Construção e análise de algoritmos

Trabalho 02

Equipe:

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

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 [ ]:

```
include("Algoritmos/Algoritmos_Ordenacao.jl")
include("Algoritmos/Ordenacao_Estrutural.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 = Algoritmos_Ordenacao.quicksort

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

Inicialização das variáveis

In [ ]:

```
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]
```

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 [30]:

```
"""
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: 0(2mn(n+1)) -> 0(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: 0(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 [10]:

```
@elapsed ordenado_insertion_sort =
    insertionsort(entrada_vetorizada)
```

Out[10]:

```
388.934654701
```

In [16]:

```
print("Está ordenado? ", insertionsort(entrada_vetorizada) |> teste_ordenacao ? "sim" :
"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 [22]:

```
@benchmark ordenado_merge_sort = mergesort($entrada_vetorizada)
```

Out[22]:

```
BenchmarkTools.Trial:
  memory estimate:  2.54 GiB
  allocs estimate:  7969925
  --------------
  minimum time:     2.182 s (25.07% GC)
  median time:      2.235 s (26.95% GC)
  mean time:        2.291 s (28.15% GC)
  maximum time:     2.455 s (31.97% GC)
  --------------
  samples:          3
  evals/sample:     1
```

In [13]:

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

```
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)

In [ ]:

```
"""
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 esse
s 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: 0(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 [14]:

```
@benchmark ordenado_heap_sort = heapsort($entrada_vetorizada)
```

Out[14]:

```
BenchmarkTools.Trial:
  memory estimate:  30.46 MiB
  allocs estimate:  6
  --------------
  minimum time:     197.728 ms (0.00% GC)
  median time:      235.683 ms (0.00% GC)
  mean time:        239.688 ms (1.40% GC)
  maximum time:     313.748 ms (14.10% GC)
  --------------
  samples:          21
  evals/sample:     1
```

In [20]:

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

```
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)

In [ ]:

```
"""
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: 0(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 [36]:

```
@benchmark quicksort($entrada_vetorizada)
```

Out[36]:

```
BenchmarkTools.Trial:
  memory estimate:  0 bytes
  allocs estimate:  0
  --------------
  minimum time:     48.812 ms (0.00% GC)
  median time:      58.962 ms (0.00% GC)
  mean time:        63.197 ms (0.00% GC)
  maximum time:     112.097 ms (0.00% GC)
  --------------
  samples:          80
  evals/sample:     1
```

In [24]:

```
print("Está ordenado? ", quicksort(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)

In [ ]:

```
"""
O quicksort adota a estratégia de divisão e conquista. A estratégia consiste em rearran
jar 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: Θ(mn² log(mn²)

Complexida de pior caso genérico: Θ(n²).
Complexidade de melhor caso genérico: Θ(nlog(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)

Foram desenvolvidas duas soluções. A primeira solução separa 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 devidamente através do método de heap mínimo.

Um dos parâmetros da função de divisão em empresa é o algoritmo de ordenação que será aplicado, portanto, cada algoritmo previamente demonstrado foi aplicado com o objetivo de verificar se há ganho de performance.

A segunda solução se baseia em observar que, na estrutura matricial, a tabela correspondente de cada empresa está organizada de forma que os valores de uma coluna decresce das posições superiores para as inferiores, e os valores de cada linha crescem da esquerda para direita.

Para aproveitar esse ordenamento existente foi construído um Heap max de listas linkadas para realizar a intercalação dos valores das colunas. As colunas individualmente são tratadas como listas linkadas, onde a primeira posição é ocupada pelo maior item, que por sua vez são os componentes do Heap max. O ordenamento da direita para esquerda crescente dos valores do topo de cada coluna é aproveitado para economizar o uso da função de construção do heap max.

A ordenação final das empresas inidivdualmente ordenadas dessa forma é realizada através de heap min, como na primeira solução.

Para ambas as soluções, por se utilizarem da estratégia de divisão do problema, 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.

Primeira solução

A seguir são mostrados os resultados de cada algoritmo utilizando a estratégia de divisão de empresas. É utilizada a estrutura vetorial para a execução da solução.

In [ ]:

```
#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
```

Mergesort

Previamente o mergesort completou a execução numa média de 2.3s.

In [42]:

```
#com paralelismo
@benchmark dividir_ordenar_intercalar($entrada_vetorizada, $mergesort, $true)
```

Out[42]:

```
BenchmarkTools.Trial:
  memory estimate:  2.08 GiB
  allocs estimate:  6850732
  --------------
  minimum time:     1.689 s (15.52% GC)
  median time:      2.247 s (34.26% GC)
  mean time:        2.303 s (33.14% GC)
  maximum time:     2.972 s (42.31% GC)
  --------------
  samples:          3
  evals/sample:     1
```

In [44]:

```
#sem paralelismo
@benchmark dividir_ordenar_intercalar($entrada_vetorizada, $mergesort, $false)
```

Out[44]:

```
BenchmarkTools.Trial:
  memory estimate:  1.72 GiB
  allocs estimate:  6730780
  --------------
  minimum time:     1.932 s (22.82% GC)
  median time:      2.103 s (25.80% GC)
  mean time:        2.176 s (28.05% GC)
  maximum time:     2.493 s (34.00% GC)
  --------------
  samples:          3
  evals/sample:     1
```

In [45]:

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

```
Está ordenado? sim
```

Heapsort

Previamente o heapsort completou a execução numa média de 239.7ms.

In [49]:

```
@com paralelismo
@benchmark dividir_ordenar_intercalar($entrada_vetorizada, $heapsort, $true)
```

Out[49]:

```
BenchmarkTools.Trial:
  memory estimate:  68.39 MiB
  allocs estimate:  1542
  --------------
  minimum time:     59.021 ms (0.00% GC)
  median time:      73.144 ms (0.00% GC)
  mean time:        89.568 ms (13.62% GC)
  maximum time:     251.943 ms (62.03% GC)
  --------------
  samples:          57
  evals/sample:     1
```

In [47]:

```
#sem paralelismo
@benchmark dividir_ordenar_intercalar($entrada_vetorizada, $heapsort, $false)
```

Out[47]:

```
BenchmarkTools.Trial:
  memory estimate:  68.32 MiB
  allocs estimate:  1026
  --------------
  minimum time:     84.938 ms (0.00% GC)
  median time:      106.170 ms (0.00% GC)
  mean time:        109.598 ms (3.97% GC)
  maximum time:     175.629 ms (31.18% GC)
  --------------
  samples:          46
  evals/sample:     1
```

In [55]:

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

```
Está ordenado? sim
```

Quicksort

Previamente o quicksort completou a execução numa média de 63.19ms.

In [50]:

```
#com paralelismo
@benchmark dividir_ordenar_intercalar($entrada_vetorizada, $quicksort, $true)
```

Out[50]:

```
BenchmarkTools.Trial:
  memory estimate:  42.47 MiB
  allocs estimate:  1032
  --------------
  minimum time:     44.617 ms (0.00% GC)
  median time:      56.067 ms (0.00% GC)
  mean time:        62.912 ms (5.86% GC)
  maximum time:     171.732 ms (0.00% GC)
  --------------
  samples:          80
  evals/sample:     1
```

In [51]:

```
#sem paralelismo
@benchmark dividir_ordenar_intercalar($entrada_vetorizada, $quicksort, $false)
```

Out[51]:

```
BenchmarkTools.Trial:
  memory estimate:  42.40 MiB
  allocs estimate:  516
  --------------
  minimum time:     61.273 ms (0.00% GC)
  median time:      76.649 ms (0.00% GC)
  mean time:        80.125 ms (3.99% GC)
  maximum time:     114.641 ms (18.36% GC)
  --------------
  samples:          63
  evals/sample:     1
```

In [56]:

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

```
Está ordenado? sim
```

Insertion sort

O insertion sort previamente completou a execução em 388.93 segundos.

In [33]:

```
#com paralelismo
@benchmark dividir_ordenar_intercalar(
    $entrada_vetorizada,
    $insertionsort,
    $true,
)
```

Out[33]:

```
BenchmarkTools.Trial:
  memory estimate:  61.31 MiB
  allocs estimate:  1202
  --------------
  minimum time:     31.601 ms (0.00% GC)
  median time:      35.243 ms (0.00% GC)
  mean time:        43.276 ms (12.96% GC)
  maximum time:     166.260 ms (48.61% GC)
  --------------
  samples:          116
  evals/sample:     1
```

In [35]:

```
@sem paralelismo
@benchmark dividir_ordenar_intercalar(
    $entrada_vetorizada,
    $insertionsort,
    $false,
)
```

Out[35]:

```
BenchmarkTools.Trial:
  memory estimate:  61.24 MiB
  allocs estimate:  686
  --------------
  minimum time:     31.488 ms (0.00% GC)
  median time:      35.457 ms (0.00% GC)
  mean time:        41.127 ms (10.14% GC)
  maximum time:     90.914 ms (49.76% GC)
  --------------
  samples:          122
  evals/sample:     1
```

In [41]:

```
print("Está ordenado? ", dividir_ordenar_intercalar(entrada_vetorizada, insertionsort,
false) |> teste_ordenacao ? "sim" : "não")
```

```
Está ordenado? sim
```

Segunda solução

Os dados de entrada são inseridos em sua forma matricial.

In [57]:

```
# criação da ordenação através da composição da função intercalação de cada empresa via
heap max e a aplicação da intercalação de k vetores via heap min.
ordenacao_heap_max_min(
    entrada_matricial::Array{Float64,3},
    paralelizar::Bool,
)::Vector{Tuple{Int64,Int64,Int64,Float64}} =
    Algoritmos_Ordenacao.ordenacao_heap_2D(entrada_matricial, paralelizar) |>
    Algoritmos_Ordenacao.intercalar_k_vetores_ordenados
```

Out[57]:

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

In [60]:

```
#com paralelismo
@benchmark ordenacao_heap_max_min($entrada_matricial, $true)
```

Out[60]:

```
BenchmarkTools.Trial:
  memory estimate:  86.18 MiB
  allocs estimate:  25816
  --------------
  minimum time:     145.474 ms (0.00% GC)
  median time:      181.287 ms (0.00% GC)
  mean time:        211.556 ms (10.35% GC)
  maximum time:     618.778 ms (54.48% GC)
  --------------
  samples:          24
  evals/sample:     1
```

In [64]:

```
#sem paralelismo
@benchmark ordenacao_heap_max_min($entrada_matricial, $false)
```

Out[64]:

```
BenchmarkTools.Trial:
  memory estimate:  86.10 MiB
  allocs estimate:  25209
  --------------
  minimum time:     196.251 ms (0.00% GC)
  median time:      250.040 ms (0.00% GC)
  mean time:        254.343 ms (4.55% GC)
  maximum time:     317.983 ms (18.80% GC)
  --------------
  samples:          20
  evals/sample:     1
```

In [66]:

```
print("Está ordenado? ", ordenacao_heap_max_min(entrada_matricial, false) |> teste_orde
nacao ? "sim" : "não")
```

```
Está 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)

$\Theta(mn^2 \log(mn^2))$