seguintes propriedades

Formulação recursiva bem definida: Caso a formulação seja recursiva, esta não deve possuir ciclos

Subestrutura ótima: A combinação de soluções ótimas de subproblemas também é ótima

Superposição de subproblemas: O espaço de subproblemas é pequeno e eles se repetem com frequência durante a solução do problema original.

TABELA DA PROGRAMAÇÃO DINÂMICA

A definição da estrutura desta tabela é de extrema importância para a eficiência do algoritmo

- Quais serão as dimensões da tabela?
- O que significa cada dimensão da tabela
- Como determinar os índices de entrada da tabela?
- O que será salvo em cada entrada da tabela?
    - Valor da solução?
    - Componentes da solução?

PROGRAMAÇÃO DINÂMICA *TOP-DOWN*

Nesta abordagem *top-down* é utilizada recursão

Problema original é decomposto em subproblemas menores que são resolvidos recursivamente

- Logo após, são combinados em uma solução global

As entradas da tabela são preenchidas conforme necessário

- Pode ser que a tabela não seja completamente preenchida

PROGRAMAÇÃO DINÂMICA *BOTTOM-UP*

Algoritmos que não utilizam recursão

- Subproblemas são resolvidos e combinados sucessivamente de maneira a construir a solução do problema original

A tabela é completamente preenchida

- Preenchida em ordem, dos subproblemas menores para os maiores

VANTAGENS E DESVANTAGENS

A programação dinâmica realiza um *trade-off* entre tempo de computação e espaço de memória

Vantagens

- Economiza a computação de soluções de subproblemas repetidos
  - Superposição de problemas
  - Subestrutura ótima

Perigos

- Tabela pode crescer exponencialmente
  - Grande gasto de memória

EXEMPLO

Qual é a melhor maneira de se fazer a multiplicação de n matrizes?

- O produto de uma matriz $p \times q$ por outra $q \times r$ requer $\mathcal{O}(pqr)$ operações

Considere o produto:
$M = M_1[10, 20] \times M_2[20, 50] \times M_3[50, 1] \times M_4[1, 100]$

- O produto $M = M_1 \times (M_2 \times (M_3 \times M_4))$ tem 125 mil operações
- O produto $M = (M_1 \times (M_2 \times M_3)) \times M_4$ tem somente 2200 operações

EXEMPLO

Seja m_{ij} o menor custo para computar o produto $M_i \times M_{i+1} \times \ldots M_j$, para $1 \leq i \leq j \leq n$

Podemos definir um algoritmo de programação dinâmica como

$$m_{ij} = \begin{cases} 0, & \text{se } i = j \\ \min_{i \leq k \leq j}(m_{ik} + m_{k+1,j} + b_{i-1} b_k b_j), & \text{se } j > i \end{cases}$$

- m_{ik} é o custo mínimo para computar $M' = M_i \times M_{i+1}, \times \ldots \times M_k$
- $m_{k+i,j}$ é o custo mínimo para computar $M'' = M_{k+1} \times M_{k+2}, \times \ldots \times M_j$
- $b_{i-1} b_k b_j$ representa o custo para multiplicar as matrizes $M'[b_{i-1}, b_k]$ e $M''[b_k, b_j]$

EXEMPLO

Os valores m_{ij} são calculados em ordem crescente

- Algoritmo *bottom-up*
- Não utiliza recursão

O algoritmo se inicia computando m_{ii} para todo i

- Depois computa $m_{i,i+1}$ para todo i
- Depois computa $m_{i,i+2}$ para todo i
- $\ldots$

EXEMPLO

$m_{11} = 0$	$m_{12} = 10.000$	$m_{13} = 1.200$	$m_{14} = 2.200$
	$m_{22} = 0$	$m_{23} = 1.000$	$m_{24} = 3.000$
		$m_{33} = 0$	$m_{34} = 5.000$
			$m_{44} = 0$

EXEMPLO

```c
#define Maxn 10
int main(int argc, char *argv[]) {
    int i, j, k, h, n, temp;
    int b[Maxn+1];
    int m[Maxn][Maxn];
    printf("Numero de matrizes n: ");
    scanf("%d", &n);
    getchar();
    printf("Dimensoes das matrizes: ");
    for (i = 0; i <= n; i++) {
        scanf("%d", &b[i]);
    }
    for (i = 0; i < n; i++) {
        m[i][i] = 0;
    }
    for (h = 1; h <= n-1; h++) {
        for (i = 1; i <= n-h; i++) {
            j = i+h;
            m[i-1][j-1] = INT_MAX;
            for (k = i; k <= j-1; k++) {
                temp = m[i-1][k-1] + m[k][j-1] + b[i-1] * b[k] * b[j];
                if (temp < m[i-1][j-1]) {
                    m[i-1][j-1] = temp;
                }
            }
            printf("m[%d][%d] = %d\n", i-1, j-1, m[i-1][j-1]);
        }
        putchar("\n");
    }
    return 0;
}
```