Instituto Politécnico de Beja

Estruturas de Dados e Algoritmos

Relatório

eda12131190311906

Docente: José Jasnau Caeiro

Abril de 2013

**Desenvolvido por:**

Tiago Conceição Nº11903

Gonçalo Lampreia Nº11906

Índice

[Introdução 5](#_Toc358297917)

[Sistema de classes 6](#_Toc358297918)

[Aplicação base 6](#_Toc358297919)

[Utilidades 7](#_Toc358297920)

[Sistema de relatórios 8](#_Toc358297921)

[Algoritmos 9](#_Toc358297922)

[Módulos 10](#_Toc358297923)

[Insertion-Sort 11](#_Toc358297924)

[Métodos e Funções 11](#_Toc358297925)

[Desempenho computacional teoricamente previsto de cada algoritmo 11](#_Toc358297926)

[Projeto da experiência que permite obter os resultados para posterior análise experimental 12](#_Toc358297927)

[Bubble-Sort 13](#_Toc358297928)

[Métodos e Funções 13](#_Toc358297929)

[Desempenho computacional teoricamente previsto de cada algoritmo 13](#_Toc358297930)

[Projeto da experiência que permite obter os resultados para posterior análise experimental 14](#_Toc358297931)

[Heap-Sort 15](#_Toc358297932)

[Métodos e Funções 15](#_Toc358297933)

[Desempenho computacional teoricamente previsto de cada algoritmo 16](#_Toc358297934)

[Projeto da experiência que permite obter os resultados para posterior análise experimental 16](#_Toc358297935)

[Merge-Sort 17](#_Toc358297936)

[Métodos e Funções 17](#_Toc358297937)

[Desempenho computacional teoricamente previsto de cada algoritmo 17](#_Toc358297938)

[Projeto da experiência que permite obter os resultados para posterior análise experimental 18](#_Toc358297939)

[QuickSort 19](#_Toc358297940)

[Métodos e Funções 19](#_Toc358297941)

[Desempenho computacional teoricamente previsto de cada algoritmo 19](#_Toc358297942)

[Projeto da experiência que permite obter os resultados para posterior análise experimental 20](#_Toc358297943)

[Radix-Sort 21](#_Toc358297944)

[Métodos e Funções 21](#_Toc358297945)

[Desempenho computacional teoricamente previsto de cada algoritmo 21](#_Toc358297946)

[Projeto da experiência que permite obter os resultados para posterior análise experimental 22](#_Toc358297947)

[Bucket-Sort 23](#_Toc358297948)

[Métodos e Funções 23](#_Toc358297949)

[Desempenho computacional teoricamente previsto de cada algoritmo 23](#_Toc358297950)

[Projeto da experiência que permite obter os resultados para posterior análise experimental 24](#_Toc358297951)

[Counting-Sort 25](#_Toc358297952)

[Métodos e Funções 25](#_Toc358297953)

[Desempenho computacional teoricamente previsto de cada algoritmo 25](#_Toc358297954)

[Projeto da experiência que permite obter os resultados para posterior análise experimental 26](#_Toc358297955)

[Comb-Sort 27](#_Toc358297956)

[Métodos e Funções 27](#_Toc358297957)

[Desempenho computacional teoricamente previsto de cada algoritmo 27](#_Toc358297958)

[Projeto da experiência que permite obter os resultados para posterior análise experimental 28](#_Toc358297959)

[Shell-Sort 29](#_Toc358297960)

[Métodos e Funções 29](#_Toc358297961)

[Desempenho computacional teoricamente previsto de cada algoritmo 29](#_Toc358297962)

[Projeto da experiência que permite obter os resultados para posterior análise experimental 30](#_Toc358297963)

[Selection-Sort 31](#_Toc358297964)

[Métodos e Funções 31](#_Toc358297965)

[Desempenho computacional teoricamente previsto de cada algoritmo 31](#_Toc358297966)

[Projeto da experiência que permite obter os resultados para posterior análise experimental 32](#_Toc358297967)

[Comparação de todos os resultados 33](#_Toc358297968)

[Conclusão 34](#_Toc358297969)

[Bibliografia 35](#_Toc358297970)

[Ilustração 1 6](#_Toc358298021)

[Ilustração 2 7](#_Toc358298022)

[Ilustração 3 8](#_Toc358298023)

[Ilustração 4 9](#_Toc358298024)

[Ilustração 5 10](#_Toc358298025)

[Ilustração 6 11](#_Toc358298026)

[Ilustração 7 12](#_Toc358298027)

[Ilustração 8 13](#_Toc358298028)

[Ilustração 9 14](#_Toc358298029)

[Ilustração 10 15](#_Toc358298030)

[Ilustração 11 16](#_Toc358298031)

[Ilustração 12 17](#_Toc358298032)

[Ilustração 13 19](#_Toc358298033)

[Ilustração 14 20](#_Toc358298034)

[Ilustração 15 21](#_Toc358298035)

[Ilustração 16 22](#_Toc358298036)

[Ilustração 17 23](#_Toc358298037)

[Ilustração 18 24](#_Toc358298038)

[Ilustração 19 25](#_Toc358298039)

[Ilustração 20 26](#_Toc358298040)

[Ilustração 21 27](#_Toc358298041)

[Ilustração 22 28](#_Toc358298042)

[Ilustração 23 29](#_Toc358298043)

[Ilustração 24 30](#_Toc358298044)

[Ilustração 25 31](#_Toc358298045)

[Ilustração 26 32](#_Toc358298046)

[Ilustração 27 33](#_Toc358298047)

# Introdução

Este relatório aborda vários conceitos relacionados à complexidade de algoritmo, tendo como objetivo apresentar uma análise experimental englobando aspetos de diversos algoritmos de ordenação, tais como:

* Insertion-sort
* Bubble-sort
* Heapsort
* Mergesort
* Quicksort
* Radix
* Bucket
* Counting
* Comb
* Shell
* Selection

Estes algoritmos de ordenação estão programados em linguagem C# seguindo o pseudo-código subjacente presente no livro da disciplina.

Em relação aos algoritmos iremos mostrar ao longo do relatório o estudo dos mesmos, comparando o desempenho experimental computacional com as previsões teóricas.

Onde existe uma possível função de complexidade: **pior caso, melhor caso e caso médio**. Para classificar a ordem das funções temos três classificações: Ordem O, Ômega e Theta, que serão analisados ao longo do relatório.

Iremos estudar o código ao nível da sua arquitetura, sistema de classes, variáveis escolhidas, métodos e funções, módulos, desempenho computacional teoricamente previsto de cada algoritmo, os resultados para posterior análise experimental, comparação dos dados previsto com os dados obtidos e por fim a análise dos mesmos.

# Sistema de classes

## Aplicação base

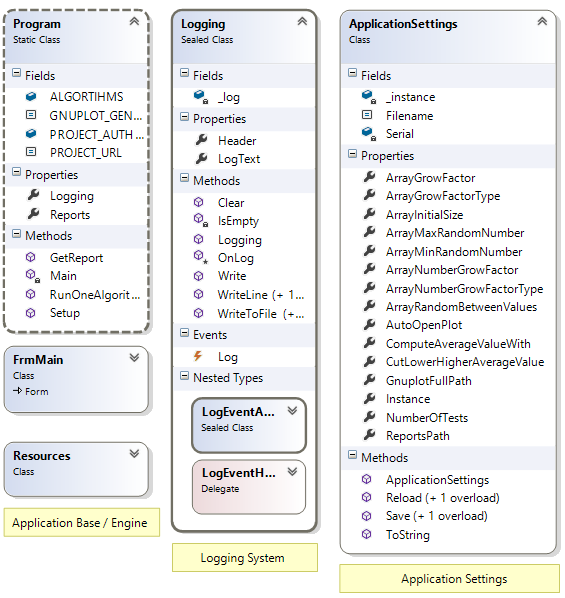


Ilustração 1

Classes principais do programa.

**Program** 🡪 Inicializador da aplicação, contém variáveis estáticas globais para acesso geral.

**FrmMain** 🡪 Codigo base da aplicação, onde se encontra o núcleo da aplicação e do ambiente gráfico da mesma.

**Resources** 🡪 Recursos da aplicação, contem todos os ficheiros necessários para o seu funcionamento. Neste caso apenas contém imagens usadas no GUI.

**Logging** 🡪 Registo de eventos. Todas as tarefas executadas serão expostas para este registo, mostrando na aplicação principal esse mesmo registo.

**ApplicationSettings** 🡪 Contém todas as opções / definições da aplicação. Essas opções são guardadas automaticamente pela aplicação e lidas quando necessário.

## Utilidades

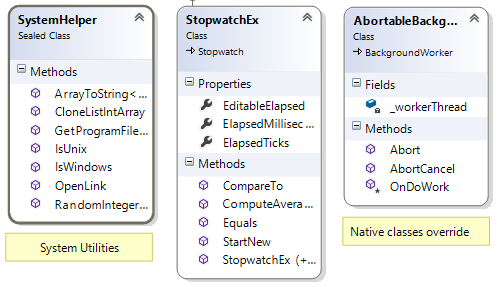


Ilustração 2

Classes de utilidades

**System helper** 🡪 Contém métodos de ajuda tais como: transformar um array em texto, fazer um clone de uma lista, verificar o sistema operativo, abrir um link no browser, fazer um array de números aleatórios.

**StopwatcherEx** 🡪 Herdado da *class* *Stopwatcher*. Contém melhoramentos à *class* suportando a modificação do tempo obtido (para os relatórios baseados em médias), retornar o valor de microssegundos no tipo *double* e um método que calcula a média de uma colecção de *StopwatcherEx*.

**AbortableBackground** 🡪 Herdado da class *BackgroundWorker*. Esta implementação permite abortar o *Thread* sem que seja necessário esperar pelo fim de uma tarefa demorada.

## Sistema de relatórios

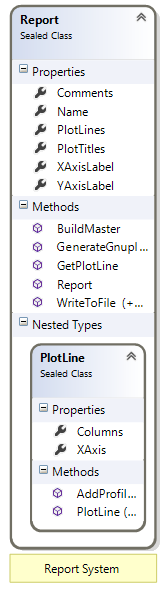


Ilustração 3

**Report** 🡪 *Class* que gera os relatórios. Cada instância desta *class* significa um relatório diferente com os seus próprios resultados.

## Algoritmos

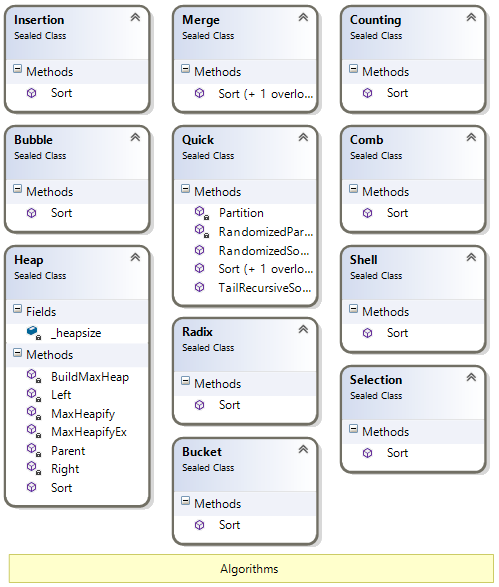


Ilustração 4

Todos os algoritmos disponíveis no programa. Método sort irá executar o algoritmo.

# Módulos

Os ficheiros localizam-se na pasta “src”, essa pasta contem a classe dos algoritmos nos ficheiros (NomeAlgoritmo.cs).

* Insertion.cs
* Bubble.cs
* Heap.cs
* Merge.cs
* Quick.cs
* Radix.cs
* Bucket.cs
* Counting.cs
* Comb.cs
* Shell.cs
* Selection.cs

Todas as classes usam o método Sort

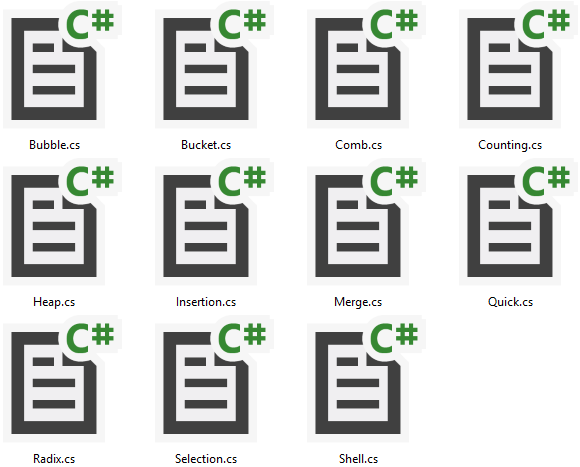


Ilustração 5

# Insertion-Sort

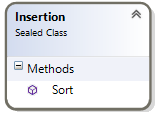


Ilustração 6

## Métodos e Funções

Neste algoritmo de ordenação (Insertion-Sort), temos apenas um método "***sort***", este método pede como parâmetro o array **int [] A**, é constituído por 2 funções (função *for* e *while*), no caso da função *for* "**for (int j = 0; j <A.Length; j++)**", tem como objetivo percorrer todos os elementos do array, enquanto o *while* percorre array A enquanto os valores da chave "***key***" forem inferiores aos elementos do array A, "**while (i> -1 && A [i]> key**) " ou seja, caso esta condição se confirme os elementos de A deslocam-se uma posição para baixo, diminuindo "**j - 1**" até 0, por fim fora do ciclo while a inserção é realizada.

## Desempenho computacional teoricamente previsto de cada algoritmo

Insertion-sort, ou ordenação por inserção, é um algoritmo de ordenação simples e eficiente. Quando aplicado a um pequeno número de elementos. Em termos gerais, ele percorre um vetor de elementos da esquerda para a direita e à medida que avança vai deixando os elementos mais à esquerda ordenados.

Pior caso: O (n²)

Caso médio: O (n²)

Melhor caso: O (n)

## Projeto da experiência que permite obter os resultados para posterior análise experimental

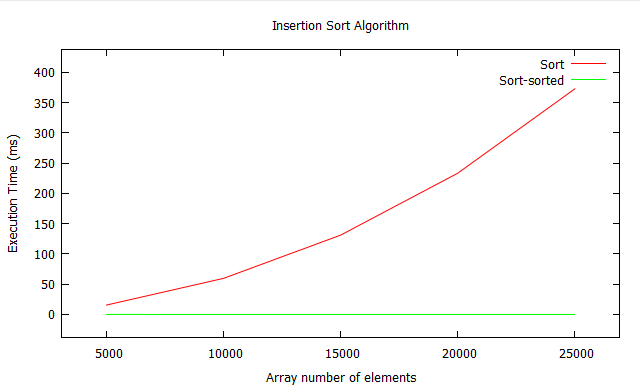


Ilustração 7

# Nº Sort Sort-sorted

5000 15.557 0.0322

10000 59.6854 0.0489

15000 131.1569 0.0651

20000 233.452 0.0833

25000 372.877 0.1003

Podemos verificar segundo este gráfico gerado pela nossa aplicação que se confirma o Pior, médio e o melhor caso, onde o pior caso será sempre crescente e o tempo de execução cresce regularmente dependendo do número de elementos.

O melhor caso confirma-se quando o array já se encontra ordenado, neste caso o tempo de execução será O (n).

# Bubble-Sort

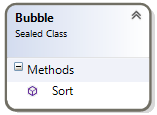


Ilustração 8

## Métodos e Funções

Neste algoritmo de ordenação (Bubble-Sort), temos apenas um método "***sort***", este método pede como parâmetro o array **int [] A**, é constituído por 3 funções (2 função *for* e uma função *if*), no caso da função *for* "**for (int i = 0; i < A.Length; i++)**", tem como objetivo percorrer todos os elementos do array, enquanto que o outro *for* "**for (int j = A.Length - 1; j >= i + 1; j--)**", ao qual atribuímos á variável **int j** o tamanho do array A – 1, este for vai retirando um valor a variável **J** ate ser igual a **i + 1,** este for é constituído por um *if* que tem como condição **if(A[j] < A[j -1])** se esta condição se confirmar o código existente dentro da função *if* ira ser executado, esse código possui uma variável **int** **key** assume **A [j] ,** de seguida o **A [j]** assume **A [j - 1]**,e por fim o **A [j - 1]** assume a variável **key.**

## Desempenho computacional teoricamente previsto de cada algoritmo

O *Bubble-sort*, ou ordenação por flutuação é um algoritmo de ordenação dos mais simples. A ideia é percorrer o vetor diversas vezes, onde a cada passagem retira o maior elemento da sequência.

Pior caso: *O (n2)*

Caso médio: *O (n2)*

Melhor caso: *O (n)*

## Projeto da experiência que permite obter os resultados para posterior análise experimental

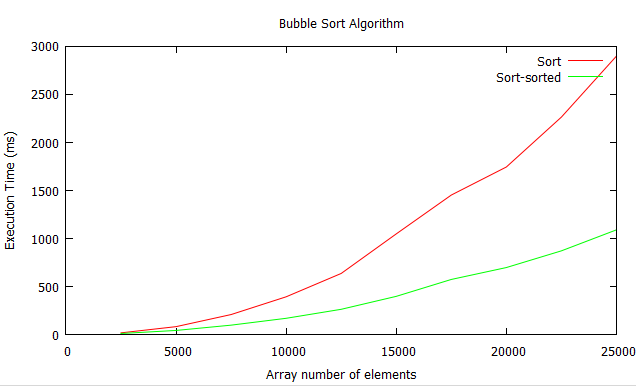


Ilustração 9

# Nº Sort Sort-sorted

2500 17.035 10.5695

5000 81.6614 42.1405

7500 207.6276 97.2523

10000 392.3867 168.5123

12500 635.8622 262.2661

15000 1047.5458 397.3973

17500 1452.1726 573.0037

20000 1744.1806 697.3344

22500 2263.7188 871.1923

25000 2899.2318 1089.7475

Através do gráfico podemos tirar algumas conclusões com por exemplo a comparação do pior para o melhor caso, como esperado o melhor caso demora uma grande diferença em relação ao Pior caso de quando se trata do tempo de execução. Isto acontece devido ao melhor caso *O(n) e o pior caso ser O (n2).*

# Heap-Sort

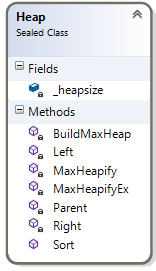
****

Ilustração 10

## Métodos e Funções

Neste algoritmo de ordenação (Heap-Sort) contém apenas uma class que é constituída por 4 métodos, como referi acima no tópico de sistemas de classes, estes métodos são:

* MaxHeapify(int[] A, int i);
* MaxHeapifyEx(int[] A, int i);
* BuildMaxHeap(int[] A);
* Sort (int [] A).

O método **MaxHeapify** é constituídopor 3 funções "*if*" ao qual são executados se seguirem a condição solicitados por a função "*if*" **if(l < \_heapsize && A[l] > A[i])** se o elemento for valido, o largest assume a variável l(left) senão o largest assume a variável i,  **if** (**r < \_heapsize && A[r] > A[largest])** se o elemento for valido a esta condição o largest assume a variável r(right), **if(largest != i)** se o largest for diferente de i, o (**int Key = A[i]**) , o (**A[i] = A[largest]),** o **(A[largest] = key)** e por fim **(MaxHeapify(A, largest)).**

O método **MaxHeapifyEx** é constituídopor 1 função **while** que contém as funções **if** do método anterior expecto (**if (largest == i)**) se o **largest** for igual ao **i** para o **while** e por fim a variável **temp** assume **A [i] (int temp = A [i]),** de seguida o **A [i]** assume **A [largest] (A [i] = A [largest])**,sucessivamente o **A [largest]** assume a variável **temp (A [largest] = temp) e** termina o método atribuindo o **largest** a variável **i (i = largest).**

O método **BuildMaxHeap** é constituído por apenas uma função **for** que por sua vez o método **MaxHeapify** ate a var i for igual a 0, para alem do **for** também define a variável **\_heapsize = A.Length-1** (Tira um valor ao tamanho do array A.

Por fim temos o método Sort que executa o método BuildMaxHeap (A), de seguida retira um valor ao tamanho do array A (\_heapsize **= A.Length-1**) e por fim executa um for que termina quando i for igual a 1 (int **i = A.Length - 1; i> = 1; i**--), o ***for*** atribui o array A [i] a variável Key (**int key = A [i]**), sucessivamente o array A [i] assume o valor A [0] (**A [i] = A [0]**) e por fim A [0] assume o valor Key (A [0**] = key**) para além de ir subtrair um a variável \_ heapsize (**\_heapsize--**) e executar o método **MaxHeapify.**

## Desempenho computacional teoricamente previsto de cada algoritmo

A organização de dados sequenciais feita for o algoritmo Heap permite obter o melhor de 2 mundos, pois contem a qualidade do algoritmo Merge-Sort e a memória do algoritmo Insertion-Sort.

A estrutura Heap é um objeto que é organizado através de uma tabela, sendo esta vista como uma árvore.

Pior cenário: 0 (n log (n))

Caso médio: 0 (n log (n))

Melhor caso: 0 (n log (n))

## Projeto da experiência que permite obter os resultados para posterior análise experimental

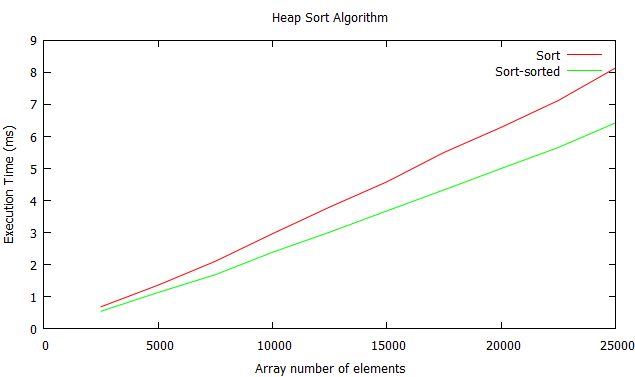


Ilustração 11

# Nº Sort Sort-sorted

2500 0.679 0.5389

5000 1.3563 1.1278

7500 2.1009 1.6816

10000 2.9615 2.3824

12500 3.7942 3.0094

15000 4.5816 3.6742

17500 5.4989 4.3292

20000 6.2845 4.9983

22500 7.1219 5.6547

25000 8.1364 6.4229

No caso do algoritmo de ordenação Heapsort temos uma ligeira diferença em relação ao tempo de execução, neste caso a diferença é menor pois tanto a fórmula do melhor caso como a do pior caso são iguais O (n log n), mesmo assim nota-se alguma diferença pois o tempo de execução diminui em relação ao melhor caso devido ao array já se encontrar ordenado.

# Merge-Sort

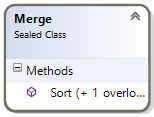
****

Ilustração 12

## Métodos e Funções

Na class Merge usamos 2 métodos, o método "**Sort (int [] A, int p, int q, int r) “**.

O método "Sort (int [] A, int p, int q, int r) " é constituído por 3 funções "for (i = 0; i <n1; i++), for (j = 0; j <n2; i++), for (int k = p; k <= r; k++) " e uma função "if (int k = p; k <= r; k++) ". A primeira função "for" têm como objetivo

## Desempenho computacional teoricamente previsto de cada algoritmo

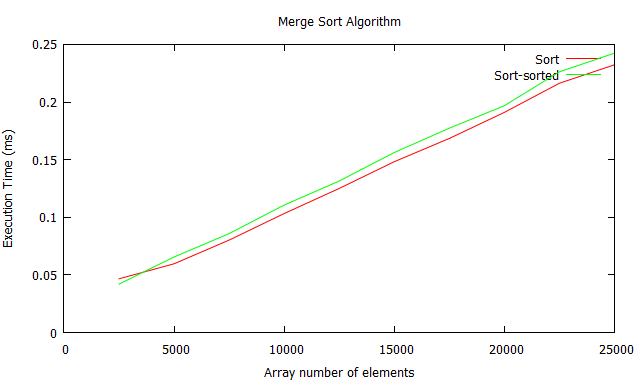
Mergesort divide os elementos do array em sub-tabelas com dimensão menor, depois das sub-tabelas estarem ordenadas, combinam as soluções resultantes para criar a solução do problema original.

Pior cenário: 0(nlogba log (n)) = 0 (nlog22 log (n)) = 0 (n log (n))

Caso médio: 0 (n log (n))

Melhor caso: 0 (n log (n))

## Projeto da experiência que permite obter os resultados para posterior análise experimental



# Nº Sort Sort-sorted

2500 0.0466 0.042

5000 0.0596 0.0655

7500 0.0802 0.0858

10000 0.1032 0.1106

12500 0.125 0.1314

15000 0.1482 0.1562

17500 0.1684 0.1773

20000 0.191 0.1968

22500 0.2163 0.2262

25000 0.2323 0.2424

Como podemos observar neste gráfico obtemos uma diferença mínima entre o melhor caso e o pior caso em relação ao tempo de execução, isto acontece devido as fórmulas de ambos os casos serem totalmente iguais.

# QuickSort

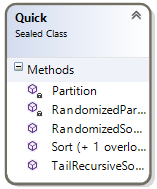


Ilustração 13

## Métodos e Funções

Este algoritmo já possui mais métodos, começando com o Sort que pede como parâmetros 3 variáveis, sendo elas **Array A, int p (Start índex), int r (End índex), Sort(int[] A, int p, int r**), este método possui apenas um função if que possui como condição **(p >= r),** para além disso chama-se a si próprio duas vezes **Sort(A, p, q - 1) e Sort(A, q + 1, r).** De seguida redefinimos o método **Sort** atribuindo apenas o parâmetro do array A e tendo apenas a função de chamar o método **RandomizedSort (A).** Depois do método Sort criamos mais 4 métodos sendo um deles o método **Partition** que contem os mesmos parâmetros do **método Sort Partition (int [] A, int p, int r),** este método contem 2 funções (uma função *for* e uma função *if*), na função *for* criamos uma variável **int j** ao qual atribuímos o a variável **p (Start Index),** esta função termina quando o **j** atingir o valor do **r(End Index),** como se pode ver **for(int j = p; j < r; j++)** este *for* é constituído por uma função *if* onde é criada uma variável **temp** que assume **A [i] ,** de seguida o **A [i]** assume **A [j]**,e por fim o **A [j]** assume a variável **temp,** depois do for ser corrido a variáveltemp assume **A [i] ,** de seguida o **A [i]** assume **A [r]**,e por fim o **A [r]** assume a variável **temp** por fim o método retorna a variável **i.**

Sucessivamente passamos a criação do método **RandomizedSort(int[] A, int p, int r)** que contem novamente os mesmos parâmetros que os métodos anteriores, este método e relativamente pequeno pois é apenas constituído por uma função *if* que possui a condição p>=r, **if (p >= r),** de seguida criamos a variável q a qual atribuímos o método RandomizedPartition **( int q = RandomizedPartition(A, p, r)),** por fim o método chama-se a si próprio 2 vezes **RandomizedSort(A, p, q - 1) e RandomizedSort(A, q + 1, r**), consecutivamente redefinimos a função **RandomizedSort(int[] A**) com o objetivo de o método se chamar a si próprio mas desta vez com diferentes parâmetros **RandomizedSort(A, 0, A.Length-1);**

**Por fim concluímos o código com a criação de dois métodos o RandomizedPartition(int[] A, int p, int r) e TailRecursiveSort(int[] A, int p, int r).**

## Desempenho computacional teoricamente previsto de cada algoritmo

O algoritmo de ordenação QuickSort tem um ótimo desemprenho computacional, apresentando no:

Pior cenário: O (n2).

Caso médio: O (n log (n))

Melhor Caso: O (n log (n))

## Projeto da experiência que permite obter os resultados para posterior análise experimental

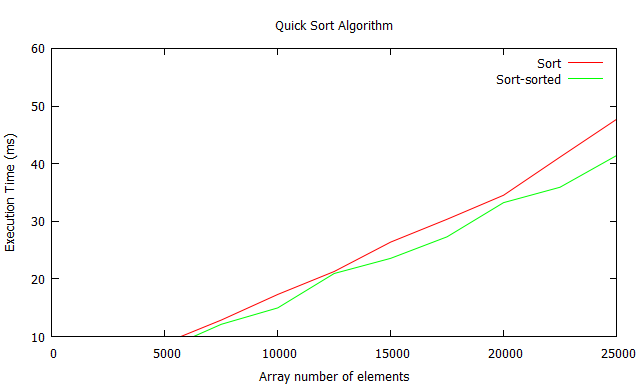


Ilustração 14

# Nº Sort Sort-sorted

2500 4.4372 3.7681

5000 8.8683 7.7838

7500 12.8538 12.12

10000 17.298 14.9595

12500 21.2783 20.9301

15000 26.3824 23.5627

17500 30.3453 27.3019

20000 34.5133 33.2304

22500 41.1341 35.9031

25000 47.6799 41.3942

Devido a ótimo desempenho deste algoritmo de ordenação obtemos um tempo de execução muito curto, para além disso a diferença ente o melhor caso e o caso médio quase que passa despercebido.

# Radix-Sort

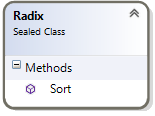


Ilustração 15

## Métodos e Funções

Este algoritmo apenas possui o método Sort “**Sort(int[] A)”,**  começamos a construção do método criando um array **T** ao qual atribuímos o tamanho do **array A,** de seguida criamos 2 variáveis **int r e int b** ao qual atribuímos o valor **4 e 32.**

## Desempenho computacional teoricamente previsto de cada algoritmo

O algoritmo de ordenação Radix-Sort é um algoritmo rápido e estável que pode ser usado para ordenar vários itens que estão identificados por chaves únicas:

Pior caso: O (n + s)

Melhor caso: O (Kn)

## Projeto da experiência que permite obter os resultados para posterior análise experimental

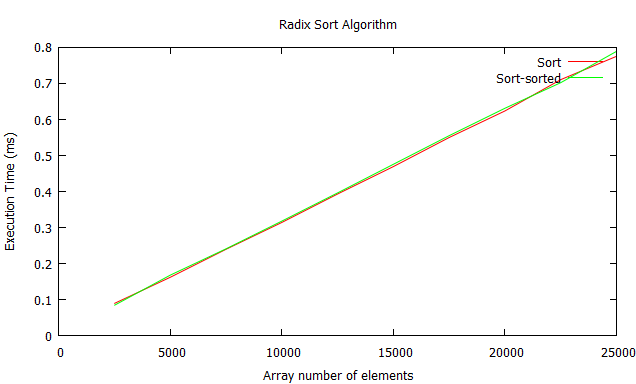


Ilustração 16

# Nº Sort Sort-sorted

2500 0.0894 0.0843

5000 0.1619 0.1679

7500 0.2396 0.2409

10000 0.314 0.3178

12500 0.393 0.3961

15000 0.4691 0.4755

17500 0.5498 0.5549

20000 0.6235 0.631

22500 0.7104 0.702

25000 0.7751 0.7889

No caso do algoritmo do radix a diferença é quase nula pois o radix é um dos algoritmos de ordenação mais rápido e estável, tendo como melhor caso O (Kn) e pior caso O (n + s).

# Bucket-Sort

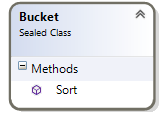


Ilustração 17

## Métodos e Funções

Este algoritmo como muitos outros que vimos anteriormente só possui o método Sort “**Sort (int [] A)”,** desta vez este método é constituído por 5 funções **(4 for e 1 if),** o primeiro for percorre o Array A este for é constituído por uma função if que só é executada se esta condição se confirmar, sucessivamente criamos um **array count**, depois percorremos o array outra vez através da segunda função *for* ao qual é executado o **count [A [i]] ++,** por fim temos os últimos 2 for, onde cabe ao primeiro percorrer o **array count ”for (int i = 0, j = 0; i <count.Length; i++) ”**, enquanto percorre o array é executado o ultimo for **“for (; count [i]> 0; (count [i]) --)** ”.

## Desempenho computacional teoricamente previsto de cada algoritmo

O algoritmo de ordenação bucket-Sort é um algoritmo que divide os elementos do algoritmo em vários recipientes, em seguida organiza os elementos de cada recipiente, usando outros algoritmos ou ele mesmo, os casos são os seguintes

Pior caso: O (n. k)

Caso médio: O (n + k)

Melhor caso: O (n2)

## Projeto da experiência que permite obter os resultados para posterior análise experimental

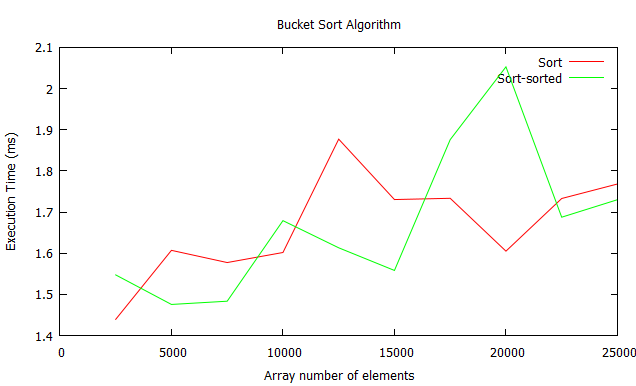


Ilustração 18

# Nº Sort Sort-sorted

2500 1.4392 1.5473

5000 1.6069 1.4755

7500 1.5772 1.4833

10000 1.6017 1.6791

12500 1.8771 1.6131

15000 1.7303 1.558

17500 1.7333 1.8758

20000 1.6049 2.0528

22500 1.7329 1.6874

25000 1.7681 1.73

Como podemos observar no gráfico acima quando executamos o algoritmo de ordenação Bucket-sort, obtemos resultados muito instáveis dai os picos a meio da reta. Em relação à diferença entre o pior caso e melhor caso obtemos uma grande diferença pois o pior caso é representado O (n.k) e o melhor caso é representado por O (n 2).

# Counting-Sort

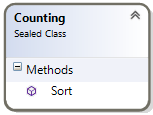


Ilustração 19

## Métodos e Funções

Este algoritmo só possui o método Sort “**Sort (int [] A)”,** este método é constituído por 5 funções **(4 for e 1 if/else if),** em primeiro criamos 2 variáveis **int max e int mim** ao qual atribuímos a ambas **A[0]** sucessivamente criamos um for com o objetivo de percorrer o arrayA este for é constituído por uma função **if/ else if**, ou seja caso **A[i]** seja maior que a variável **max** é atribuído o **A[i]** a variável **max** mas se não se confirmar que o **A[i]** não é maior que o **max** recorremos a segunda opção onde a condição é A[i] tem de ter um valor menor que a variável **min** que só é executada se esta condição se confirmar, sucessivamente criamos um **variável numValues** ao qual atribuímos **max - min + 1,** um array **count** tendo como valor **int[numValues]**, depois percorremos o array outra vez através da segunda função for ao qual executa o **count [A [i] - min] ++,** por fim criamos a variável **outputPos** dando o valor de **0 ,** consecutivamente é executado a função **for “for (int i = 0; i < numValues; i++)”** que é constituído por outra função for **“for (int j = 0; j < counts[i]; j++)”.**

## Desempenho computacional teoricamente previsto de cada algoritmo

O algoritmo de ordenação Counting-Sort é um algoritmo de ordenação estável tendo uma complexidade O (n). As chaves tomam valores de 0 e M-1, mas se existir uma K-1 corresponde a 0.

Pior caso: O (n + k)

Caso médio: O (n + k)

Melhor caso: O (n + k)

## Projeto da experiência que permite obter os resultados para posterior análise experimental

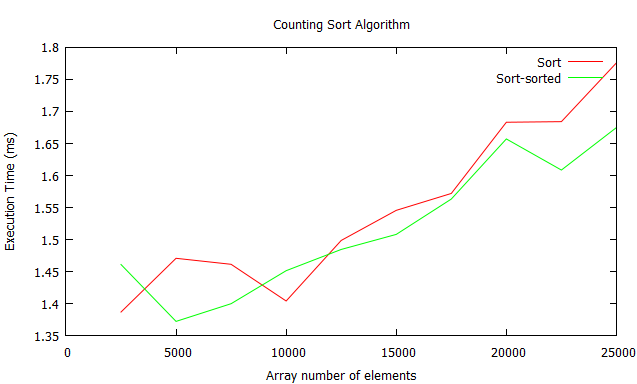


Ilustração 20

# Nº Sort Sort-sorted

2500 1.3866 1.4609

5000 1.4706 1.372

7500 1.4612 1.3997

10000 1.4039 1.4512

12500 1.4985 1.4844

15000 1.5455 1.5079

17500 1.5719 1.5631

20000 1.6831 1.657

22500 1.684 1.6084

25000 1.7759 1.6748

Podemos verificar segundo este gráfico tem umas pequenas irregularidades, mas a nível da diferença entre o melhor caso e o pior caso podemos concluir uma pequena diferença devido ao algoritmo ser estável.

# Comb-Sort

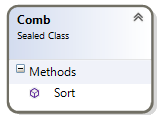


Ilustração 21

## Métodos e Funções

Este algoritmo só possui o método Sort “**Sort (int [] A) ”,** este método é constituído por 4 funções **(1 while, 2 if e 1 for),** em primeiro criamos 2 variáveis **int gap e bool swapped** ao qual atribuímos a variável **gap** o tamanho do array A e ao a variável de valor booleano atribuímos **false** sucessivamente criamos um While que é constituído por as restantes funções, se as condições do *while* **“while ((gap > 1) || swapped)”** forem verificadas passamos para a função if **“if (gap > 1)”** esta função tem como objetivo redefinir o valor da variável gap **“gap = (int) (gap / 1.247330950103979);”**, de seguida percorrermos o array A através da função for **“for (int i = 0; gap + i < A.Length; ++i)”** ao mesmo tempo que executada a função if isto se acabar por se verificar a condição proposta por a mesma “**if (A[i] - A[i + gap] > 0)”.**

## Desempenho computacional teoricamente previsto de cada algoritmo

O algoritmo de ordenação Comb-Sort é um algoritmo de ordenação simples, o comb foi feito com o objetivo de melhorar o algoritmo Bubble-sort, a ideia básica do algoritmo é recorrer a eliminação dos pequenos valores próximos do final da lista.

Pior caso: Ω(n2)

Caso médio: Ω(n2 / 2b)

Melhor caso: O(n)

## Projeto da experiência que permite obter os resultados para posterior análise experimental

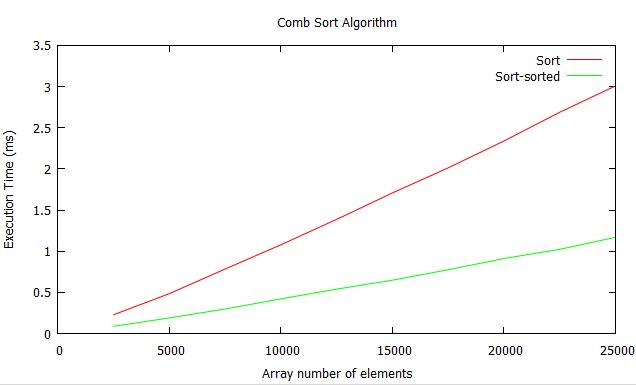


Ilustração 22

# Nº Sort Sort-sorted

2500 0.2283 0.0875

5000 0.4849 0.1894

7500 0.7837 0.2979

10000 1.0779 0.4212

12500 1.3869 0.5392

15000 1.7072 0.6475

17500 2.0124 0.7747

20000 2.3388 0.911

22500 2.6876 1.022

25000 3.0071 1.1688

No caso do Comb-Sort obtemos uma grande diferença a nível do tempo de execução entre o melhor e pior caso, pois como podemos ver através do gráfico o melhor caso tem um máximo de aproximadamente 1.68, enquanto o pior caso tem um máximo de 3.

# Shell-Sort

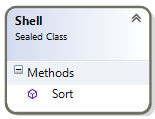


Ilustração 23

## Métodos e Funções

Este algoritmo só possui o método Sort “**Sort (int [] A) ”,** desta vez este método é constituído por 3 funções **(1 for e 2 while),** em primeiro criamos 4 variáveis **int n, int h, int c e int j** ao qual atribuímos a variável **n** o tamanho do array A e a variável h atribuímos **n/2,** as restantes não atribuímos valor nenhum. Sucessivamente criamos um While que é constituído por as restantes funções, se as condições do while **“while (h> 0) ”** forem verificadas passamos para a função for **“for (int i = h; i <n; i++) ”** nesta função atribuímos a variável **int c “c = A[i]”** e a variável j **“j = i”**, por fim temos uma função while “**while (j >= h && A[j - h] > c)”.**

## Desempenho computacional teoricamente previsto de cada algoritmo

O algoritmo de ordenação Shell-Sort é um algoritmo de classificação mais eficiente dentro da categoria complexidade quadrática, este algoritmo recorrendo a ordenação do array considerando que o array e constituído com vários segmentos, sendo aplicada o método de inserção a direita em cada um deles.

Pior caso: O(n log2 n)

Caso médio: O(n)

Melhor caso: O(n)

## Projeto da experiência que permite obter os resultados para posterior análise experimental

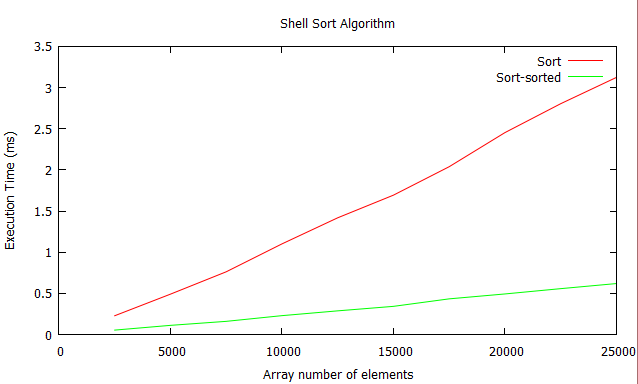


Ilustração 24

# Nº Sort Sort-sorted

2500 0.2276 0.0536

5000 0.4899 0.1109

7500 0.762 0.1599

10000 1.1008 0.229

12500 1.4178 0.2872

15000 1.6927 0.3422

17500 2.0385 0.4331

20000 2.4526 0.4926

22500 2.8037 0.5573

25000 3.1231 0.6192

O gráfico do algoritmo Shell short é muito semelhante ao gráfico do algoritmo anterior (Comb-Sort), mas neste caso ainda obtemos uma maior diferença entre ambos os casos, o melhor caso é representado por O (n) e o piro caso por O (n log2 n).

# Selection-Sort

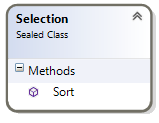


Ilustração 25

## Métodos e Funções

Este algoritmo só possui o método Sort “**Sort (int [] A) ”,** desta vez este método é constituído por 3 funções **(2 for e 1 if),** em primeiro criamos um for que percorre o array A**“for (int i = 0; i < A.Length; i++)”**, esta função contem 4 variáveis **“int minElementIndex = i”**, **“int minElementValue = A[i]”**, **“A[minElementIndex] = A[i]”** e **“A[i] = minElementValue”** mas para além disso a função for conta com mais outras 2 funções for **“for (int j = i + 1; j < A.Length; j++)”** (percorre array A) e por fim o if **“if (A[j] < minElementValue)”.**

## Desempenho computacional teoricamente previsto de cada algoritmo

O algoritmo de ordenação Selection-Sort é um algoritmo de ordenação que se baseia em passar o elemento com menor valor para o primeira posição, ou o maior dependendo da ordem que o utilizador pretende ordenar os elementos, em seguida o segundo menor valor para a segunda posição, continuando assim o processo até os elementos estarem ordenados conforme o pretendido.

Pior caso: O (n2)

Caso médio: O (n2)

Melhor caso: O (n2)

## Projeto da experiência que permite obter os resultados para posterior análise experimental

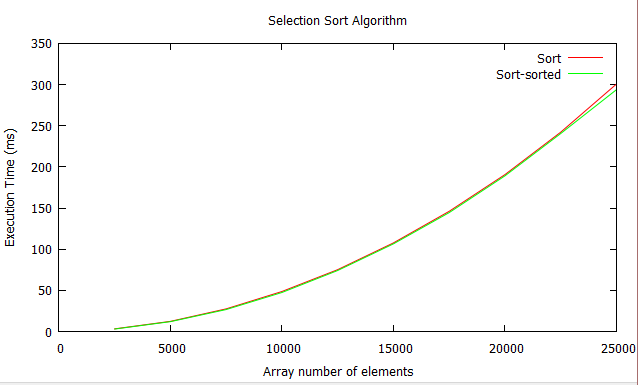


Ilustração 26

# Nº Sort Sort-sorted

2500 3.232 2.9973

5000 12.3287 11.9807

7500 27.4999 26.6452

10000 48.5909 47.4012

12500 75.1666 74.102

15000 107.6313 106.5069

17500 146.0439 144.426

20000 190.6101 189.1041

22500 242.0928 240.4674

25000 300.6341 293.7253

Através do gráfico podemos deter que o Selection-sort é um algoritmo estável, mas possui um tempo de execução elevado devido aos casos serem todos representados por O(n2)

# Comparação de todos os resultados

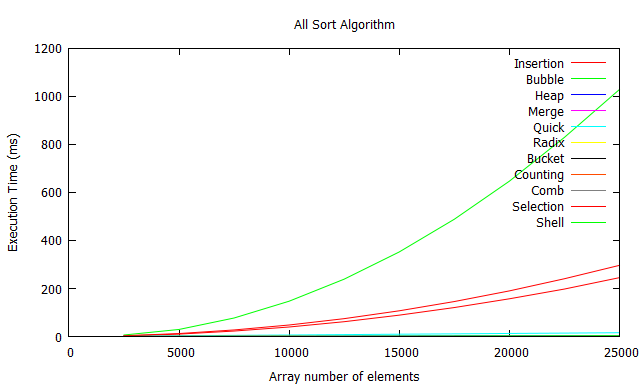


Ilustração 27

Neste gráfico podemos ver a comparação entre todos os algoritmos.

# Conclusão

# Bibliografia

Insertion sort <http://pt.wikipedia.org/wiki/Insertion_sort>

Bubble sort <http://en.wikipedia.org/wiki/Bubble_sort>

Heap sort <http://en.wikipedia.org/wiki/Heapsort>

Merge sort <http://en.wikipedia.org/wiki/Merge_sort>

Quick sort <http://pt.wikipedia.org/wiki/Quicksort>

Radix sort <http://en.wikipedia.org/wiki/Radix_sort>

Bucket sort <http://en.wikipedia.org/wiki/Bucket_sort>

Counting sort <http://en.wikipedia.org/wiki/Counting_sort>

Comb sort <http://en.wikipedia.org/wiki/Comb_sort>

Selection sort <http://en.wikipedia.org/wiki/Selection_sort>

Shell sort <http://en.wikipedia.org/wiki/Shell_sort>