Universidade Federal de Ouro Preto
Departamento de Computação - DECOM
BCC241 - Projeto e Análise de Algoritmos
Prof. Anderson Almeida Ferreira

Nome: Caio Silas de Araujo Amaro
Matrícula: 21.1.4111

a) Encontrar o maior valor em um vetor.

Forma 1:

Usando o algoritmo selection(S, k) do slide 36 da aula sobre divisão e conquista, use k igual a última posição do vetor, ou seja, k = n. Ou seja, procure o n-ésimo menor valor do vetor de tamanho n, que é o maior elemento do vetor. A complexidade do algoritmo selection é O(n).

Forma 2:

Divida o vetor na metade e procure recursivamente o maior elemento dos dois subvetores. Depois disso, retorne o maior entre eles.

```
function maxValue(S, i, f)
Entrada: um vetor de inteiros, o início e final do intervalo de busca
Saída: maior número do intervalo

n <- f-i+1
if (n <= 0):
        return (-infinito)
if (n == 1):
        return S[i]

meio <- (i+f)/2

maxE <- maxValue(S, i, meio)
maxD <- maxValue(S, meio+1, f)

return max(maxE, maxD)
```

Esse algoritmo usa a estratégia de divisão e conquista para resolver o problema. A divisão tem tempo constante, pois apenas indicamos os índices, a conquista é $T(n/2)$ e são feitas duas chamadas recursivas, a combinação é em tempo constante, pois apenas uma comparação é feita. Por isso, a complexidade desse algoritmo é $T(n) = 2T(n/2) + O(1)$. Pelo teorema mestre: $a = 2$, $b = 2$ e $d = 0$. Então, $\log_b a = \log_2 2 = 1 > 0 = d$. Logo $T(n) = O(n)$. A complexidade do algoritmo é $O(n)$.

b) Encontrar o maior e o menor elemento em um vetor.

Forma 1:

Faça duas chamadas do algoritmo selection(S, k) do slide 36 da aula sobre divisão e conquista, uma com k igual a primeira posição do vetor (k = 1) para obter o menor número do vetor e outra com k igual a última posição do vetor (k = n) para obter o maior número do vetor.

A complexidade do algoritmo *selection* é O(n). Logo, temos O(n) + O(n) = O(n). A complexidade do algoritmo é O(n).

Forma 2:

Divida o vetor na metade e procure recursivamente o menor e o maior elemento dos dois subvetores. Depois disso, retorne o menor entre os dois menores e o maior entre os dois maiores.

```
function minMax(S, i, f)
Entrada: um vetor de inteiros, o início e final do intervalo de busca
Saída: o menor e maior número do intervalo

n <- f-i+1
if (n <= 0):
        return (infinito, -infinito)
if (n == 1):
        return (S[i], S[i])

meio <- (i+f)/2

(minE, maxE) <- minMax(S, i, meio)
(minD, maxD) <- minMax(S, meio+1, f)

return (min(minE, minD), max(maxE, maxD))
```

Esse algoritmo usa a estratégia de divisão e conquista para resolver o problema. A divisão tem tempo constante, pois apenas indicamos os índices, a conquista é T(n/2) e são feitas duas chamadas recursivas, a combinação é em tempo constante, pois são feitas duas comparações. Por isso, a complexidade desse algoritmo é $T(n) = 2T(n/2) + O(1)$. Pelo teorema mestre: $a = 2$, $b = 2$ e $d = 0$. Então, $\log_b a = \log_2 2 = 1 > 0 = d$. Logo $T(n) = O(n)$. A complexidade do algoritmo é O(n).

c) Exponenciação.

Sejam a, b número inteiros e a^b a exponenciação que deseja calcular. Calcule o valor de $a^{b/2}$ recursivamente e multiplique por ele mesmo. Caso b seja ímpar, multiplique o resultado por a.

```
function exponenciacao(b, e)
Entrada: Dois inteiros “b” e “e”
Saída: o resultado de b^e

if (e == 0):
        return 1;

n <- exponenciacao(b, e/2)

n <- n * n

if (n é ímpar):
        n <- n * b

return n
```

Esse algoritmo usa a estratégia de divisão e conquista para resolver o problema. A divisão tem tempo constante, pois apenas dividimos e por 2, a conquista é T(n/2) e apenas uma chamada recursiva é realizada, a combinação é em tempo constante, pois ocorrem 2 multiplicações se e for ímpar e 1 se e for par. Por isso, a complexidade desse algoritmo é $T(n) = T(n/2) + O(1)$. Pelo teorema mestre: $a = 1$, $b = 2$ e $d = 0$. Então, $\log_b a = \log_2 1 = 0 = d$. Logo $T(n) = O(\log n)$. A complexidade do algoritmo é $O(\log n)$.