UAlg FCT
UNIVERSIDADE DO ALGARVE
FACULDADE DE CIÊNCIAS E TECNOLOGIA

Algoritmos e Estruturas de Dados

2021-2022

Projecto 2 – Ordenação

Introdução

No 2.º Projecto pretende-se explorar a criação e o estudo de vários algoritmos de ordenação.

Não é necessário ter um problema total implementado para obter a cotação correspondente no projecto. Mesmo que não tenha tempo para implementar algum dos métodos pedidos, deverá escrever o cabeçalho do mesmo, para que o *Mooshak* consiga compilar o código e executar os testes sobre os métodos que implementou.

Problema A: Implementação e estudo *JumpBubbleSort* (6 valores)

Um dos problemas do algoritmo Bubble Sort, é que no pior caso, para arrays muito grandes, um elemento que esteja na parte inferior do *array* necessita de muitas trocas até ser trazido para o topo[1]. Para tentar melhorar esta característica, foi proposta uma variante que iremos chamar de *JumpBubbleSort*. O funcionamento do *JumpBubbleSort* é muito semelhante ao do *BubbleSort* leccionado nas aulas, no entanto, em vez de o ciclo interno (*inner loop*) comparar elementos adjacentes, o *JumpBubbleSort* compara elementos que estão a uma distância de *d* posições, tal como ilustrado na figura seguinte.

1

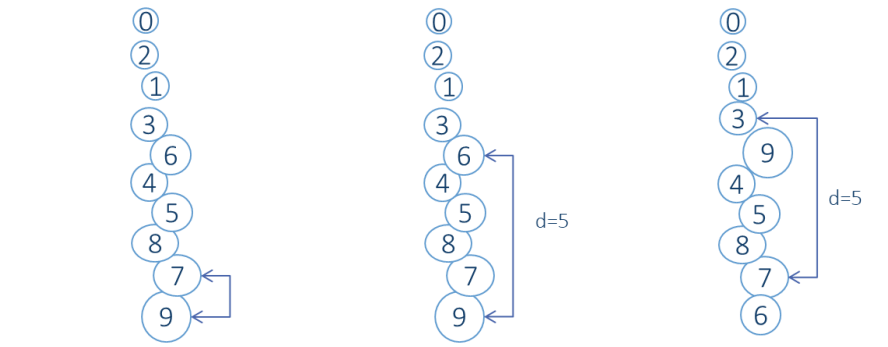


Figura 1. Lado esquerdo representa o Bubble Sort tradicional. Figuras do meio e da direita representam o JumpBubleSort para um salto j de tamanho 5: a 1.ª iteração do ciclo interno compara o elemento 9 com o 6, e a 2.ª iteração compara o 7 com o 3.

Inicialmente, o valor de *d* a usar para o salto é dado por *n*-1 (sendo n o número de elementos a ordenar). Ou seja, a 1.ª iteração apenas irá comparar o 1.º com o último elemento, e a cada iteração seguinte o *d* é diminuído.

[1] Este é um problema semelhante ao InsertionSort, que levou ao aparecimento do Shellsort. Ou seja, aqui o que pretende é implementar uma ideia parecida ao que o shellsort fez sobre o insertionsort.

Após vários testes empíricos, os criadores do algoritmo determinaram que a forma mais eficiente de atualizar a distância d é determinada pela seguinte fórmula:

$$d = \lfloor d \times 0.77 \rfloor$$

Obviamente a distância d deverá ser sempre um número inteiro, por isso deverá ser arredondada para baixo. No entanto, o valor de d nunca poderá ser inferior a 1.

Ao contrário do BubbleSort original a nova versão não vai diminuindo o nível, e o algoritmo *JumpBubbleSort* só termina quando fizer um ciclo interno com uma distância d = 1, e nessa iteração não houver trocas.

Implemente a classe *JumpBubbleSort*, que implementa o algoritmo descrito para ordenar um *array* recebido. Será fornecido um *template* para a classe *JumpBubbleSort* e para a classe mãe *Sort*. Poderá implementar outros métodos privados auxiliares caso assim o entenda, mas deverá implementar obrigatoriamente os 2 seguintes métodos estáticos:

JumpBubbleSort – Implementa métodos de ordenação baseados no algoritmo JumpBubbleSort		
`Void`	`sort(T[] a)`	Recebe um *array* de elementos comparáveis T, e ordena-o do menor para o maior.
`Void`	`main(String[] args)`	Método main que deverá ser usado para testar o método acima.

Objectivos/Requisitos Técnicos

Método *sort*

Este método deverá corresponder ao algoritmo *JumpBubbleSort* descrito acima. Deverá implementar o algoritmo da forma mais eficiente possível (de modo a passar os testes de eficiência).

Uma vez implementado o método *sort* utilize testes empíricos para comparar a eficiência da ordenação para várias situações:

- *array* já ordenado;
- *array* ordenado pela ordem inversa;
- *array* parcialmente ordenado[2];
- *array* ordenado de forma aleatória.

Recomenda-se múltiplos testes para *arrays* de tamanho elevado (ex: 100 000 elementos, e repita o teste 30 a 50 vezes obtendo uma média do tempo de execução). Indique o tempo de execução médio para os vários testes, e indique também - usando testes de razão dobrada - qual a ordem de crescimento assimptótica do algoritmo para as várias situações distintas. **Escreva um sumário dos testes e resultados, e uma breve descrição como comentários no ficheiro Java a submeter.**

Método main

Este método deverá conter alguns exemplos de teste que foram usados para testar o correto funcionamento do algoritmo de ordenação. Adicionalmente, deverá também implementar testes e os ensaios de razão dobrada utilizados para testar os métodos acima. Embora este

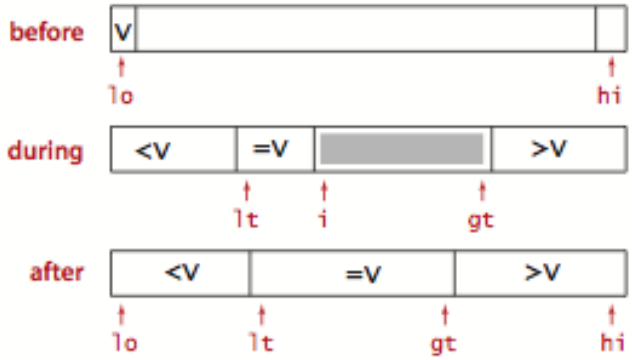
[2] Um array parcialmente ordenado pode ser obtido facilmente gerando um array ordenado, e trocando de forma aleatória um subconjunto (ex: n/8) dos elementos do array.

método não seja validado de forma automática pelo Mooshak será tido em consideração na validação do projecto, e contará para a nota final do mesmo.

Implemente a classe *JumpBubbleSort* no ficheiro *JumbleBubbleSort.java* fornecido. Submeta **apenas o ficheiro JumpBubbleSort.java** no Problema A.

Problema B: Implementação e estudo *QuickSort Tripartido* (7 valores)

O algoritmo *quicksort* tripartido tem como objetivo melhorar a performance do *quicksort* tradicional quando existem vários elementos repetidos no array a ordenar. Para isso, o seu método *partition*, em vez de tentar partir o *array* em 2, vai tentar partir o *array* em 3 partes (dai o nome tripartido): a parte esquerda - onde todos os elementos são menores que o pivô, a parte direita - onde todos os elementos são maiores que o pivô, e a parte do meio – onde todos os elementos são iguais ao pivô.



3-way partitioning overview

Poderá implementar outros métodos privados auxiliares caso assim o entenda, mas deverá implementar obrigatoriamente os 2 seguintes métodos estáticos:

QuickSort3Way – Implementa métodos de ordenação baseados no algoritmo *Quicksort* tripartido		
`Void`	`sort(T[] a)`	Recebe um *array* de elementos comparáveis T, e ordena-o do menor para o maior.
`Void`	`main(String[] args)`	Método main que deverá ser usado para testar o método acima.

Objectivos/Requisitos Técnicos

Método *sort*

Este método deverá corresponder ao algoritmo *QuickSort* tripartido, implementado da forma descrita nas aulas teóricas. Deverá implementar o algoritmo da forma mais eficiente possível (de modo a passar os testes de eficiência). O seu algoritmo deverá também ser eficiente na situação do pior caso possível (o *array* já se encontra ordenado).

Uma vez implementado o método *sort* utilize testes empíricos para comparar a eficiência da ordenação para várias situações:

- *array* já ordenado;
- *array* ordenado pela ordem inversa;
- *array* parcialmente ordenado;

- *array* ordenado de forma aleatória;
- *array com poucos elementos repetidos;*
- *array com muitos elementos repetidos;*

Recomenda-se múltiplos testes para *arrays* de tamanho elevado (ex: 100 000 elementos, e repita o teste 30 a 50 vezes obtendo uma média do tempo de execução). Indique o tempo de execução médio para os vários testes, e indique também - usando testes de razão dobrada - qual a ordem de crescimento assimptótica do algoritmo para as várias situações distintas. **Escreva um sumário dos testes e resultados, e uma breve descrição como comentários no ficheiro Java a submeter.**

Método main

Este método deverá conter alguns exemplos de teste que foram usados para testar o correto funcionamento do algoritmo de ordenação. Adicionalmente, deverá também implementar testes e os ensaios de razão dobrada utilizados para testar os métodos acima. Embora este método não seja validado de forma automática pelo Mooshak será tido em consideração na validação do projecto, e contará para a nota final do mesmo.

Implemente a classe *QuickSort3Way* no ficheiro *QuickSort3Way.java* fornecido. Submeta **apenas o ficheiro *QuickSort3Way*.java** no Problema B.

Problema C: Implementação de Filas Prioritárias através de *UnrolledLinkedList* (7 valores)

Implemente a classe *MinPriorityQueue*, que representa uma fila prioritária que processa os seus elementos do menor para o maior. A implementação desta fila deverá usar a Lista Ligada Desenrolada desenvolvida no 1.º Projeto[3]. A implementação deverá seguir a seguinte especificação:

MinPriorityQueue<T extends Comparable<T>> – implementa uma fila prioritária genérica (em que os elementos podem ser de um Tipo T que implemente a interface *Comparable*. Esta fila está organizada de modo a selecionar em primeiro lugar os elementos de menor valor.		
	`MinPriorityQueue()`	Cria uma fila prioritária vazia.
`void`	`insert(T element)`	Insere um elemento, de forma ordenada, na fila.
`T`	`removeMin()`	Remove e retorna o próximo item da fila prioritária (corresponde ao mínimo). Retorna *null* caso a fila esteja vazia.
`T`	`peekMin()`	Retorna o próximo item a ser processado na fila prioritária (corresponde ao mínimo), mas não o remove. Retorna *null* caso a fila esteja vazia
`boolean`	`isEmpty()`	Retorna *true* se a fila estiver vazia e *false* caso contrário
`Int`	`size()`	Devolve o tamanho (número de elementos) da fila.
`MinPriorityQueue`	`shallowCopy()`	Retorna uma cópia superficial da fila prioritária.
`T[]`	`getElements()`	Retorna um array com todos os elementos da fila[4].

[3] Embora não seja a implementação mais eficiente de uma fila prioritária, é um excelente exercício pedagógico.

[4] Este método retorna um array com tamanho igual ao número de elementos da fila, com todos os elementos lá dentro, ordenados do menor para o maior. Implementem este método da forma mais simples possível (não precisa de ser eficiente), pois será usado apenas para testes no Mooshak.

`void`	`main(String[] args)`	Método main, que deverá ser usado para testar os métodos acima

Objectivos/Requisitos Técnicos

Esta fila prioritária deverá ser implementada obrigatoriamente usando a *UnrolledLinkedList* implementada no 1.º Projeto. Apenas poderão usar os métodos públicos definidos na especificação do 1.º projeto.

Método *MinPriorityQueue*()

Este método é usado para criar uma fila prioritária vazia. Para cumprir o requisito do projeto, deverão usar uma *UnrolledLinkedList*. No entanto, deverão escolher um blockSize que seja eficiente. Uma *MinPriorityQueue* vai ser uma *UnrolledLinkedList* em que todos os seus elementos estão ordenados do menor (índice 0) para o maior (índice size – 1)

Método *insert*

O método *insert* vai inserir um novo elemento na lista – que já se encontra ordenada – de modo a que a lista continue ordenada depois da inserção.

Para determinar qual o sítio correto onde inserir o novo elemento, deverá ser utilizado um algoritmo de pesquisa binária, aplicado à lista.

Um algoritmo de pesquisa binária serve para pesquisar por um elemento num *array* ou lista ordenada (ou determinar onde inserir), e funciona da seguinte forma:

Dá jeito termos 2 índices, por exemplo *low* e *high*, que marcam respetivamente o início da lista, e o fim da lista.

1. Se temos 0 elementos entre *low* e *high*, retornamos *low*, pois ainda não existem elementos e caso queiramos inserir, devemos fazê-lo no início.
2. Calculamos a posição que está no meio[5] da lista (no meio entre *low* e *high*)
3. Analisar a posição que está no meio da lista
4. Se o elemento que estamos à procura for menor que o elemento que está no meio, então quer dizer que o elemento ou sítio que estamos à procura está do lado esquerdo. Vamos ter que continuar à procura do elemento do lado esquerdo mas podemos ignorar o lado direito da lista. Atualizamos o limite superior da lista, e repetimos o processo a partir de 1.
5. Se o elemento que estamos à procura for maior que o elemento que está no meio, então quer dizer que o elemento ou sítio que estamos à procura está do lado direito. Vamos ter que continuar à procura do elemento do lado direito mas podemos ignorar o lado esquerdo da lista. Atualizamos o limite inferior da lista, e repetimos o processo a partir de 1.
6. Se as duas condições anteriores falharem, quer dizer que o elemento que está no meio é igual ao elemento que estamos à procura, e devemos retornar a posição do meio encontrada.

Uma vez implementados os métodos *insert* e *removeMin* utilize testes empíricos para determinar a eficiência destes métodos. Recomenda-se múltiplos testes para *listas* de tamanho elevado (ex: 10 000 elementos, e repita o teste 30 a 50 vezes obtendo uma média do tempo de

[5] Caso o número de elementos seja par, existem 2 posições no meio, usamos a mais à esquerda (índice menor).

execução). Indique o tempo de execução médio para os vários testes, e indique também - usando testes de razão dobrada - qual a ordem de crescimento assimptótica do algoritmo para as várias situações distintas. **Escreva um sumário dos testes e resultados, e uma breve descrição como comentários no ficheiro Java a submeter.**

Método *getElements()*

O método *getElements* deve retornar um único *array* de elementos do Tipo T, com o tamanho igual ao número de elementos na lista, e com os items ordenados. O índice 0 do *array* deve corresponder ao menor elemento, e o índice maior deve corresponder ao maior elemento.

Método main

Para além de testes à funcionalidade dos vários métodos implementados, este método deverá implementar ensaios de razão dobrada utilizados para determinar de forma empírica a ordem de crescimento dos métodos: *insert* e *removeMin*. Escreva como comentários no código os valores obtidos para a razão dobrada r para ambos os métodos, e indique qual a ordem de complexidade. Embora este método não seja validado de forma automática pelo Mooshak será tido em consideração na validação do projeto, e contará para a nota final do mesmo.

Implemente a classe *MinPriorityQueue* no ficheiro MinPriorityQueue.java fornecido. Submeta **apenas o ficheiro MinPriorityQueue.java** no Problema C.

Condições de realização

O projecto deve ser realizado individualmente. Projectos iguais, ou muito semelhantes, originarão a reprovação na disciplina. O corpo docente da disciplina será o único juiz do que se considera ou não copiar num projecto.

O código do projecto deverá ser entregue obrigatoriamente por via electrónica, através do sistema Mooshak, **até às 23:59 do dia 26 de Novembro.** As validações terão lugar na 1.ª semana letiva de Janeiro. Os alunos terão de validar o código juntamente com o docente **durante** o horário de laboratório correspondente ao turno em que estão inscritos. **A avaliação e correspondente nota do projecto só terá efeito após a validação do código pelo docente.**

A avaliação da execução do código é feita automaticamente através do sistema Mooshak, a partir do início da próxima semana, usando vários testes configurados no sistema. O tempo de execução de cada teste está limitado, bem como a memória utilizada. Não é necessário o registo para quem já se registou no 1.º projecto, podendo usar o mesmo username e password. Para quem ainda não se tenha registrado, poderá fazê-lo usando o seguinte link:

http://deei-mooshak.ualg.pt/~dshak/cgi-bin/getpass-aed21

Deverão introduzir o vosso número de aluno e submeter. Irá ser gerada uma password que vos será enviada por email. Caso não recebam a password, verifiquem a vossa caixa de spam, e entrem em contacto com o corpo docente.

Uma vez criado o registo poderão fazer login no sistema Mooshak com o vosso número de aluno e a password recebida. O link para o sistema Mooshak é o seguinte:

http://deei-mooshak.ualg.pt/~dshak/