Construção e análise de algoritmos

Trabalho 02

Equipe:

- Eduardo Façanha
- Giovanni Brígido
- João David
- Maurício
- Nicole Rocha
- Tiago Ponte

Todo o código está acessível em:

https://github.com/eduardodut/projetos-algoritmos/tree/master/Projeto%202
(https://github.com/eduardodut/projetos-algoritmos/tree/master/Projeto%202)

a)

Realizar a leitura do arquivo de entrada, armazenando os dados dos contratos (0,5 ponto) (Data de entrega: 16/06):

1. Em um vetor de objetos do tipo contrato; e
2. Em uma matriz, indexada pelo fornecedor, pelo mês de início e pelo mês de fim do contrato, onde cada elemento armazena o valor do contrato respectivo.

Import do código desenvolvido para o trabalho

In [296]:

```
include("Algoritmos/Algoritmos_Ordenacao.jl")
include("Algoritmos/Inicializacao.jl")
include("Algoritmos/Saida.jl")
include("Algoritmos/Grafico.jl")

using BenchmarkTools, Juno, DataFrames, Plots, StatsPlots

insertionsort! = Algoritmos_Ordenacao.insertionsort!
mergesort! = Algoritmos_Ordenacao.mergesort!
heapsort! = Algoritmos_Ordenacao.heapsort!
quicksort!(vetor::Vector{Tuple{Int64,Int64,Int64,Float64}})::Vector{Tuple{Int64,Int64,I
nt64,Float64}} = Algoritmos_Ordenacao.quicksort!(vetor,1,length(vetor))


formata_ms = Saida.formata_ms
formata_s = Saida.formata_s
formata_casas = Saida.formata_casas_decimais

teste_ordenacao(vetor::Vector{Tuple{Int64,Int64,Int64,Float64}})::Bool = issorted(DataF
rame(vetor)[!, 4]) ? true : false

function calculo_tempo_medio(vetor::Vector{Tuple{Int64,Int64,Int64,Float64}}, iteracoes
::Int64, funcao_ordenacao)::Float64
    tempo = 0.0
    Juno.@progress for i in 1:iteracoes
      entrada = copy(vetor)
      tempo = tempo + @elapsed funcao_ordenacao(entrada)
    end
    return tempo / Float64(iteracoes)
end
```

```
WARNING: replacing module Algoritmos_Ordenacao.
WARNING: replacing module Inicializacao.
WARNING: replacing module Saida.
WARNING: replacing module Grafico.
```

Out[296]:

```
calculo_tempo_medio (generic function with 1 method)
```

Inicialização das variáveis

In [297]:

```
var = Inicializacao.inicializar_variaveis("Dados/entrada.txt")
const entrada_matricial = var[1]
const entrada_vetorizada = var[2]
const m = var[3]
const n = var[4]
```

```
WARNING: redefining constant entrada_matricial
WARNING: redefining constant entrada_vetorizada
```

Out[297]:

```
120
```

b)

Explicar qual das duas maneiras de armazenamento dos dados é mais eficiente, em relação ao consumo de memória, tanto em termos assintóticos quanto em termos de absolutos. (Data de entrega: 18/06) (0,5 ponto)

In [298]:

```
"""
O número de posições da entrada vetorizada é 4*m*n*(n+1)/2, onde 4 é o número de dados
 por contrato
e o restante da expressão representa o número de linhas do arquivo.
A estrutura é densa, todas as posições são ocupadas, pois todos os dados do contrato sã
o guardados nessa estrutura.
Assintóticamente, a complexida de espaço é: Θ(2mn(n+1)) -> Θ(mn²)
"""
n_posicoes_entrada_vetorizada = 4*length(entrada_vetorizada)
println("n_posicoes_entrada_vetorizada :",n_posicoes_entrada_vetorizada)

"""
A entrada matricial é um vetor de 3 dimensões, onde cada dimensão é representada pela e
mpresa,
mes_inicial e mes_final do contrato, e apenas o valor do contrato é guardado nessa estr
utura.

entrada_matricial[empresa, mes_inicial, mes_final] = valor

entrada_matricial[100, 120, 120]

A estrutura é esparsa, pois reserva espaços para contratos que não existem. No total sã
o reservadas
m*n^2 posições de memória.
Assintóticamente, a complexida de espaço é: Θ(mn²)
"""

n_posicoes_entrada_matricial = length(entrada_matricial)
println("n_posicoes_entrada_matricial :",n_posicoes_entrada_matricial)
print(n_posicoes_entrada_matricial/n_posicoes_entrada_vetorizada)
```

```
n_posicoes_entrada_vetorizada :2904000
n_posicoes_entrada_matricial :1440000
0.49586776859504134
```

c)

Implementar o método de ordenação Insertion Sort para ordenar o vetor de objetos do tipo contrato. Criar um clone do vetor (simplesmente para não alterar a organização original dos dados) e aplicar o método de ordenação, exibindo ao final o tempo total de execução da ordenação. (Data de entrega: 16/06) (1,0 ponto)

In [299]:

```
entrada = copy(entrada_vetorizada)
tempo_insertionsort = @elapsed ordenado_insertion_sort =
    insertionsort(entrada)
println(" ")
```

In [300]:

```
println("Duração: ", formata_casas(tempo_insertionsort), " s.")
```

```
Duração: 383.27 s.
```

In [301]:

```
print("Está ordenado? ", insertionsort(ordenado_insertion_sort) |> teste_ordenacao ? "s
im" : "não")
```

```
Está ordenado? sim
```

d)

Apresentar a complexidade da função descrita no item anterior, fazendo uso de notação assintótica e tendo como parâmetros somente a quantidade n de meses e a quantidade m de fornecedores. (Data de entrega: 18/06) (0,5 ponto)

$\Theta(m^2n^4)$, pois o número de linhas do arquivo é função do número de empresas (m) e do número de meses (n). Para um caso genérico o insertion sort possui o melhor caso **$\Theta(n)$** e pior caso **$\Theta(n^2)$**.

e)

Implementar o método de ordenação **Mergesort** para ordenar o vetor de objetos do tipo contrato. Criar um clone do vetor e aplicar o método de ordenação, exibindo ao final o tempo total de execução da ordenação. (Data de entrega: 23/06) (1,0 ponto)

In [302]:

```
tempo_mergesort = calculo_tempo_medio(entrada_vetorizada, 100, mergesort)
println("Duração média: ", formata_casas(tempo_mergesort*1000.0), " ms.")
```

```
Duração média: 2643.12 ms.
```

In [323]:

```
print("Está ordenado? ", mergesort(copy(entrada_vetorizada)) |> teste_ordenacao ? "sim"
: "não")
```

```
Está ordenado? sim
```

f)

Apresentar a complexidade da função descrita no item anterior, fazendo uso de notação assintótica e tendo como parâmetros somente a quantidade n de meses e a quantidade m de fornecedores. (Data de entrega: 23/06) (0,5 ponto)

""" O merge sort é um exemplo de algoritmo de ordenação por comparação do tipo dividir-para-conquistar. Sua ideia básica consiste em Dividir (o problema em vários subproblemas e resolver esses subproblemas através da recursividade)
e Conquistar (após todos os subproblemas terem sido resolvidos ocorre a conquista que é a união das resoluções dos subproblemas).
A complexida: Θ(mn² log(mn²)).

"""

g)

Implementar o método de ordenação **Heapsort** para ordenar o vetor de objetos do tipo contrato. Criar um clone do vetor e aplicar o método de ordenação, exibindo ao final o tempo total de execução da ordenação. (Data de entrega: 23/06) (1,0 ponto)

In [304]:

```
tempo_heapsort = calculo_tempo_medio(entrada_vetorizada, 100, heapsort)
println("Duração média: ", formata_casas(tempo_heapsort*1000.0), " ms.")
```

```
Duração média: 239.33 ms.
```

In [305]:

```
print("Está ordenado? ", heapsort(copy(entrada_vetorizada)) |> teste_ordenacao ? "sim"
: "não")
```

```
Está ordenado? sim
```

h)

Apresentar a complexidade da função descrita no item anterior, fazendo uso de notação assintótica e tendo como parâmetros somente a quantidade n de meses e a quantidade m de fornecedores. (Data de entrega: 23/06) (0,5 ponto)

""" O algoritmo heapsort é um algoritmo de ordenação generalista, e faz parte da família de algoritmos de ordenação por seleção.
Tem um desempenho em tempo de execução muito bom em conjuntos ordenados aleatoriamente, tem um uso de memória bem comportado
e o seu desempenho em pior cenário é praticamente igual ao desempenho em cenário médio. Como o Insertion Sort, mas diferentemente
do Merge Sort, o Heapsort é um algoritmo do tipo local: apenas um número constante de elementos do array é armazenado fora do
array de entrada durante a sua execução. Assim, o Heapsort combina os melhores atributos dos dois algoritmos de classificação citados.
A complexida: Θ(mn² lg(mn²)). """

i)

Implementar o método de ordenação **Quicksort** para ordenar o vetor de objetos do tipo contrato. Criar um clone do vetor e aplicar o método de ordenação, exibindo ao final o tempo total de execução da ordenação. (Data de entrega: 23/06) (1,0 ponto)

In [306]:

```
tempo_quicksort = calculo_tempo_medio(entrada_vetorizada, 100, quicksort)
println("Duração média: ", formata_casas(tempo_quicksort*1000), " ms.")
```

```
Duração média: 143.13 ms.
```

In [307]:

```
print("Está ordenado? ", quicksort(copy(entrada_vetorizada)) |> teste_ordenacao ? "sim"
: "não")
```

```
Está ordenado? sim
```

j)

Apresentar a complexidade da função descrita no item anterior, fazendo uso de notação assintótica e tendo como parâmetros somente a quantidade n de meses e a quantidade m de fornecedores. (Data de entrega: 23/06) (0,5 ponto)

""" O quicksort adota a estratégia de divisão e conquista. A estratégia consiste em rearranjar as chaves de modo que as chaves
"menores" precedam as chaves "maiores". Em seguida o quicksort ordena as duas sublistas de chaves menores e maiores recursivamente até que a lista completa se encontre ordenada. Complexidade: $\Theta(mn^2 \log(mn^2)$

Complexida de pior caso genérico: $\Theta(n^2)$.
Complexidade de melhor caso genérico: $\Theta(n\log(n))$. """

k)

Baseando-se nas ideias dos algoritmos apresentados, criar um método de ordenação que, entretanto, explore as regras de formação de preços de contratos de um fornecedor (expressas na matriz de contratos), para realizar a ordenação dos contratos pelo valor, exibindo ao final o tempo total de execução da ordenação. (Data de entrega: 25/06) (2,0 pontos)

A solução desenvolvida consiste em separar as o arquivo em 100 vetores, através do cálculo dos índices de início e fim de cada empresa, aproveitando-se da estrutura de formação do arquivo, e aplicando um algoritmo de ordenação. Após a ordenação individual é realizado o merge dos 100 vetores através do método de heap mínimo.

Um dos parâmetros da função de divisão em empresa é o algoritmo de ordenação que será aplicado. Para a apresentação foi escolhido o insertionsort como algoritmo de ordenação individual.

Por utilizarmos a estratégia de divisão do problema em problemas independentes, foi implementado um argumento booleano para selecionar ou não a execução em paralelo da ordenação individual de cada empresa. Os resultados são mostrados nas células a seguir.

In [308]:

```
#criação da função dividir_ordenar_intercalar através da composição da estratégia de di
visão entre empresas e a aplicação da intercalação de k vetores.

dividir_ordenar_intercalar(
    entrada_vetorizada::Vector{Tuple{Int64,Int64,Int64,Float64}},
    funcao_ordenacao,
    paralelizar::Bool,
)::Vector{Tuple{Int64,Int64,Int64,Float64}} =
    Algoritmos_Ordenacao.dividir_e_ordenar(
        entrada_vetorizada,
        funcao_ordenacao,
        paralelizar,
    ) |> Algoritmos_Ordenacao.intercalar_k_vetores_ordenados

function calculo_tempo_divisao_ordenacao(::Vector{Tuple{Int64,Int64,Int64,Float64}},
    iteracoes::Int64,
    funcao_ordenacao,
    paralelizar::Bool)::Float64
  t = 0.0
  for i in 1:100
    entrada = copy(entrada_vetorizada)
    t = t + @elapsed dividir_ordenar_intercalar(entrada, funcao_ordenacao, paralelizar)
  end

  return t/Float64(iteracoes)


end
```

Out[308]:

```
calculo_tempo_divisao_ordenacao (generic function with 1 method)
```

In [309]:

```
println("Previamente, o insertionsort completou a execução em ",formata_casas(tempo_ins
ertionsort)," s.")
```

```
Previamente, o insertionsort completou a execução em 383.27 s.
```

In [319]:

```
#sem paralelismo
novo_tempo_insertionsort = calculo_tempo_divisao_ordenacao(entrada_vetorizada, 100, ins
ertionsort!, false)
println("Duração média: ", formata_casas(novo_tempo_insertionsort*1000), " ms.")
```

```
Duração média: 190.68 ms.
```

In [320]:

```
#com paralelismo
novo_tempo_insertionsort = calculo_tempo_divisao_ordenacao(entrada_vetorizada, 100, ins
ertionsort!, true)
println("Duração média: ", formata_casas(novo_tempo_insertionsort*1000), " ms.")
```

```
Duração média: 155.05 ms.
```

Testes de ordenação

In [312]:

```
println("Ordenado? ", dividir_ordenar_intercalar(copy(entrada_vetorizada), insertionsor
t, false) |> teste_ordenacao ? "sim" : "não")
```

```
Ordenado? sim
```

I)

Apresentar a complexidade da função descrita no item anterior, fazendo uso de notação assintótica e tendo como parâmetros somente a quantidade n de meses e a quantidade m de fornecedores. (Data de entrega: 25/06) (1,0 ponto)

pior caso $\Theta(mn^4)$ melhor caso $\Theta(mn^2log(m))$