Introdução a Complexidade de Algoritmos

Profa. Juliana Kaizer Vizzotto

Complexidade de Algoritmos - Aula 3

Roteiro

- Projeto e Análise do Merge-sort
- Exercícios

Dividir e conquistar: mergesort

```
Merge(A,p,q,r)
n1 <- q - p + 1
n2 <- r - q
criar arrays L[1..n1+1] e R[1 ...n2+1]
for i <- 1 to n1
   do L[i] <- A[p+ i-1]
for j <- 1 to n2
   do R[j] <- A[q+j]
L[n1+1]<- infinito
R[n2+1] <- infinito
i <- 1
j <- 1
...
```

Dividir e conquistar: mergesort

```
...
for k <- p to r
  do if L[i] <= R[j]
        then A[k] <- L[i]
             i <- i + 1
        else A[k] <- R[j]
             j <- j +1
```

Dividir e conquistar: mergesort

Exemplo

- Execute o pseudocódigo acima na chamada Merge(A,9,12,16) quando o subarray A[9..16] contém a sequência $[2, 4, 5, 7, 1, 2, 3, 6]$
- Enuncie o loop invariante para o último for. Prove as três propriedades do loop invariante: inicialização, manutenção e término.
- Qual a complexidade de tempo O para o procedimento Merge?

Dividir e conquistar: mergesort

Merge-Sort

- Agora podemos usar o procedimento Merge como uma subrotina no algoritmo de ordenação por intercalação.
- O procedimento $Merge - Sort(A, p, r)$

```
Merge-Sort(A,p,r)
 if p < r
    then q <- [(p+r)/2]
         Merge-Sort(A,p,q)
         Merge-Sort(A,q+1,r)
         Merge(A,p,q,r)
```

Dividir e conquistar: mergesort

Análise do Algoritmo

- Quando um algoritmo contém uma chamada recursiva a si próprio, seu tempo de execução frequentemente pode ser descrito por uma **equação de recorrência**.
- Essa equação descreve o tempo de execução global sobre um problema de tamanho n em termos do tempo de execução sobre entradas menores.
- Considere $T(n)$ o tempo de execução sobre um problema de tamanho n
- Se o tamanho do problema for pequeno o bastante, digamos $n \leq c$ para alguma constante c, a solução direta demorará um tempo constante.

Dividir e conquistar: mergesort

Análise do Algoritmo

- Vamos supor que o problema seja dividido em a subproblemas, cada um dos quais com $1/b$ do tamanho do problema original (no caso do merge-sorte qual o tamanho de a e b?)
- Se levarmos o tempo $D(n)$ para dividir o problema em subproblemas e o tempo $C(n)$ para combinar as soluções dadas aos subproblemas na solução para o problema original, obteremos:

$$T(n) = \begin{cases} O(1) & sen \leq c \\ aT(n/b) + D(n) + D(n) + C(n) & cc \end{cases}$$

Dividir e conquistar: mergesort

Análise do Algoritmo

- Dividir: a etapa de dividir simplesmente calcula o ponto médio do subarranjo, o que demora um tempo constante. Portante, $D(n) = O(1)$
- Conquistar: Resolvemos recursivamente dois subproblemas, cada um tem tamanho $n/2$ e contribui com $2T(n/2)$ para o tempo de execução
- Combinar: Já observamos que o procedimento Merge em um subarranjo de n elementos leva o tempo $O(n)$. Logo $C(n) = O(n)$.

Dividir e conquistar: mergesort

Análise do Algoritmo

- Somando $D(n)$ e $C(n)$ estamos somando uma função que é $O(n)$ a uma função que é $O(1)$. Logo $O(n)$.
- A adição dessa função ao termo $2T(n/2)$ da etapa de conquistar fornece a **recorrência** para o tempo de execução do pior caso $T(n)$:

$$T(n) = \begin{cases} O(1) & sen \leq c \\ 2T(n/2) + O(n) & cc \end{cases}$$

- A solução dessa equação de recorrência nos diz que $T(n) = O(n \log_2 n)$

Dividir e conquistar: mergesort

Análise do Algoritmo

- Entendento intuitivamente a recorrência:

$$T(n) = \begin{cases} O(1) & sen \leq c \\ 2T(n/2) + cn & cc \end{cases}$$

- A constante c representa o tempo exigido para resolver problemas de tamanho 1
- Vamos pensar na árvore de chamadas recursivas!!
- O número total de níveis de uma árvore de recursão para n é $\log_2 n + 1$ (pense na prova por indução!!)
- Nos fornece uma árvore de tamanho $\log_2 n$

Dividir e conquistar: mergesort

Análise do Algoritmo

- Caso básico ocorre quando $n = 1$, e nesse caso somente há um nível. Como $\log_2 1 = 0$, temos que $\log_2 n + 1$ fornece o número correto de níveis.
- Agora suponha, como hipótese indutiva, que o número de níveis para 2^i nós seja $\log_2 2^i + 1 = i + 1$. Consideramos o tamanho da próxima entrada como 2^{i+1}. Uma árvore com 2^{i+1} nós tem um nível a mais em relação a uma árvore de 2^i nós, e então o número total de níveis é $(i + 1) + 1 = \log_2 2^{i+1} + 1$