**Universidade Federal de Ouro Preto Departamento de Computação**

Augusto Ferreira Guilarducci (20.1.4012)

Caio Monteiro (20.1.4110)

Paulo Correa (20.1.4036)

Pedro Lucas Damasceno (20.1.4003)

**Trabalho Prático II – Estrutura de Dados II**

Implementação e análise experimental de métodos de ordenação em memória secundária

**Ouro Preto, Minas Gerais**

**2021**

**SUMÁRIO**

1. **Introdução** .......................................................................................................... 3
2. ***Método de Seleção por Substituição***....................................................................?

a. Utilização do *Heapsort* como método de ordenação interna ….......................?

b. Geração dos blocos ordenados por meio do *Heapsort* ....................................?

c. A fase de Intercalação dos blocos ordenados……… ......................................?

d. Análise experimental ...................................................................................... ?

1. ***Quicksort* Externo** .............................................................................................. ?
2. Definição do TipoArea .................................................................................. ?
3. Inicialização dos apontadores de leitura e escrita .......................................... ?
4. Funções *InsereItem*, *RetiraUltimo* e *RetiraPrimeiro* ..................................... ?
5. Utilização do *insertion sort* como método de ordenação interna .................. ?
6. Análise experimental ..................................................................................... ?
7. **Conclusão** ............................................................................................................ ?

* **Introdução**

Este relatório se refere ao segundo trabalho prático da disciplina Estrutura de Dados 2, realizado durante o período letivo 21.1, cujo objetivo consiste na implementação e estudo da complexidade dos seguintes métodos de ordenação externa apresentados em sala de aula:

1. Intercalação balanceada de vários caminhos (*2f* fitas) utilizando, na etapa de geração dos blocos ordenados, o método de ordenação interna *selection sort* apresentado na disciplina “Estrutura de Dados I”.
2. Intercalação balanceada de vários caminhos (*2f* fitas) utilizando, na etapa de geração dos blocos ordenados, a técnica de seleção por substituição apresentada na disciplina “Estrutura de Dados II”.
3. Quicksort externo.

Dividimos tanto a implementação quanto as etapas de análise, sendo cada membro responsável por um método. A princípio, desenvolvemos a modularização e o fluxo de execução de modo que cada membro pudesse desenvolver sua parte separadamente e no seu tempo. Em seguida, implementamos a conversão do arquivo (*PROVAO.TXT*) para binário e vice-versa, dada a maior facilidade de implementação dos métodos de ordenação externa, e funções de uso geral (verificação dos parâmetros de execução, impressão de registros, métodos de ordenação interna) e outras diretivas do enunciado do trabalho.

Após diversas tentativas de subdividir a *string* contendo os dados não numéricos, optamos por não o fazer, haja vista que nenhuma etapa da ordenação externa (que considera exclusivamente um tipo *float*) e o enunciado do trabalho não requerem essa divisão. Além disso, vários erros subjetivos, que variavam de acordo com o sistema operacional dos membros, contribuíram para essa decisão. Após diversos testes, asseguramos que essa implementação não comprometeu sequer um caractere e produziu apenas resultados exatos. A implementação da estrutura de dados *Registro* em questão é:



Durante a análise experimental dos métodos, utilizamos a função *clock* da biblioteca *time.h* para calcular o tempo de execução e variáveis *int* globais para somar as comparações e leituras realizadas. As etapas de conversão dos arquivos foram descartadas e consideramos apenas a execução dos métodos de ordenação. Os *outputs* utilizados para a construção dos gráficos estão localizados no diretório ‘analise experimental’ do repositório https://github.com/augustofgui/grupo-21.1, que atualmente se encontra privado até a conclusão da disciplina para impossibilitar plágio.

* ***Método de Seleção por Substituição***

1. Utilização do *Heapsort* como método de ordenação interna

Um dos diferenciais desse método é o *Heapsort.* O *Heapsort* pega o vetor bloco de registros e os ordena em uma estrutura de árvore binária. A partir daí ele começa a ordenar os valores contidos no vetor. No nosso caso foi necessário usar o heap-mínimo, onde os nós-filhos possuem valores maiores que os nós-pais.

O *Heapsort* foi implementado com duas funções, a 'principal', que é chamada para começar a ordenação e a *peneira*, onde as comparações e as trocas entre um nó-pai e o nó filho são feitas.

O nosso *Heap* checa duas coisas durante a ordenação:

if((vet[j].nota < vet[j+1].nota || vet[j].f==1) && vet[j].f==0){

if((aux.nota < vet[j].nota || vet[j].f == 1) && aux.f == 0){

A primeira é o valor da nota do registro atual, que no primeiro caso é comparada com a próxima no vetor e no segundo caso é comparada com *aux* que possui um nó-pai, e a segunda coisa é a checagem da marca f, que checa se o item atual é o maior de todos (se *f* == 1) e, caso seja, para evitar a troca desse item.

1. Geração dos blocos ordenados por meio do *Heapsort*

Para o criação dos blocos, estabelecemos um registro, que representa a memória interna, para que nele fosse armazenado - através da leitura do arquivo - cada item contido no arquivo.

fread(mem\_interna, sizeof(Registro), TAM\_BLOCO, arquivo\_binario);

if (imprimir\_dados)

for (k = 0; k < TAM\_BLOCO; k++)

PrintFRead(&mem\_interna[k]);

Para a marcação de cada item - quando necessário - acrescentamos ao registro um ‘int f’, sua dinâmica é basicamente: se ‘f’ igual a 0, o item não está marcado, se for igual a 1, ele está marcado. Sua marcação seguiu a mesma lógica, se o próximo item é menor que o que está saindo, então ele é marcado, assim, sendo tratado como maior que todos os itens do bloco corrente.

if (mem\_interna[0].nota < anterior.nota) {

num\_reg\_marcados++;

mem\_interna[0].f = 1;

}

Quando todos os itens presentes no bloco são marcados, o bloco corrente é encerrado e um novo bloco ordenado é iniciado. E isso continua até que todos os registros tenham sido lidos e consequentemente todos os blocos ordenados.

while (num\_total\_reg\_lidos < num\_registros){

num\_reg\_marcados = 0;

num\_atual\_reg\_escritos = 0;

for (i = 0; i < TAM\_BLOCO; i++)

mem\_interna[i].f = 0;

while (num\_reg\_marcados < TAM\_BLOCO && num\_total\_reg\_lidos < num\_registros)

1. A fase de Intercalação dos blocos ordenados

Basicamente, substituímos o item do topo da fila de prioridades, escrevendo-o em uma fita de saída, pelo próximo item do mesmo bloco do item que está sendo substituído. E em seguida, a propriedade da fila foi reconstituída, o processo foi repetido até que não haja mais itens nos blocos ordenados.

A leitura do item no arquivo de origem e a escrita do item na fita de saída, respectivamente:

fread(&mem\_interna[i], sizeof(Registro), 1, origem[i].arquivo);

fwrite(&mem\_interna[pos\_menor], sizeof(Registro), 1, destino[j].arquivo);

A condição de parada verifica se ainda existe uma fita disponível e que todos os registros já tenham sido lidos, o loop segue até não possuir mais nenhuma.

while (num\_total\_leituras < nro\_registros) {

…

while (alguma\_fita\_disponivel(fitas\_disponiveis, NUM\_FITAS\_ENTRADA))

…

}

A função que verifica se existe alguma fita disponível - caso encontre algum valor dentro do vetor, ‘true’ é retornado, caso contrário, retorna ‘false’.

bool alguma\_fita\_disponivel(bool \*vetor, int tam)

{

int i;

for (i = 0; i < tam; i++)

if (vetor[i])

return true;

return false;

}

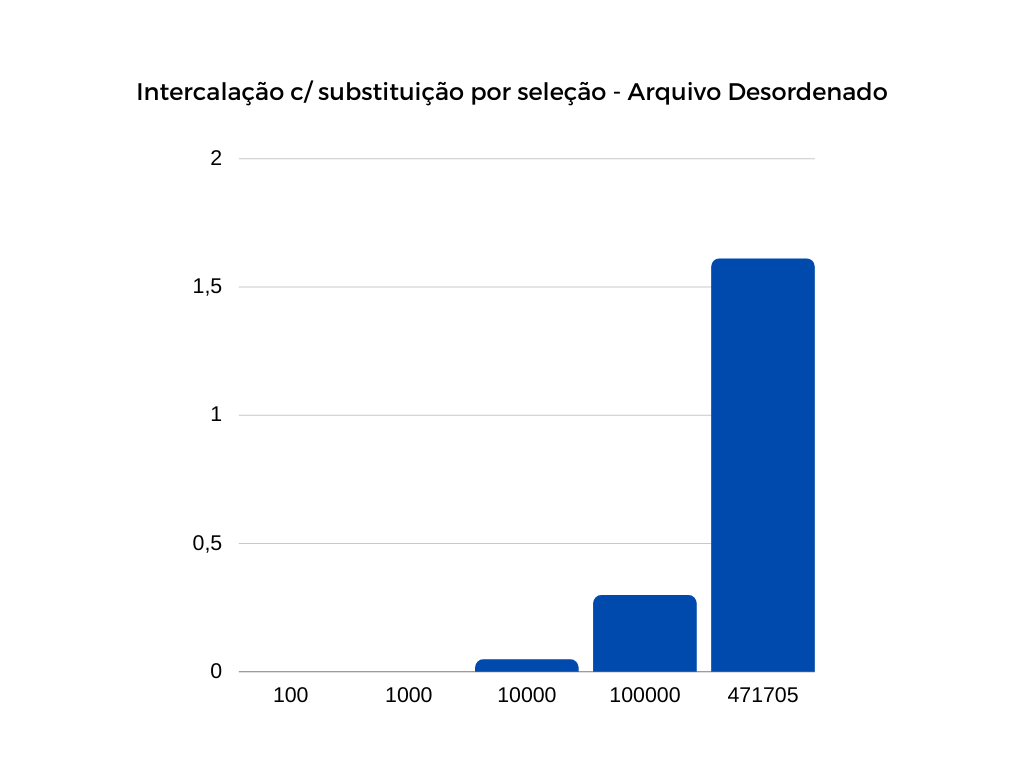
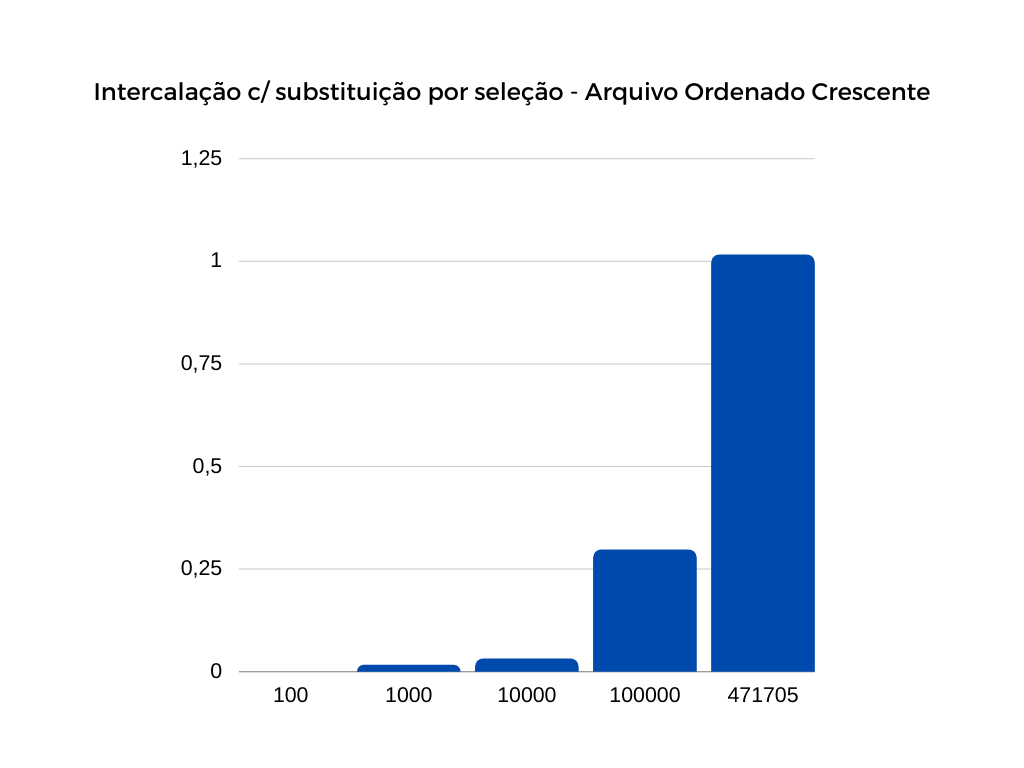
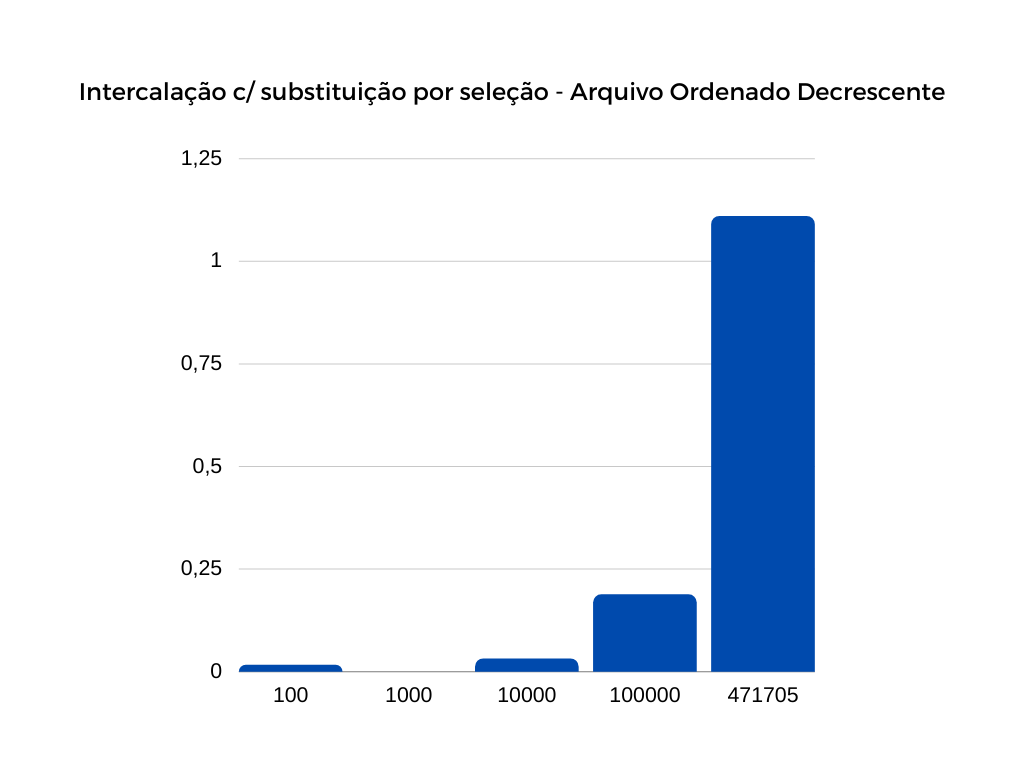
1. Análise experimental

Durante o processo de análise, observamos que todas as grandezas medidas (comparações, leituras, escritas e tempo de execução) crescem proporcionalmente à quantidade de registros a serem ordenados para qualquer situação de ordenação do arquivo original, como esperado.

Durante os testes utilizando o arquivo ordenado crescente e o decrescente, foi possível notar que a diferença entre o tempo de execução dos dois é bem pequena. Porém o tempo de execução foi constantemente menor no crescente, já que não foi preciso ordenar os blocos e apenas uma intercalação foi necessária, enquanto o decrescente precisa de duas ou mais.

Considerando os testes com o arquivo desordenado, foi possível perceber que o seu tempo de execução é constantemente o dobro do arquivo ordenado crescente, o que torna uma diferença considerável. Isso acontece devido ao elevado número de comparações, que também dobra em relação ao arquivo ordenado crescente, por conta do arquivo estar fora de ordem.

Abaixo, gráficos contendo dados referentes ao tempo de execução e número de registros ordenados. Outros gráficos considerando as grandezas de comparações, leituras e escritas estão presentes no diretório ‘analise experimental’.



* ***Quicksort* Externo**

1. Definição do *TipoArea*

Com a intenção de otimizar o método em geral, optamos por implementar uma estrutura de dados contendo o vetor de registros e um uma variável auxiliar para armazenar o número de índices ocupados. Dessa forma, para realizar a exclusão lógica quando e a ordenação do vetor, atribuímos um o valor de *INT\_MAX* à nota do índice removido e ordenamos apenas a quantidade necessária de elementos do vetor. Em um vetor com 5 posições ocupadas, por exemplo, ao remover o primeiro registro, ordenamos apenas as primeiras 5 posições, economizando comparações entre as outras 15 do vetor.



Para inicializar uma área vazia, atribuímos o mesmo valor de *INT\_MAX* a todas as posições do vetor e zero à variável *nro\_cels\_ocupadas*. Ademais, a variável *NRArea* presente no código de exemplo disponibilizado foi dispensada, haja vista que *nro\_cels\_ocupadas* cumpre exatamente a mesma função.

1. Inicialização dos apontadores de escrita e leitura e partição

Tal como no código de exemplo, utilizamos duas variáveis para a leitura e escrita inferiores e apenas uma para as superiores. Ambas são inicializadas sobre o arquivo binário convertido a partir do *.txt* com as permissões de abertura para leitura e escrita *r+b*.

As etapas de partição também seguem o código de exemplo disponibilizado, diferenciando apenas nos tipos de algumas variáveis como *Linf* e *Lsup* que, como as notas são do tipo float, não poderiam ser do tipo *int*; *OndeLer*, que teve o tipo alterado para booleano; e *NRArea*, que foi dispensada.

1. Funções *InsereItem, RetiraUltimo e RetiraPrimeiro*

Implementadas de acordo com a estrutura de dados *TipoArea*, realizam a inserção e remoção dos registros de forma ordenada de acordo com a quantidade de índices ocupados.

1. Utilização do *insertion sort* como método de ordenação interna

Em um primeiro momento, implementamos o método de ordenação *merge sort*, dada sua complexidade logarítmica. Todavia, após testes com a totalidade do arquivo de 471.705 registros, observamos uma diferença de aproximadamente 8 segundos em favor do *selection sort* que, apesar da complexidade quadrática, se provou mais eficiente.

Acreditamos que esse fato esteja relacionado à pequena dimensão do vetor de registros, que possui apenas 20 posições. Como o método de ordenação interna é referenciado várias vezes durante a execução do código, o *merge sort*, por precisar alocar e desalocar memória repetidas vezes, teve sua eficiência um pouco comprometida.

Ainda na busca pelo método de ordenação interna ideal, implementamos o método *insertion sort* dada sua eficiência para vetores quase ordenados, haja vista que a inserção e remoção efetua apenas a transferência de 1 elemento para dentro ou fora do vetor. Dentre os 3 testados, o *insertion sort* se provou mais eficiente.

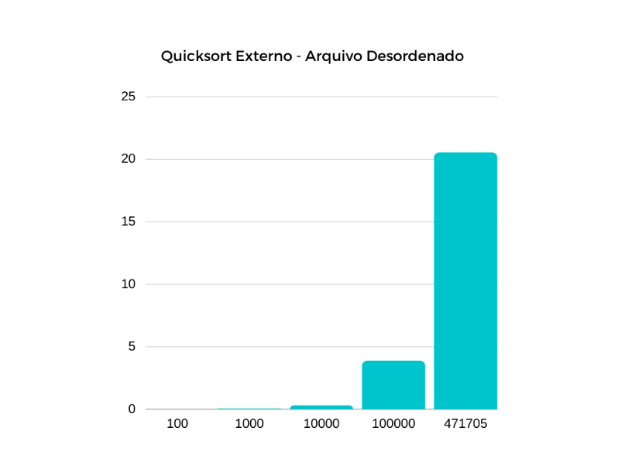
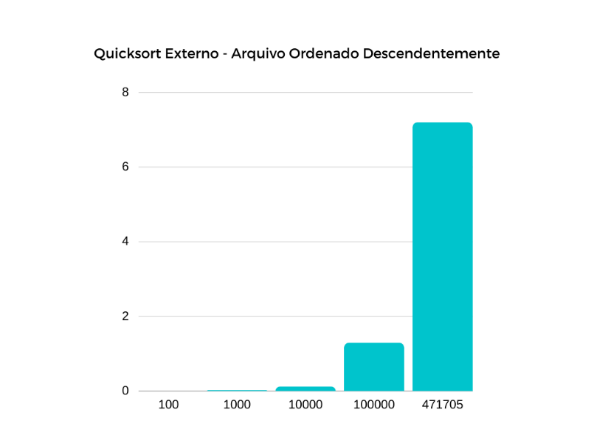
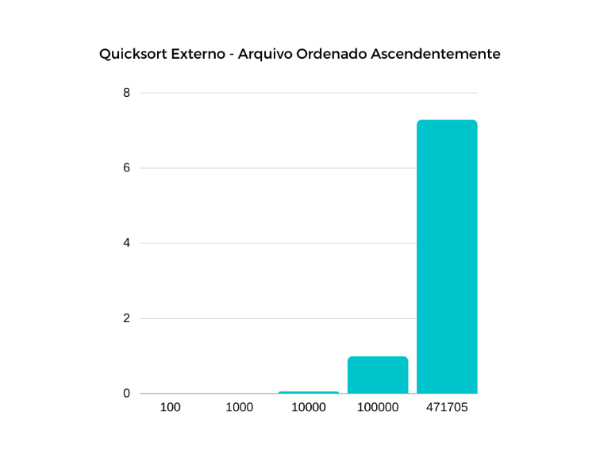
1. Análise experimental

Durante o processo de análise, observamos que todas as grandezas medidas (comparações, leituras, escritas e tempo de execução) crescem proporcionalmente à quantidade de registros a serem ordenados para qualquer situação de ordenação do arquivo original, como esperado.

Em relação às situações de ordenação, observamos que o arquivo desordenado realiza significativamente mais comparações, leituras e escritas, além de demorar mais de 3 vezes o tempo de execução para arquivos ordenados ascendentemente. Isso se deve ao fato da maior ocorrência de leitura de registros que não estejam entre os limites de *Linf* e *Lsup*, o que coincide com a menor quantidade de comparações realizadas pelo método de ordenação interna, e faz com que esses registros sejam imediatamente gravados nos subarquivos para serem referenciados novamente em uma chamada recursiva mais adiante. Já entre os arquivos previamente ordenados, observamos uma pequena diferença em favor do arquivo ordenado ascendentemente, como esperado do melhor caso do algoritmo.

O pior caso (partições geradas de tamanhos maior e menor possível) não foi identificado, visto que a grande quantidade de registros tende essa probabilidade de ocorrência a zero.

Abaixo, gráficos contendo dados referentes ao tempo de execução e número de registros ordenados. Outros gráficos considerando as grandezas de comparações, leituras e escritas estão presentes no diretório ‘analise experimental’.



* **Conclusão**

Após comparar os resultados obtidos nos diferentes ambientes à disposição, optamos por extrair as estatísticas a partir dos testes realizados no ambiente onde os tempos de execução foram menores. O ambiente em questão possui as seguintes especificações:

Processador: *Intel Core i5 8265U (1,6 GHz até 3,9 GHz) 6 MB Cache*

Memória: *8 GB de RAM DDR4 2400 MHz*

Dispositivo de memória secundária: *HD de 1 TB 5400 RPM*

Dentre todos os algoritmos implementados, elegemos o método de intercalação com substituição por seleção como o mais eficiente para ordenar o arquivo no contexto do trabalho prático, chegando a atingir a marca de 1.609375 segundos para ordenar todos os 471.705 registros do arquivo desordenado. O método de intercalação com ordenação interna se mostrou ligeiramente menos eficiente em comparação à substituição por seleção, demorando poucas frações de segundo a mais durante a execução.

O método de *quicksort* externo se mostrou o menos eficiente dentre os 3 implementados, demorando 20.515625 segundos para ordenar todo o arquivo desordenado. Todavia, vale ressaltar que, diferentemente dos métodos que utilizam a intercalação, o método de *quicksort* ordena *in situ*, o que pode ser uma característica decisiva a depender da circunstância na qual a ordenação externa é aplicada e que requer um processamento significativamente maior sobre o arquivo a ser ordenado.

* **Considerações Finais**

Durante os testes realizados com o *quicksort* externo, identificamos uma anomalia específica de um ambiente de testes. No caso, ao realizar a ordenação de apenas 100 registros, algumas frações do arquivo permaneciam desordenadas ao final da execução. Para outras quantidades de registros, o resultado obtido era o esperado (ordenado) e, após muito esforço para identificar a causa, concluímos que ela é particularidade do ambiente em questão e não está localizada na implementação. Ademais, como mencionado, a anomalia é específica de apenas um ambiente e não foi observada noutros 4 nos quais o mesmo código foi testado.