

Técnicas de Projeto (Parte 2)

Projeto e Análise de Algoritmo

Pontifícia Universidade Católica de Minas Gerais

Técnicas de Projeto

1) Divisão e Conquista

Divisão e Conquista

- Consiste em dividir o problema em partes menores, encontrar soluções para essas partes (supostamente mais fácil), e combina-las em uma solução global.
 - Geralmente leva a soluções eficientes e elegantes, principalmente se forem recursivas.
- Basicamente essa técnica consiste das seguintes fases (executadas nesta ordem):
 - Divisão (particionamento) do problema original em sub-problemas similares ao original mas que são menores em tamanho;
 - Resolução de cada sub-problema sucessivamente e independentemente (em geral de forma recursiva);
 - Combinação das soluções individuais em uma solução global para todo o problema.

Divisão e Conquista

- Um algoritmo de “divisão e conquista” é normalmente relacionado a uma equação de recorrência que contém termos referentes ao próprio problema.

$$T(n) = aT(\frac{n}{b}) + f(n)$$

onde a indica o número de sub-problemas gerados, b o tamanho de cada um deles e $f(n)$ o custo para fazer a divisão.

Exemplo: Max/Min

- Seja A um vetor de inteiros, $A[1..n]$, $n \geq 1$ que não está ordenado.
- Pede-se:
 - Determine o maior e o menor elementos desse vetor usando divisão e conquista;
 - Determine o custo (número de comparações) para achar esses dois elementos supondo que A possui n elementos.

Exemplo: Max/Min

Cada chamada de `MaxMin4` atribui às variáveis *Max* e *Min* o maior e o menor elementos em $A[Linf]..A[Lsup]$.

`MAXMIN4(Linf, Lsup, Max, Min)`

▷ Variáveis auxiliares: *Max1, Max2, Min1, Min2, Meio*

```

1  if (Lsup - Linf) ≤ 1                                ▷ Condição da parada recursiva
2  then if A[Linf] < A[Lsup]
3      then Max ← A[Lsup]
4          Min ← A[Linf]
5      else Max ← A[Linf]
6          Min ← A[Lsup]
7  else Meio ← ⌊ $\frac{Linf+Lsup}{2}$ ⌋                        ▷ Acha o menor e maior elementos de cada partição
8      MAXMIN4(Linf, Meio, Max1, Min1)
9      MAXMIN4(Meio+1, Lsup, Max2, Min2)
10     if Max1 > Max2
11         then Max ← Max1
12         else Max ← Max2
13     if Min1 < Min2
14         then Min ← Min1
15         else Min ← Min2

```

Exemplo: Max/Min

- Análise:

Seja $f(n)$ o número de comparações entre os elementos de A , que possui n elementos.

$$\begin{aligned} f(n) &= 1, & \text{para } n \leq 2, \\ f(n) &= f(\lfloor n/2 \rfloor) + f(\lceil n/2 \rceil) + 2, & \text{para } n > 2. \end{aligned}$$

Quando $n = 2^i$ para algum inteiro positivo i , temos que:

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

Exemplo: Max/Min

- Análise:

Resolvendo esta equação de recorrência (em função de n e i), temos:

$$\begin{array}{ll}
 f(n) &= 2f(\frac{n}{2}) + 2 \\
 2f(\frac{n}{2}) &= 2^2 f(\frac{n}{2^2}) + 2^2 \\
 2^2 f(\frac{n}{2^2}) &= 2^3 f(\frac{n}{2^3}) + 2^3 \\
 &\vdots \\
 2^{i-3} f(\frac{n}{2^{i-3}}) &= 2^{i-2} f(\frac{n}{2^{i-2}}) + 2^{i-2} \\
 2^{i-2} f(\frac{n}{2^{i-2}}) &= 2^{i-1} f(\frac{n}{2^{i-1}}) + 2^{i-1} \\
 &= 2^{i-1} f(2) + 2^{i-1} \\
 &= 2^{i-1} + 2^{i-1}
 \end{array}
 \qquad
 \begin{array}{ll}
 f(2^i) &= 2f(2^{i-1}) + 2 \\
 2f(2^{i-1}) &= 2^2 f(2^{i-2}) + 2^2 \\
 2^2 f(2^{i-2}) &= 2^3 f(2^{i-3}) + 2^3 \\
 &\vdots \\
 2^{i-3} f(2^3) &= 2^{i-2} f(2^2) + 2^{i-2} \\
 2^{i-2} f(2^2) &= 2^{i-1} f(2^1) + 2^{i-1} \\
 &= 2^{i-1} f(2) + 2^{i-1} \\
 &= 2^{i-1} + 2^{i-1}
 \end{array}$$

Fazendo a expansão desta equação temos:

$$\begin{array}{ll}
 2^{i-2} f(2^2) &= 2^{i-1} + 2^{i-1} \\
 2^{i-3} f(2^3) &= 2^{i-1} + 2^{i-1} + 2^{i-2} \\
 &\vdots \\
 2^2 f(2^{i-2}) + 2^2 &= 2^{i-1} + 2^{i-1} + 2^{i-2} + \dots + 2^3 \\
 2f(2^{i-1}) + 2 &= 2^{i-1} + 2^{i-1} + 2^{i-2} + \dots + 2^3 + 2^2 \\
 f(2^i) &= 2^{i-1} + 2^{i-1} + 2^{i-2} + \dots + 2^3 + 2^2 + 2 \\
 &= 2^{i-1} + \sum_{k=1}^{i-1} 2^k = 2^{i-1} + 2^i - 2 \\
 f(n) &= \frac{n}{2} + n - 2 = \frac{3n}{2} - 2.
 \end{array}$$

Logo, $f(n) = 3n/2 - 2$ para o melhor caso, pior caso e caso médio.

Exemplo: Max/Min

- Conforme mostrado anteriormente, o algoritmo apresentado neste exemplo é **ótimo**.
- Entretanto, ele pode ser pior do que os já apresentados, pois, a cada chamada recursiva, salva *Linf*, *Lsup*, *Max* e *Min*, além do endereço de retorno da chamada para o procedimento.
- Além disso, uma comparação adicional é necessária a cada chamada recursiva para verificar se $Lsup - Linf \leq 1$ (condição de parada).
- O valor de $n + 1$ deve ser menor do que a metade do maior inteiro que pode ser representado pelo compilador, para não provocar *overflow* na operação $Linf + Lsup$.

Exemplo: Exponenciação

Problema: Calcular a^n , para todo real a e inteiro $n \geq 0$.

Primeira solução (incremental):

- Caso base: $n = 0$; $a^0 = 1$.
- Hipótese de indução: Suponha que, para qualquer inteiro $k < n$ e real a , sei calcular a^k .
- Passo da indução: Queremos provar que conseguimos calcular a^k , para $k=n$. Por hipótese de indução, sei calcular a^{n-1} . Então, calculo a^n multiplicando a^{n-1} por a .

Exemplo: Exponenciação

Exponenciação(a , n)

```
se  $n = 0$  então retorne(1)
senão  $an := \text{Exponenciação}(a, n - 1)$ 
 $an := an * a$ 
retorne ( $an$ )
```

Análise:

Vamos agora projetar um algoritmo para o problema usando o método de divisão e conquista.

Exemplo: Exponenciação

ExponenciaçãoDC(a, n)

```
se n = 0 então retorne(1)
senão
    an := ExponenciaçãoDC(a, n/2)
    an := an * an
    se (n mod 2) = 1 an := an * a
retorne (an)
```

Análise:

Colocar 2 condições de contorno: $n=0$, $n=1$

Exemplo: Busca Binária

BuscaBinaria(A , e , d , x)

Entrada: Vetor A , delimitadores e e d do subvetor e x .

Saída: Índice $1 \leq i \leq n$ tal que $A[i] = x$ ou $i = 0$.

se $e = d$ **então**

se $A[e] = x$ **então** retorne(e) **senão** retorne(-1)

senão

$i := (e + d) / 2$

se $A[i] = x$ **então** retorne(i)

senão se $A[i] > x$

$i := \text{BuscaBinaria}(A, e, i - 1, x)$

senão

$i := \text{BuscaBinaria}(A, i + 1, d, x)$

retorne(i)

Exemplo: Busca Binária

- Análise:
- Caso médio:
 - Cada elemento tem probabilidade $1/n$ de ser o valor procurado.
 - Usar uma árvore para análise.

Exercícios:

- Proponha versões não recursivas para os exemplos acima. A eliminação da recursividade altera a complexidade das soluções?

Exemplo: QuickSort

- Algoritmo de ordenação baseado na estratégia de Dividir e Conquistar
- Em contraste ao Mergesort, no Quicksort é a operação de divisão a mais custosa: depois de escolhermos o *pivot*, temos que separar os elementos do vetor maiores que o *pivot* dos menores que o *pivot*.

Exemplo: QuickSort

- Conseguimos fazer essa divisão com $\Theta(n)$ operações: basta varrer o vetor com dois apontadores, um varrendo da direita para a esquerda e outro da esquerda para a direita, em busca de elementos situados na parte errada do vetor, e trocar um par de elementos de lugar quando encontrado.
- Após essa etapa, basta ordenarmos os dois trechos do vetor recursivamente para obtermos o vetor ordenado, ou seja, a conquista é imediata.

Exemplo: QuickSort

Quicksort(A, esq, dir)

// Entrada: Vetor A de inteiros e os índices esq e dir que delimitam início e fim do subvetor a ser ordenado.

//Saída: Subvetor de A de esq a dir ordenado.

início

 i=esq

 j=dir

 pivô=A[dir]

repita

enquanto (A[i] < pivô) **faça** i= i + 1

enquanto (A[j] > pivô) **faça** j= j - 1

se (i <= j) **então**

 troca (A[i], A[j])

 i = i + 1

 j = j - 1

até_que (i > j)

se (j > esq) **então** QuickSort(A, esq, j)

se (i < dir) **então** QuickSort(A, i, dir)

fim

Exemplo: Quicksort

- **Análise do pior caso:**
- Quantas comparações e quantas trocas o algoritmo Quicksort executa no pior caso?
- Certamente a operação de divisão tem complexidade $\Theta(n)$, mas o tamanho dos dois subproblemas depende do pivot escolhido.
- No pior caso, cada divisão sucessiva do Quicksort separa um único elemento dos demais, recaindo na recorrência:
 - » $T(n) = 0, n = 1$
 - » $T(n) = T(n - 1) + n, n > 1,$
- Portanto, $\Theta(n^2)$ comparações e trocas são executadas no pior caso.
- Então, o algoritmo Quicksort é assintoticamente menos eficiente que o Mergesort no pior caso.
- Veremos que, no caso médio, o Quicksort efetua $\Theta(n \log n)$ comparações e trocas.
- Assim, na prática, o Quicksort é bastante eficiente, com uma vantagem adicional em relação ao Mergesort: é in place, isto é, não utiliza um vetor auxiliar.

Exemplo: Quicksort

- **Análise do caso médio:**
- Considere que i é o índice da posição do *pivot* escolhido no vetor ordenado.
- Supondo que qualquer elemento do vetor tem igual probabilidade de ser escolhido como o *pivot*
- Então, na média, o tamanho dos subproblemas resolvidos em cada divisão sucessiva será ($n \geq 2$):

$$\frac{1}{n} \sum_{i=1}^n (T(i-1) + T(n-i))$$

Exemplo: Quicksort

- Supondo $T(0)=0$, Não é difícil ver que:

$$\sum_{i=1}^n T(i-1) = \sum_{i=1}^n T(n-i) = \sum_{i=1}^{n-1} T(i)$$

- Assim, no caso médio, o número de operações efetuadas pelo Quicksort é dado pela recorrência:

$$T(n) = \begin{cases} 0, & n < 2 \\ \frac{2}{n} \sum_{i=1}^{n-1} T(i) + n - 1, & n \geq 2 \end{cases}$$

- Esta recorrência é $\Theta(n \log n)$. Portanto, na média, o Quicksort executa $\Theta(n \log n)$ trocas e comparações.

Considerações Finais

- Este paradigma não é aplicado apenas a problemas recursivos.
- Existem pelo menos três cenários onde divisão e conquista é aplicado:
 1. Processar independentemente partes do conjunto de dados.
 - Exemplo: Mergesort.
 2. Eliminar partes do conjunto de dados a serem examinados.
 - Exemplo: Pesquisa binária.
 3. Processar separadamente partes do conjunto de dados mas onde a solução de uma parte influencia no resultado da outra.
 - Exemplo: Somador apresentado.

Considerações Finais

- O projeto de algoritmos, é importante procurar sempre manter o **balanceamento** na sub-divisão de um problema em partes menores.
- Divisão e conquista não é a única técnica em que balanceamento é útil.
- Exemplo:
 - Pior caso do quicksort.