

Notas de Aula - AED2 – Técnicas e Análise de Algoritmos: divisão e conquista  
Jefferson T. Oliva

## Divisão e Conquista

Ideia básica:

- divisão: dividir o problema em subproblemas menores recursivamente até que seja atingido um problema de tamanho mínimo (caso base)
- conquista: solucionar os problemas recursivamente (do menor para o maior)
- combinação: combinar os resultados recursivamente até que o problema geral seja resolvido

No slide 4 é apresentado um pseudo-código para a solução de problemas de divisão e conquista.

Entre as linhas 1 e 3 é o caso base, onde o problema (sub-problema) possui um tamanho mínimo. Na linha 4, o problema é dividido em  $n$  subproblemas (geralmente,  $n = 2$ ). Entre as linhas 5 e 8, são realizadas chamadas recursivas para os subproblemas (onde poderão haver mais decomposições). Na linha 9, os resultados das chamadas recursivas na linha 6 são combinadas

*Divisao\_e\_Conquista(x)*

```
1:  if  $x$  é pequeno ou simples then
2:      return resolver( $x$ )
3:  else
4:      decompor  $x$  em conjuntos menores  $x_0, x_1, \dots, x_n$ 
5:      for ( $i = 0$  até  $n$ ) do
6:           $y_i = \text{Divisao\_e\_Conquista}(x_i)$ 
7:           $i = i + 1$ 
8:      end for
9:      combinar  $y_i$ 's
10:  end if
11:  return  $y$ 
```

Podemos aplicar divisão e conquista em diversos problemas, dos quais, iremos ver nessa aula

- encontrar o maior valor
- potenciação
- mochila

### Exemplo 1: encontrar o maior valor

O problema consiste em encontrar o maior elemento de um array  $A[1..n]$ .

Na solução ingênua (slide 5), o item procurado é comparado, sequencialmente, com cada elemento do vetor. Nessa busca, mesmo que o maior elemento esteja na primeira posição, ainda são realizadas mais  $n - 1$  comparações. Por essa razão, a complexidade do algoritmo é  $O(n)$ .

#### *Solução Ingênua*

```
1:  max = A[1]
2:  for i = 2 até n do
3:      if A[i] > max then
4:          max = A[i]
5:      end if
6:  end for
7:  return max
```

Esse problema pode ser resolvido por divisão e conquista, conforme o algoritmo apresentado no slide 6: onde  $A$  é o arranjo (vetor),  $x$  e  $y$  são as posições dos vetores.

O caso base (linha 2) ocorre quando comparamos elementos do vetor que são adjacentes, ou seja, a diferença das posições  $x$  e  $y$  será 1 ou 0 (ocorre quando o vetor ou sub-vetor (contém elementos nas posições entre  $x$  e  $y$ ) possui tamanho ímpar). Nesse caso, basta obter o valor máximo entre  $v[x]$  e  $v[y]$ . Nesse caso, ocorre a “**conquista**”.

No caso iterativo (linhas 4-6), quando  $x$  está “mais longe” de  $y$ , é necessário dividir o espaço de busca do arranjo (particionar o vetor ou sub-vetor em dois). Para isso, primeiro é calculada a posição intermediária entre  $x$  e  $y$  (basta somar ambos e dividir por 2 na linha 4). Em seguida, na primeira chamada recursiva, é procurado o maior elemento para as posições entre  $x$  e  $m$ . Na segunda, chamada, a busca é entre as posições  $m + 1$  e  $y$ . Nesse caso (iterativo), ocorre a “**divisão do problema**”.

Por fim, na linha 8 são **combinadas** os resultados da divisão e conquista. Nessa linha, é obtido o valor máximo entre os resultados das duas chamadas recursivas.

**Quando o algoritmo for chamado, os argumentos  $x$  e  $y$  devem ser a primeira e a última posição do vetor, respectivamente.**

#### *Solução Divisão e Conquista*

```
1:  if y - x ≤ 1 then
2:      return max(A[x], A[y])
3:  else
4:      m = (x + y)/2
5:      v1 = Maxim(A[x..m])
6:      v2 = Maxim(A[m + 1..y])
7:  end if
8:  return max(v1, v2)
```

Afinal, qual é a complexidade desse algoritmo?

Como o algoritmo é recursivo, então devemos analisá-lo por meio de recorrências (slide 7).

No caso base, o valor da função é uma constante  $c$ . Como a análise de algoritmos é termos do tamanho da entrada ( $n$ ), então a condição para que seja caso base é quando  $n$  seja menor ou igual 2, ou seja, o vetor ou sub-vetor possui, no máximo, dois elementos.

No caso iterativo ( $n > 2$ ), fazemos duas chamadas recursivas, onde o tamanho do conjunto é dividido pela metade e também são realizadas outras  $c$  operações. Desse modo, para o caso iterativo,  $T(n) = 2T(n/2) + c$ . Como podemos ver no slide 7, a nossa recorrência “encaixa” no caso 1 do teorema mestre, onde  $a = 2$ ,  $b = 2$ , log de  $b$  na base  $a$  é igual a 1 e  $f(n) = c$ . Para uma constante epsilon maior que zero (essa constante não pode ser  $\leq 0$ ) e menor igual a 1,  $f(n)$  pertence a  $O(1)$ . Logo, a complexidade da recorrência é na ordem de  $\Theta(n)$ . Também, não seria errado afirmar que o algoritmo possui a complexidade de  **$O(n)$** , pois estaríamos nos referindo ao pior caso, que é o que geralmente interessa.

## Exemplo 2: potenciação

Dada duas variáveis de números naturais  $a$  e  $n$ : o objetivo seria calcular  $a$  elevada a  $n$ .

Em uma solução ingênua, para  $n > 2$ , basta multiplicar  $a$  por  $a$  sucessivamente, ou seja,  $n - 1$  vezes. Na solução ingênua, quando  $n$  for menor ou igual a zero é ignorado. A solução abaixo possui complexidade de  $O(n)$ .

```
potencia(a, n)
1:   p = a
2:   for i = 2 até n do
3:       p = p * a
4:   end for
5:   return p
```

## Solução divisão e conquista

Caso base 1: quando  $n$  for igual a zero, basta retornar 1.

Caso base 2: quando  $n$  for igual a um, basta retornar  $a$ .

Caso iterativo 1: quando  $n$  for par, obtenha o valor  $a$  elevada a  $n/2$  (divisão) e armazene o resultado em  $x$ . Em seguida, multiplique  $x$  por  $x$ .

Caso iterativo 2: quando  $n$  for ímpar, obtenha o valor  $a$  elevada a  $(n - 1)/2$  (divisão) e armazene o resultado em  $x$ . Por que  $a$  elevada a  $(n - 1)/2$  nesse caso? Para que uma chamada recursiva não contenha número fracionário como parâmetro. Em seguida, faça a operação  $a * x * x$ .

*potencia(a, n)*

```

1:   if n = 0 then
2:       return 1
3:   else if n = 1 then
4:       return a
5:   else
6:       if n é par then
7:           return potencia(a, n/2) * potencia(a, n/2) * a
8:       else
9:           return potencia(x, (n - 1)=2) × potencia(a, (n - 1)=2) * a
10:      end if
11:  end if

```

Para o cálculo da complexidade desse algoritmo, também é necessária a análise de recorrência. Para isso, o caso base foi definido como  $T(0) = T(1) = c$  e o caso iterativo, como  $T(n) = T(n/2) + c$ . Podemos resolver essa recorrência por qualquer um dos métodos que vimos em sala de aula. Neste caso, vamos utilizar o teorema mestre. Assim,  $a = 1$ ,  $b = 2$ , log de  $a$  na base 2 é igual a zero,  $f(n) = c$ .

Como  $f(n)$  pertence a  $\Theta(1)$  ( $n$  elevado a 0 ou a log de  $a$  na base 2), que é o caso 2 do teorema, então a complexidade do algoritmo é de  $\Theta(\log n)$ . Assim, o pior caso para o nosso algoritmo é de  **$O(\log n)$** .

### Exemplo 3: mochila

A solução do problema da mochila binária por divisão e conquista, divide o conjunto de objetos em dois até que cada um tenha tamanho 1. Em seguida, há tentativas de inclusão de cada conjunto na mochila.

Se o vetor tiver o tamanho igual a 1, a função verifica se o peso do item não irá extrapolar a capacidade da mochila. Se a capacidade não for extrapolada, subtraia-a com o peso do item, já que o mesmo será adicionado na mochila e, em seguida, retorne o custo do item. Caso contrário, apenas retorne 0.

Caso o tamanho do vetor seja maior que 1:

- **Divisão:** divida o vetor ao meio.
- **Conquista 1:** tente incluir, na mochila, um objeto da primeira metade recursivamente.
- **Conquista 2:** tente incluir, na mochila, um objeto da primeira metade recursivamente.
- **Combinação:** some o resultado das duas metades.

Implementação em C:

```
mochilaDQ(P, C, b, i, f)
1:   if i = f and b - P[ini] ≥ 0 then
2:       b ← b - P[i]
3:       return C[i]
4:   else if i = f and b - P[ini] < 0 then
5:       return 0
6:   else
7:       m ← (i + f) / 2
8:       return mochilaDQ(P, C, b, i, m) + mochilaDQ(P, C, b, m + 1, f)
9:   endif
```

A complexidade  $T(n)$  do algoritmo é uma fórmula de recorrência:

- $T(1) = c$
- $T(n) = T(n/2) + T(n/2) + cn$ , para  $n > 1$

Ao resolver a recorrência acima pela árvore de recursão (slides 15 e 16), o algoritmo, tem o custo, no pior caso, na ordem de  $O(n \log n)$ .

É importante ressaltar que a solução do problema da mochila utilizando divisão e conquista pode não gerar solução ótima!

### Outros exemplos de aplicação

O método da divisão e conquista também pode ser aplicado em outros métodos:

- Distância Euclidiana
- Busca Binária
- Mediana
- Quicksort
- etc

Vantagens: altamente paralelizáveis; fácil implementação; simplificação de problemas complexos

Desvantagens: necessidade de memória auxiliar e pode haver repetição de subproblemas

Quando utilizar?

- Deve ser possível decompor um problema em sub-problemas
- A combinação dos resultados deve ser eficiente
- Os subproblemas devem ter, mais ou menos, o mesmo tamanho

## Referências

Cormen, T. H.; Leiserson, C. E.; Rivest, R. L.; Clifford, S. Algoritmos: teoria e prática. Elsevier, 2012.

Horowitz, E., Sahni, S. Rajasekaran, S. Computer Algorithms. Computer Science Press, 1998.

Szwarcfiter, J.; Markenzon, L. Estruturas de Dados e Seus Algoritmos. LTC, 2010.

Ziviani, M. Projetos de Algoritmos: com implementações em Pascal e C. Thomson, 2004.

## Apêndice: mergesort

Mergesort, ou ordenação por intercalação, divide o arranjo em dois até que cada um tenha tamanho 1. Em seguida, os sub-arranjos são intercalados até a ordenação completa.

Se o vetor tiver o tamanho igual a 1 (caso base), a função apenas retorna o elemento.

A divisão é feita pela função mergesort. A conquista e a combinação é feita função merge.

Caso o tamanho do vetor seja maior que 1

1 – Divisão: divida o vetor ao meio

2 – Conquista 1: ordene a primeira metade recursivamente

3 – Conquista 2: ordene a segunda metade recursivamente

4 – Combinação: intercale as duas metades de forma ordenada

```
mergesort(A, p, r)
1:   if p < r then
2:       q = (p + r)/2
3:       mergesort(A, p, q)
4:       mergesort(A, q + 1, r)
5:       merge(A, p, q, r)
6:   end if
```

Na função merge (slide 13), são utilizados dois arranjos auxiliares B e C, que são utilizados como pilhas estáticas, onde o topo é o primeiro elemento (B[0] ou C[0]) e o fundo é um valor infinito, que tem a finalidade de evitar que a pilha “estoure”, pois nenhum elemento é maior que o infinito. Essas pilhas são intercaladas de forma ordenada entre as linhas 9 e 15.

```
merge(A, p, q, r)
1:   for i = p to q do
2:       B[i] = A[i]
3:   end for
4:   for j = q + 1 to r do
5:       C[j - q] = A[j] /*{Se for uma linguagem em que o primeiro elemento de vetores é acessado na posição
0, então C[j - q - 1] = A[j]}*/
6:   end for
7:   i = p
8:   j = q
9:   for k = p to r do
10:      if B[i] ≤ C[j] then
11:          A[k] = B[i]
12:          i = i + 1
13:      else
14:          A[k] = C[j]
15:          j = j + 1
16:      end if
17:   end for
```

Na análise de recorrência, temos o caso base constante e o iterativo,  $T(n) = \text{arredonda\_para\_cima}(T(n/2)) + \text{arredonda\_para\_baixo}(T(n/2)) + cn$

Em cada nível da chamada recursiva, onde o último nível é o caso base, sempre são manipulados  $n$  números. Por isso que na parte iterativa da recorrência é adicionado **cn**.

Resolvendo a recorrência pela árvore de recorrência, chegamos ao pior caso na ordem de  **$O(n \log n)$**  (ver demonstração nos slide 15 e 16)

## Implementação em C

```
void merge(int v[], int p, int q, int r){
    int i, j, k;
    int n1 = q - p + 1;
    int n2 = r - q;
    int L[n1 + 1];
    int R[n2 + 1];

    for (i = 0; i < n1; i++)
        L[i] = v[p + i];

    for (j = 0; j < n2; j++)
        R[j] = v[q + j + 1];

    L[n1] = inf;
    R[n2] = inf;

    i = 0;
    j = 0;

    for (k = p; k <= r; k++)
        if (L[i] <= R[j]){
            v[k] = L[i];
            i = i + 1;
        }
        else{
            v[k] = R[j];
            j = j + 1;
        }
    }

void mergesort(int v[], int p, int r){
    int q;

    if (p < r){
        q = (p + r) / 2;

        mergesort(v, p, q);
        mergesort(v, q + 1, r);

        merge(v, p, q, r);
    }
}
```



```
int main(){
    int i, v[] = {22, 33, 55, 77, 99, 11, 44, 66, 88};

    printf("Antes da ordenacao\n");

    for (i = 0; i < 9; i++)
        printf("%d\n", v[i]);

    mergesort(v, 0, 8);

    printf("\nApos ordenacao\n");

    for (i = 0; i < 9; i++)
        printf("%d\n", v[i]);

    return 0;
}
```

Como o algoritmo Merge Sort usa a recursividade, há um alto consumo de memória e tempo de execução, tornando esta técnica não muito eficiente em alguns problemas.

Os três passos úteis dos algoritmos de divisão e conquista, ou divide and conquer, que se aplicam ao merge sort são:

- Dividir: Calcula o ponto médio do sub-arranjo
- Conquistar: Recursivamente resolve dois subproblemas, cada um de tamanho  $n/2$
- Combinar: Unir os sub-arranjos em um único conjunto ordenado