UNIVERSIDADE DE ÉVORA

Departamento de Informática

licenciatura em Engenharia Informática

- ALGORITMOS DE SORTEAMENTO -

(Cadeira de Estructuras De Dados e Algoritmos I)

2ºAno

Daniel Gonçalo Jesus Ramos, 29423

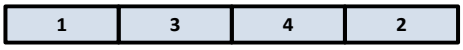
Marcus Vinicius Coelho Santos, 29764

Introdução

Para este trabalho prático, pretende-se implementar e analisar a performance de vários algoritmos de sorteamento para uma array gerada aleatoriamente de tamanho variável. Cada algoritmo será constituído por uma tabela onde serão visualizados os tempos mínimo, máximo e médio para uma array de tamanho N.

Cada teste feito aos algoritmos serão posteriormente gravados num ficheiro de acordo com o tamanho da array e o tempo em nanosegundos.

Para manter alguma consistência na explicação de cada algoritmo, será usada a seguinte array abaixo mencionada com um tamanho reduzido.



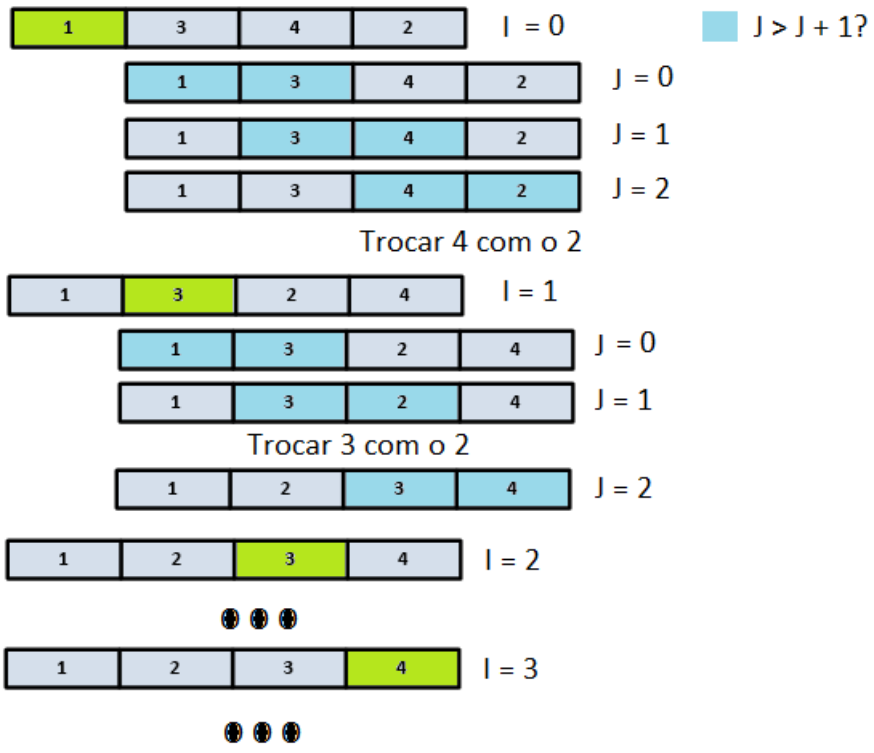
Algoritmos de sorteamento implementados

1. **Bubblesort**

   Este algoritmo permite sortear uma array por comparação de valores ao percorrer a array diversas vezes, movendo os valores maiores para o fim da array.

   Método:

   - Percorrer cada índice i da array
   - Para cada índice i da array percorrer cada índice j da array
   - Se o valor do índice J for maior que o valor do índice J + 1, trocar valores.



Quadro de performance:

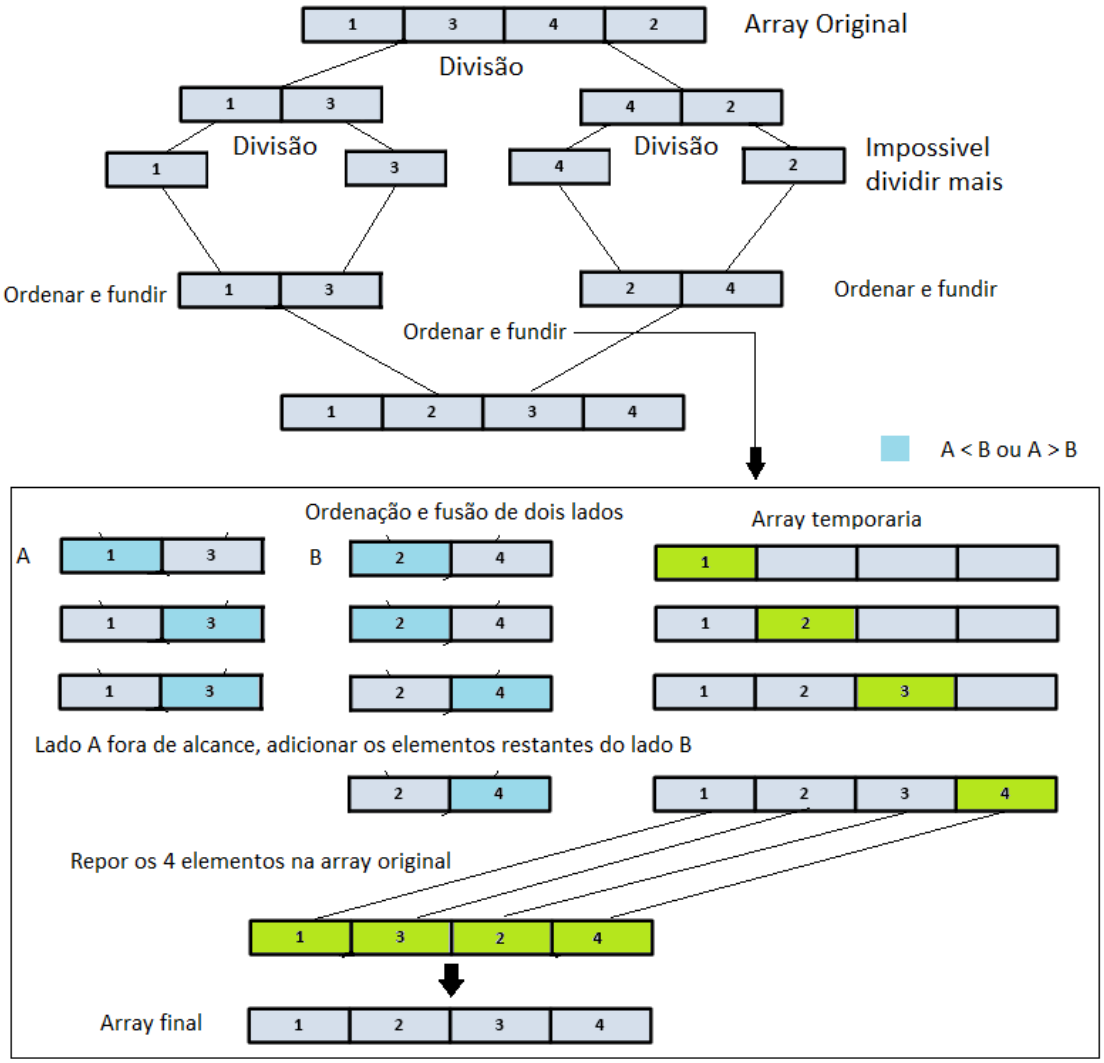
Bubblesort			
N	**T. Máximo**	**T. Minimo**	**T. Médio**
10	**4665**	**3265**	**3324**
100	**341954**	**21459**	**57381**
1000	**1942091**	**1869315**	**1884943**
10000	**231905619**	**226514136**	**229589093**

2. Mergesort

O algoritmo mergesort é um algoritmo recursivo de divisão e conquista, sendo o seu objectivo "partir" uma array em arrays mais pequenas dividindo a array e as suas sub arrays em dois, fundindo depois cada lado depois de ordenado.

Método:

- Dividir a array em 2 partes até não ser possível mais a sua divisão, isto é, só resta um elemento.
- Fundir a parte direita e esquerda depois de uma divisão em duas partes.
- Se um elemento do lado direito for menor que o do lado esquerdo, inserir esse valor na array temporária.
- Se um elemento do lado direito for maior que o do lado esquerdo, inserir o valor do lado esquerdo na array temporária.
- Adicionar os elementos restantes do lado A que não foram sorteados a array temporária.
- Adicionar os elementos restantes do lado B que não foram sorteados a array temporária.
- Adicionar os elementos da array temporária de volta a array original nas suas posições correctas.
- A divisão em lados A e B e a respectiva fusão é feita recursivamente enquanto for possível.



Quadro de performance:

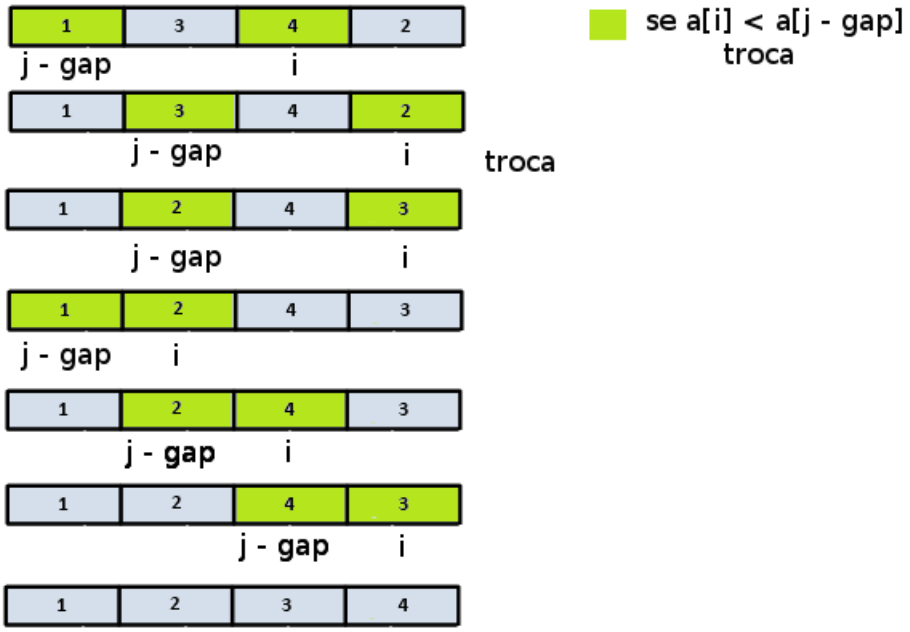
Mergesort			
N	**T. Máximo**	**T. Minimo**	**T. Médio**
10	**8397**	**4665**	**5015**
100	**66244**	**13062**	**33064**
1000	**108231**	**91903**	**98200**
10000	**2523832**	**1162548**	**1208616**

3. Shellsort

O algoritmo ShellSort é um algoritmo recursivo que ordena o Array a partir de um intervalo n, que inicialmente é metade do tamanho do Array. Por cada iteração esse intervalo é dividido por dois até que o mesmo seja um.

Método:

- Inicializa o intervalo de comparação para metade do tamanho do array.
- Faz uma iteração sobre Array e compara os elementos do mesmo com um intervalo gap.
- Por cada iteração divide-se o intervalo n por dois até que o mesmo seja 1.
- Compara as posições i e j - gap
- Caso o Array na posição i seja maior do que o Array na posição i + n, trocam-se os elementos e volta-se a compara as posições anteriores do Array enquanto houverem trocas, caso contrario continua-se a iteração normalmente.



Quadro de performance:

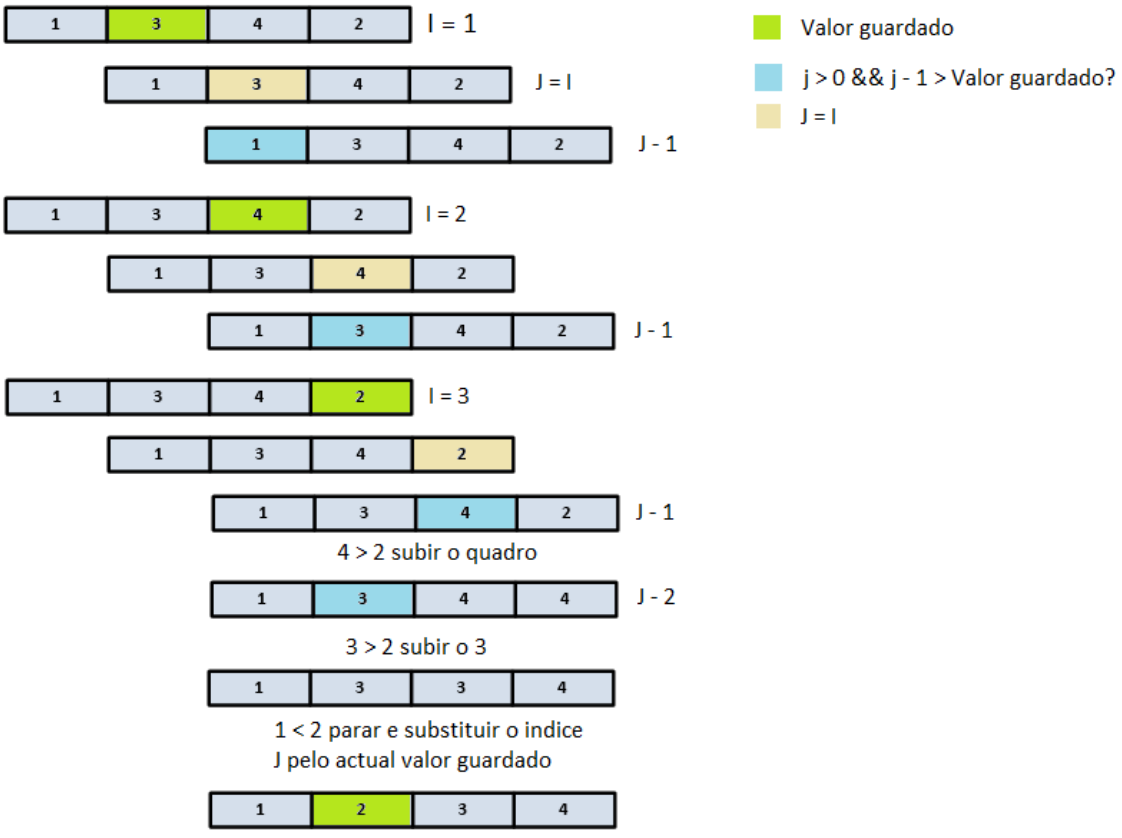
Shellsort			
N	**T. Máximo**	**T. Minimo**	**T. Médio**
10	**4665**	**2333**	**2740**
100	**145086**	**13529**	**15861**
1000	**1116831**	**1003468**	**1062598**
10000	**121522267**	**116376170**	**118817545**

4. Insertionsort

O algoritmo Insertionsort é um algoritmo de comparação em que os valores são ordenados conforme a ordem da sua inserção.

Método:

- Percorrer cada índice i da array
- Guardar o valor actual do índice i
- Percorrer cada índice j da array inversamente sendo o valor inicial igual ao índice i e apenas percorrer enquanto ouver valores maiores que o valor guardado.
- Atribuir o valor guardado no índice em que j parou, o que significa que já não há valores maiores que o valor guardado.
- Executar esta rotina até chegar ao fim da array.



Quadro de performance:

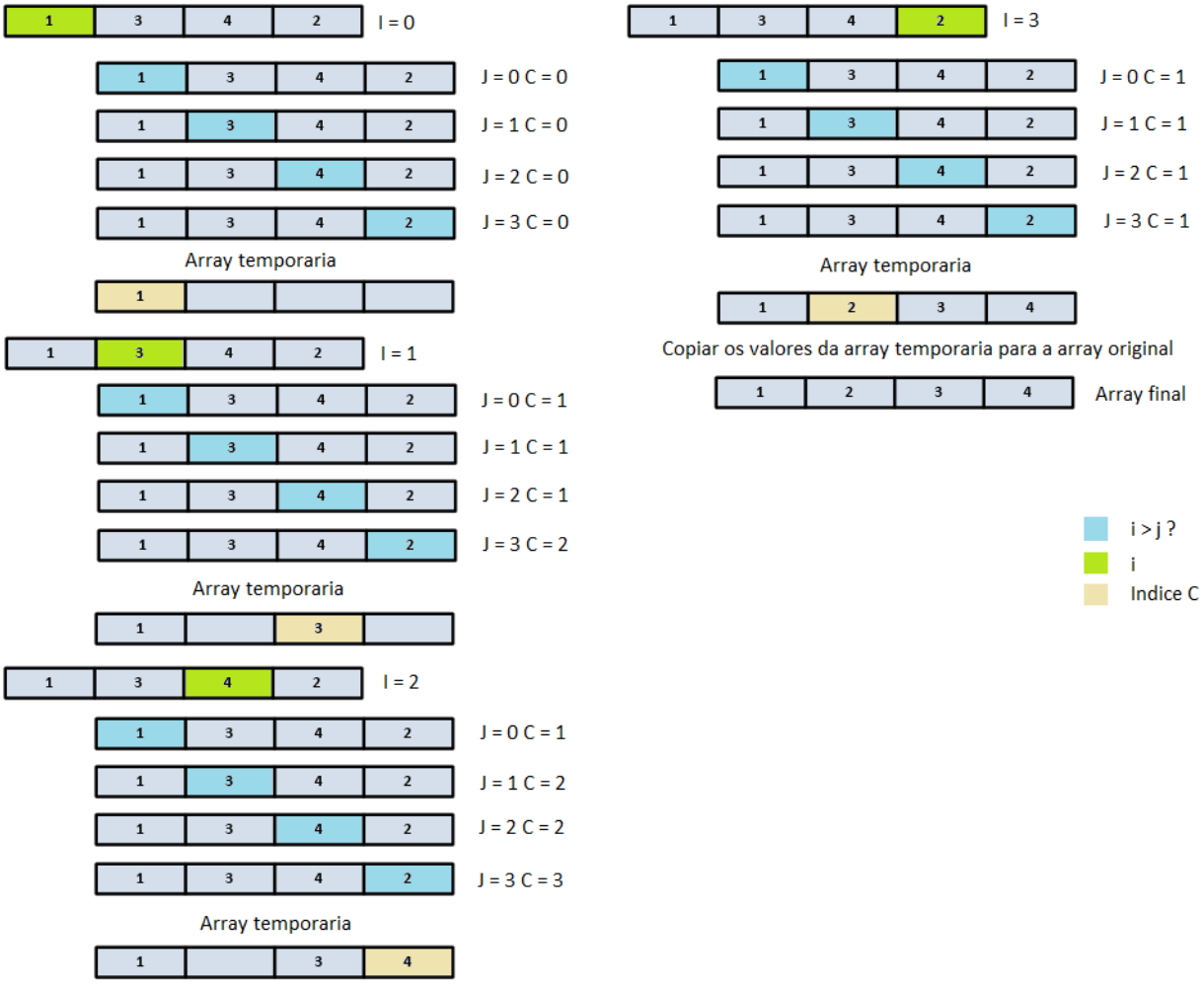
Insertionsort			
N	**T. Máximo**	**T. Minimo**	**T. Médio**
10	**1400**	**933**	**1224**
100	**49917**	**3732**	**21284**
1000	**363413**	**302300**	**326441**
10000	**32761762**	**31893117**	**32349365**

5. Ranksort

O algoritmo Ranksort é um algoritmo de comparação em que os valores são sorteados baseados no seu rank, isto é, no seu valor/peso numérico. Os valores com maior rank serão inseridos primeiro na sua posição correspondente, sendo a sua posição o índice calculado.

Método:

- Percorrer cada índice i da array
- Atribuir o valor de zero a variável que conta o índice de destino do valor de i
- Percorrer cada índice j da array
- Se o valor no índice i for maior que o valor no índice j, incrementar o valor do contador em 1
- Depois de percorrido cada índice j da array, atribuir o valor no índice i ao índice do contador na array temporária.
- Repetir até a array ser toda percorrida.



Quadro de performance:

Ranksort			
N	**T. Máximo**	**T. Minimo**	**T. Médio**
10	**4665**	**3265**	**3673**
100	**169811**	**36388**	**71493**
1000	**4410874**	**3950426**	**4176451**
10000	**412785984**	**400519975**	**407642335**

6. Radixsort

O algoritmo radixsort é um algoritmo de sorteamento baseado no LSD (less significant digit), logo a implementação feita neste trabalho pratico é o algoritmo LSD Radixsort.

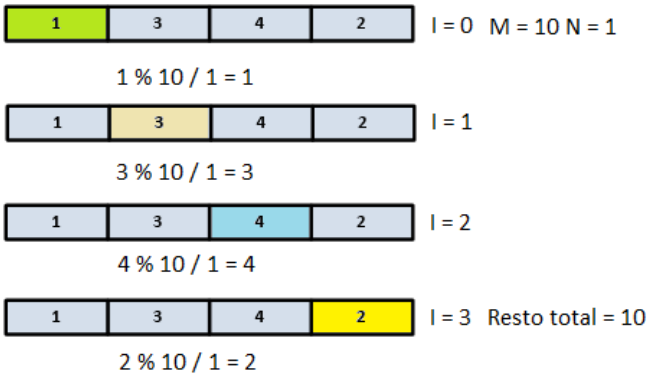
Este algoritmo começa pelo dígito menos significativo até ao mais significativo. Cada digito corresponde a uma chave que será inserida numa lista de chaves sendo que as chaves mais curtas vem antes das chaves mais longas (por exemplo 0 vem antes de 9).

Este algoritmo é feito recursivamente e a ordenação é feita ao adicionar os números na respectiva lista baseada na chave calculada. Este cálculo será feito até não haver mais dígitos para extrair.

Como o algoritmo na sua forma original não suporta números negativos, tivemos o cuidado de adicionar o seu suporte.

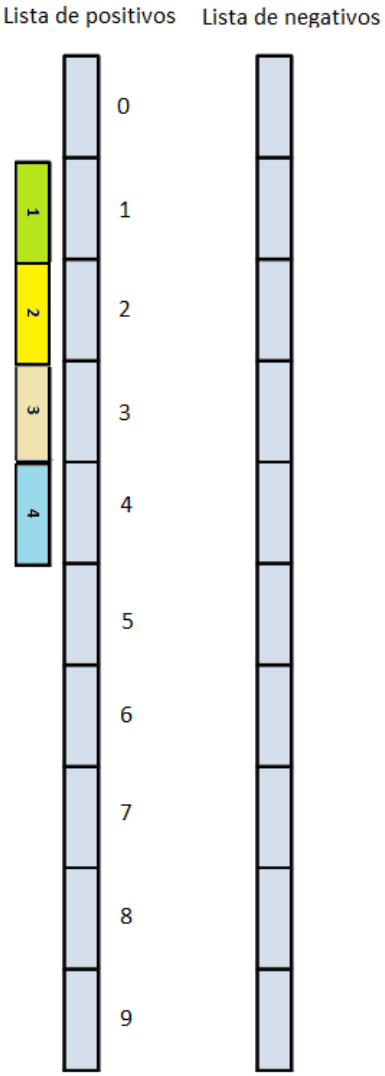
Método:

- Percorrer cada índice i da array até ao fim
- Obter o resto baseado em m e n em que começam em 10 e 1 respectivamente e são multiplicados por 10 cada vez que é chamado o método recursivamente para seleccionar o digito pretendido de um numero incrementalmente.
- Se o valor no índice i for negativo, converter o resto para positivo, incrementar o resto total e adicionar o valor na lista dos números negativos, utilizando o resto como chave/índice.
- Se for valor no índice i for positivo, incrementar o resto total e adicionar o valor na lista dos números positivos, utilizando o resto como chave/índice.
- Adicionar os valores negativos inversamente na array original, percorrendo todas as chaves da lista.
- Adicionar os valores positivos na array original, percorrendo todas as chaves da lista.
- Limpar os dados das listas.
- Chamar o método recursivamente caso o resto total seja maior que zero e multiplicar m e n por 10.



Adicionar os valores das listas ao array original

Não existem mais digitos para verificar se não tinha-se que multiplicar M * 10 e N * 10 e repetir o processo. Cada chave na lista pode conter vários números e não apenas um como exemplificado neste exemplo.



Quadro de performance:

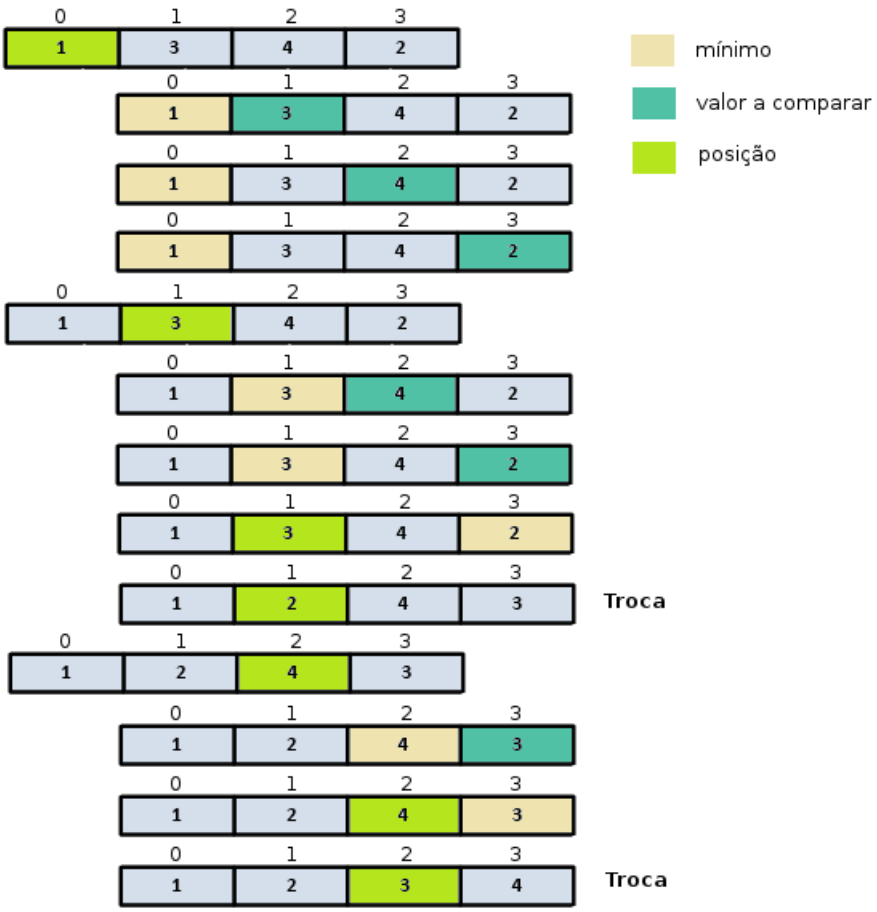
Radixsort			
N	**T. Máximo**	**T. Minimo**	**T. Médio**
10	**600868**	**61113**	**64612**
100	**315363**	**32189**	**121351**
1000	**295769**	**215063**	**250867**
10000	**4251327**	**2381079**	**2896983**

7. Selectionsort

O algoritmo SelectionSort é um algoritmo iterativo que ordena o Array baseado no index do valor mínimo.

Método:

- Para 0 <= i < array.length
- Inicializa o valor mínimo com a posição i
- Percorre o array à procura de um valor menor do que o valor mínimo.
- Quando encontrar um valor menor substitui o index do valor mínimo pelo index do menor valor.
- Troca com valor da posição index-mínimo com o valor da posição i e segue para a próxima iteração (i++)



Quadro de performance:

Selectionsort			
N	**T. Máximo**	**T. Minimo**	**T. Médio**
10	**1866**	**1399**	**1691**
100	**78841**	**10263**	**36563**
1000	**767413**	**710032**	**727875**
10000	**66508333**	**65395700**	**65811479**

8. Heapsort

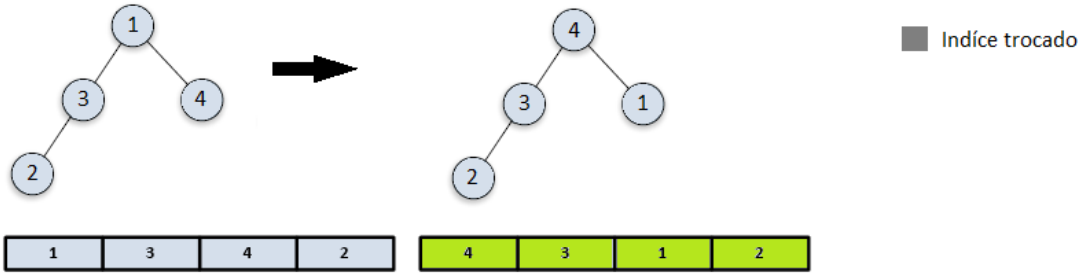
O algoritmo heapsort é um algoritmo em que usa uma estrutura de dados chamada heap. A heap pode ser representada como uma árvore binária em que o elemento menor fica na raiz.

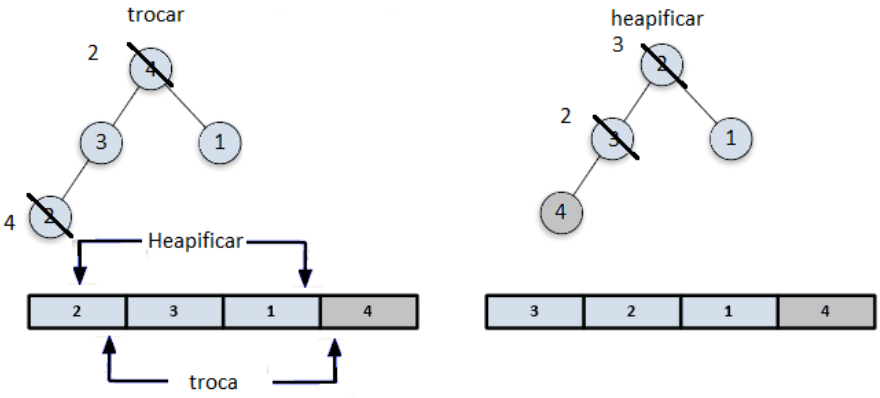
Método:

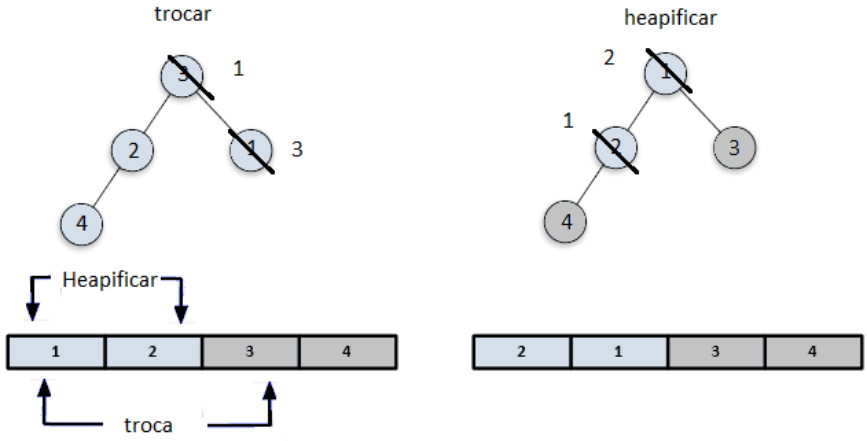
- Construir a heap.
- Percorrer a array do fim para o principio.
- Trocar o valor da raiz com o valor no índice i, mantendo assim os valores que já foram trocados inalteráveis.
- Heapificar a array depois de uma troca do índice 0 ate ao índice i-1(este serve para definir o índice máximo que é possível heapificar).
- Para heapificar, deve-se calcular o filho esquerdo a partir do índice do nó.
- Caso o índice do filho esquerdo seja menor que o tamanho do índice máximo que é possível heapificar, deve-se prosseguir da seguinte forma.

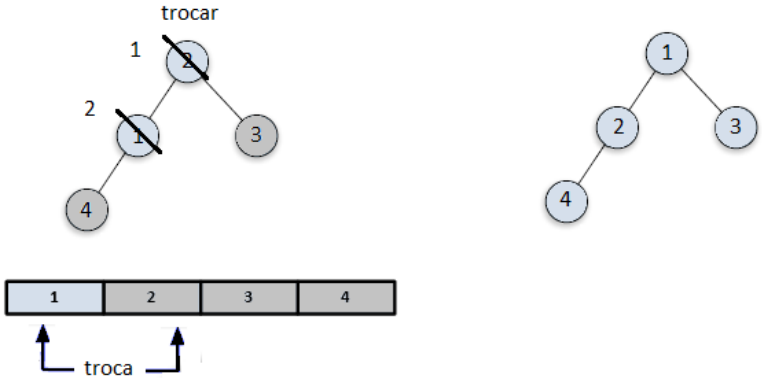
1. Verificar se o valor do filho esquerdo é menor que o valor do filho direito. Caso seja, incrementar o índice J em 1 para que se verifique o valor do filho direito.
2. Verificar se o valor do nó é menor que o valor do filho direito ou esquerdo conforme a verificação anterior. Caso seja, trocar o valor do nó com o valor do filho correcto e retornar a heapificar a array a partir do filho actual (J).
3. Proseguir até não haver mais filhos esquerdos.

Heapificar - Construção Inicial









Quadro de performance:

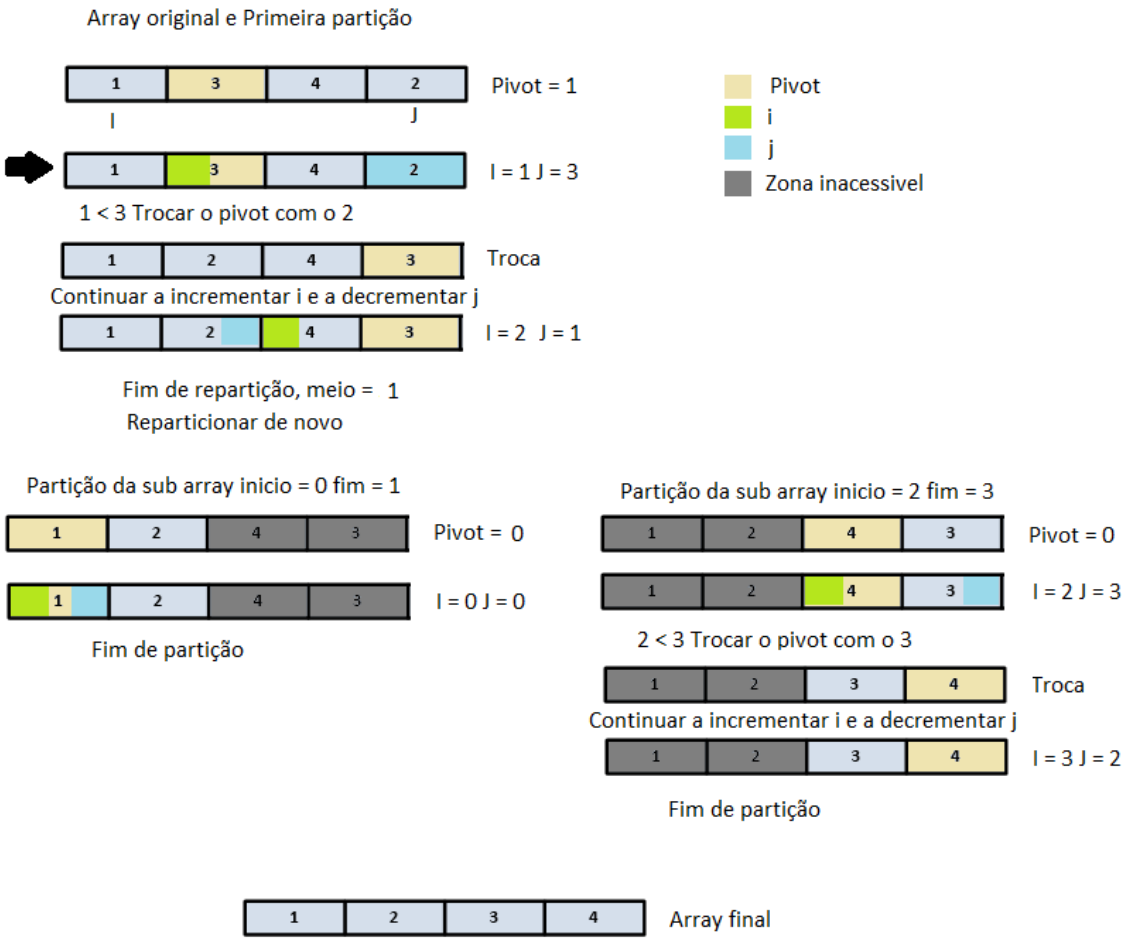
Heapsort			
N	**T. Máximo**	**T. Minimo**	**T. Médio**
10	**11196**	**5132**	**5598**
100	**62046**	**9797**	**26532**
1000	**149750**	**89570**	**102399**
10000	**1310900**	**1180276**	**1243896**

9. Quicksort

O algoritmo quicksort é um algoritmo recursivo de divisão e conquista, sendo o seu objectivo "dividir" uma array em arrays mais pequenas em que é escolhido um pivot onde os números do lado esquerdo que são maiores que o pivot passam para o lado direito e os números do lado direito que são menores que o pivot passam para o lado esquerdo.

Método:

- Enquanto o índice esquerdo for menor que o índice direito, particionar a array, obtendo o meio.
- Na partição, incrementar o índice i enquanto não encontrar um valor maior que o pivot. Decrementar o índice j enquanto não encontrar um valor menor que o pivot.
- Caso encontre, se o índice i e j não forem simultaneamente o do pivot. Trocar os valores que estão no índice i e j, incrementando i e decrementando j de seguida.
- Repetir o processo enquanto os índices i e j não coincidirem com o índice do pivot.
- Retornar o índice do pivot.
- Continuar recursivamente a particionar a array baseado no índice retornado para dividir a array em dois.



Quadro de performance:

Quicksort			
N	**T. Máximo**	**T. Minimo**	**T. Médio**
10	**5132**	**1866**	**2157**
100	**42920**	**9797**	**19360**
1000	**82106**	**62046**	**65136**
10000	**832258**	**768812**	**809457**

Estructura do programa de ordenação de algoritmos

Classes do programa

Classe FileOutput

Modifier and Type	Method and Description
void	**finish**()
void	**initiate**(java.lang.String headerName)
void	**writeResult**(int iteracao, java.lang.String result)
void	**writeResult**(long numIteracoes, long tempoMaior, long tempoMenor, long tempoMedio)
void	**writeResult**(java.lang.String result)

Classe RandomArray

Modifier and Type	Method and Description
int[]	**get**() Retorna um array de inteiros escolhidos aleatoriamente.

Class Sort

Modifier and Type	Method and Description
static long	**bubbleSort**(int[] a) Algoritmo BubbleSort
static long	**heapSort**(int[] a) Algpritmo heapsort
static long	**insertionSort**(int[] array) Algoritmo insertionsort
static long	**mergeSort**(int[] a) Algoritmo MergeSort.
static long	**quickSort**(int[] a) Algoritmo quicksort.
static long	**radixSort**(int[] a) Algoritmo RadixSort.
static long	**rankSort**(int[] a) Algoritmo RankSort.
static long	**selectionSort**(int[] array) Algoritmo selectionSort.
static long	**shellSort**(int[] a) Algoritmo ShellSort.

Class Testes

Modifier and Type	Method and Description
void	**resetTempos**()
void	**testarBubblesort**(**FileOutput** output, int tamanhoArray, int numIteracoes)
void	**testarHeapsort**(**FileOutput** output, int tamanhoArray, int numIteracoes)
void	**testarInsertionsort**(**FileOutput** output, int tamanhoArray, int numIteracoes)
void	**testarMergesort**(**FileOutput** output, int tamanhoArray, int numIteracoes)
void	**testarQuicksort**(**FileOutput** output, int tamanhoArray, int numIteracoes)
void	**testarRadixsort**(**FileOutput** output, int tamanhoArray, int numIteracoes)
void	**testarRanksort**(**FileOutput** output, int tamanhoArray, int numIteracoes)
void	**testarSelectionsort**(**FileOutput** output, int tamanhoArray, int numIteracoes)
void	**testarShellsort**(**FileOutput** output, int tamanhoArray, int numIteracoes)

Class Timer

Modifier and Type	Method and Description
static java.lang.String	**convertToMS**(long t)
void	**end**()
void	**start**()
long	**timeElapsed**()
java.lang.String	**toString**()

Class DoubleLinkedList

Modifier and Type	Method and Description
void	**add**(int index, **T** e) Adiciona um elemento a um indice especifico.
void	**add**(**T** element) Adiciona um elemento.
void	**clear**() Faz um reset a lista.
boolean	**contains**(**T** e) Verifica se o elemento existe na lista.
T	**get**(int index) Obtem o elemento de um indice especifico caso exista.
T	**getFirst**() Obtem o primeiro elemento se existir.
T	**getLast**() Obtem o ultimo elemento se existir.
DoubleNode<**T**>	**getTail**() Retorna a cauda da lista.
java.util.Iterator	**iterator**() Retorna o iterator da lista.
T	**remove**(int index) Remove o elemento que esta num determinado indice e retorna-o.
boolean	**remove**(**T** element) Remove um elemento se existir.
T	**removeLast**()
void	**removeNode**(**DoubleNode**<**T**> node) Remove um node, desligando-o do node anterior e do node seguinte.

void	**set**(int index, **T** e) Substitui o elemento num indice especifico caso este exista.
int	**size**() Retorna o tamanho actual da lista.
java.lang.String	**toString**()

Class DoubleLinkedListIterator

Modifier and Type	Method and Description
boolean	**hasNext**() Retorna verdadeiro de ouver um elemento seguinte.
T	**next**() Retorna o elemento actual.
void	**remove**() Remove o elemento actual.

Class DoubleNode

Modifier and Type	Method and Description
T	getElement() Retorna o elemento.
DoubleNode<T>	getNext() Retorna o node seguinte.
DoubleNode<T>	getPrev() Obtem o node anterior.
void	setElement(T element) Atribui o elemento.
void	setNext(DoubleNode<T> next) Atribui o node seguinte.
void	setPrev(DoubleNode<T> prev) Atribui o node anterior.

Class NoSuchElementException

Constructor and Description
NoSuchElementException()
NoSuchElementException(java.lang.String s)

Class IndexOutOfBoundsException

Constructor and Description
IndexOutOfBoundsException()
IndexOutOfBoundsException(java.lang.String s)

Interfaces do programa

Interface LinkedList

Modifier and Type	Method and Description
void	**add**(int index, **T** e)
void	**add**(**T** e)
void	**clear**()
T	**get**(int index)
T	**remove**(int index)
boolean	**remove**(**T** e)
void	**set**(int index, **T** e)
int	**size**()

Conclusão

Em síntese pode-se concluir através dos testes efectuados que para arrays muito pequenas o algoritmo insertionSort é o mais eficiente. Para arrays mediadas que rodem os 100 elementos, o algoritmo shellsort é o mais eficiente.

Para arrays grandes o algoritmo quicksort é o mais eficiente.

O algoritmo menos eficiente para arrays de dimensões 10 e 100 é o algoritmo radixsort.

O algoritmo menos eficiente para arrays grandes é o algoritmo ranksort.

Devido a demora do processamento dos algoritmos, não foi possível incluir os tempos relativamente a um N de 100000.

Bibliografia

Radixsort : https://www.youtube.com/watch?v=xhr26ia4k38

Ranksort: http://ww2.cs.mu.oz.au/498/notes/node32.html

Mergesort: Mark Weiss , Data Structures and Algorithm Analysis in Java, third edition.

Bubblesort: Baseado nos diapositivos apresentados nas aulas de EDA I.

Shellsort: Mark Weiss , Data Structures and Algorithm Analysis in Java, third edition.

Insertionsort: Mark Weiss , Data Structures and Algorithm Analysis in Java, third edition.

Selectionsort: https://www.youtube.com/watch?v=TW3_7cD9L1A

Quicksort: Baseado nos diapositivos apresentados nas aulas de EDA I.

Heapsort: Baseado nos diapositivos apresentados nas aulas de EDA I.