Estruturas de Dados e Algoritmos

Exame de Época Normal

José Jasnau Caeiro

21 de junho de 2022

LER ATENTAMENTE

Submeta a resposta desta questão, nos próximos **30 minutos**. Submeta no local *moodle* do IPBeja[1] usando o seu acesso de aluno.

Entregue só um ficheiro, usando a linguagem de programação `Julia`, designado por `main.jl` que engloba todas as suas respostas e contendo todo o código necessário para demonstrar que atingou uma resposta correta.

Utilize o código no ficheiro `ordenacao.jl` que é fornecido para auxiliar na resolução do exame.

Se utilizar *packages* externos indique sob a forma de comentário qual o *package*. Refira todas as fontes consultadas sob a forma de comentário na sua resposta.

No caso de não referir as fontes consultadas será considerada tentativa de fraude. E se existirem respostas com estrutura de código similar, de 2 ou mais alunos, também será considerada fraude e anulado todo o exame.

Identifique claramente:

- o seu nome completo;
- o seu número de aluno;
- a alínea a que respondeu[2].

Leia com calma a questão e tente não depender excessivamente da informação na *Internet*.

Questão 1

Crie uma pasta para a questão designada por `edaq1`:

1.a) (1 valor) Use a função `a_sua_sina()`, representada na Fig. 1, passando em argumento uma *string* com o seu nome completo, escrito sem acentos nem cedilhas.

O resultado desta função indica-lhe os algoritmos de ordenação que deve usar nas questões seguintes. Escreva uma função que imprime na consola o resultado da chamada da função `a_sua_sina()` com o seu nome.

[1] `http://cms.ipbeja.pt`

[2] Caso não o faça pode ser anulada a resposta.

1.b) (2 valores) Escreva código que representa um ponto no espaço cartesiano bidimensional, com coordenadas do tipo `Float64` através dum `mutable struct`. Programe um método que calcula o módulo do vetor distância dum ponto P à origem O.

1.c) (2 valores) Modifique o código correspondente ao primeiro algoritmo de ordenação que lhe foi determinado na primeira alínea, vidé as Figs. 2, 3, 4, 5, 6 e 7. Programe de modo a ordenar tabelas de elementos do tipo `Ponto3D`. Designe a função de ordenação por `sort_ponto`. Utilize como métrica de comparação da ordenação o módulo/norma do vetor formado pelo ponto e a origem do sistema de coordenadas. Teste a ordenação com 10 números. Imprima o resultado na consola.

1.d) (2 valores) Meça $M = 30$ vezes o tempo de execução dos 2 algoritmos de ordenação que lhe foram atribuídos, para tabelas em que as chaves são do tipo $Float64$.

Forme duas tabelas: a primeira designada por X contém as medidas dos tempos de execução do primeiro algoritmo, e a segunda, designada por Y, contém as medidas dos tempos de execução do segundo algoritmo.

Calcule uma estimativa da média do tempo de execução, para uma dimensão de tabela N suficientemente grande para que o tempo de execução seja de pelo menos um segundo para cada amostra.

```
"""
    a_sua_sina( s )
calcula a sua sina...
o argumento s não deve ter acentos nem cedilhas
"""
function a_sua_sina( s::String )
    N = 5
    soma = sum( [ Int( c ) for c in s ] ) % N
    if soma == 0
        a = "insertion"
    elseif soma == 1
        a = "bubble"
    elseif soma == 2
        a = "merge"
    elseif soma == 3
        a = "heap"
    else
        a = "quick"
    end

    try
        meia = sum( [ Int( s[ k ] ) + 1
                      for k = 1:Int(floor(length( s ) / 2)) ] ) % N
    catch e
        println( "escreva o seu nome completo" )
        println( "SEM USAR ACENTOS NEM CEDILHAS" )
        return
    end

    meia = sum( [ Int( s[ k ] ) + 1
                  for k = 1:Int(floor(length( s ) / 2)) ] ) % N

    meia = (meia != soma) ? meia : (( meia + 1 > N) ? 0 : meia)

    if meia == 0
        b = "insertion"
    elseif meia == 1
        b = "bubble"
    elseif meia == 2
        b = "merge"
    elseif meia == 3
        b = "heap"
    else
        b = "quick"
    end
    a, b
end
```

Figura 1: Função que determina o algoritmo de ordenação a usar.

```
"""
    insertion_sort!( A::Array{Int16} )
algoritmo de ordenação insertion sort
"""
function insertion_sort!( A::Array{Int16} )
    for j = 2:length(A)
        key = A[ j ]
        i = j - 1
        while i > 0 && A[ i ] > key
            A[ i + 1 ] = A[ i ]
            i = i - 1
        end
        A[ i + 1 ] = key
    end
end
```

Figura 2: Insertion-Sort

```
"""
    bubble_sort!( A )
ordenação da tabela A pelo algoritmo bubble sort
"""
function bubble_sort!( A::Array{Int16} )
    for i = 1:length( A ) - 1
        for j = length( A ):-1:i+1
            if A[ j ] < A[ j - 1 ]
                A[ j ], A[ j - 1 ] = A[ j - 1 ], A[ j ]
            end
        end
    end
end
```

Figura 3: Bubble-Sort

```
"""
    merge!( A )
chamada para a ordenação da tabela A pelo algoritmo merge sort
"""
function merge!( A::Array{Int16}, p, q, r )
    n = q - p + 1
    n = r - q
    L = Array{Int16}(undef, n +1)
    R = Array{Int16}(undef, n +1)
    for i = 1:n
        L[ i ] = A[ p + i - 1 ]
    end
    for j = 1:n
        R[ j ] = A[ q + j ]
    end
    L[ n  + 1 ] = typemax(Int16)
    R[ n  + 1 ] = typemax(Int16)
    i = 1
    j = 1
    for k = p:r
        if L[ i ] <= R[ j ]
            A[ k ] = L[ i ]
            i = i + 1
        else
            A[ k ] = R[ j ]
            j = j + 1
        end
    end
end

"""
    merge_sort!( A )
ordenação da tabela A pelo algoritmo merge sort
"""
function merge_sort!( A::Array{Int16}, p, r )
    if p < r
        q = convert(Int64,
                    floor( (convert(Float64, p) +
                        convert(Float64, r)) / 2.0 ))
        merge_sort!( A, p, q )
        merge_sort!( A, q+1, r )
        merge!( A, p, q, r )
    end
end
```

Figura 4: Merge-Sort

```
"""
    left( i )
parte da ordenação pelo algoritmo heap sort
"""
function left( i )
    2 * i
end

"""
    right( i )
parte da ordenação pelo algoritmo heap sort
"""
function right( i )
    2 * i + 1
end

mutable struct HeapArray
    A::Array{Int64}
    heap_size
    HeapArray( A, n ) = new(A, n)
end

"""
    max_heapify!( A::Array{Int16}, i )
parte da ordenação pelo algoritmo
heap sort
"""
function max_heapify!( H::HeapArray, i )
    l = left( i )
    r = right( i )
    if l <= H.heap_size && H.A[ l ] > H.A[ i ]
        largest = l
    else
        largest = i
    end
    if r <= H.heap_size && H.A[ r ] > H.A[ largest ]
        largest = r
    end
    if largest != i
        H.A[ i ], H.A[ largest ] = H.A[ largest ], H.A[ i ]
        max_heapify!( H, largest )
    end
end
```

Figura 5: HEAP-SORT (parte 1)

```
"""
    build_max_heap!( H )
parte da ordenação pelo algoritmo heap sort
"""
function build_max_heap!( H::HeapArray )
    H.heap_size = length( H.A )
    N = convert(Int64, floor(convert(Float64, length( H.A )) / 2.0))
    for i = N:-1:1
        max_heapify!(H, i)
    end
end

"""
    heap_sort!( A )
ordenação pelo algoritmo heap sort
"""
function heap_sort!( A::Array{Int16} )
    H = HeapArray(A, length( A ))
    build_max_heap!( H )
    for i = length( H.A ):-1:2
        H.A[ 1 ], H.A[ i ] = H.A[ i ], H.A[ 1 ]
        H.heap_size = H.heap_size - 1
        max_heapify!(H, 1)
    end
    A::Array{Int16} = H.A
end
```

Figura 6: HEAP-SORT (parte 2)

```
"""
    partition( A, p, r )

parte da ordenação da tabela A pelo algoritmo
quick sort
"""
function partition( A::Array{Int16}, p, r )
    x = A[ r ]
    i = p - 1
    for j = p:r-1
        if A[ j ] <= x
            i = i + 1
            A[ i ], A[ j ] = A[ j ], A[ i ]
        end
    end
    A[ i + 1 ], A[ r ] = A[ r ], A[ i + 1 ]
    i + 1
end

"""
    quick_sort!( A, p, r )

ordenação da tabela A pelo algoritmo
quick sort
"""
function quick_sort!( A::Array{Int16}, p, r )
    if p < r
        q = partition( A, p, r )
        quick_sort!( A, p, q - 1 )
        quick_sort!( A, q + 1, r )
    end
end
```

Figura 7: QUICK-SORT