Instituto Politécnico de Beja

Estruturas de Dados e Algoritmos

Relatório

Docente: José Jasnau Caeiro

Abril de 2013

Índice

[Introdução 4](#_Toc353561667)

[Insertion-Sort 5](#_Toc353561668)

[Arquitetura do código 5](#_Toc353561669)

[Sistema de classes 5](#_Toc353561670)

[Variáveis escolhidas 5](#_Toc353561671)

[Métodos e Funções 5](#_Toc353561672)

[Módulos 5](#_Toc353561673)

[Desempenho computacional teoricamente previsto de cada algoritmo 5](#_Toc353561674)

[Projeto da experiência que permite obter os resultados para posterior análise experimental 5](#_Toc353561675)

[Gráﬁcos comparativos do desempenho computacional teoricamente esperado com os resultados experimentais 5](#_Toc353561676)

[Análise dos resultados experimentais obtidos 5](#_Toc353561677)

[Bubble-Sort 6](#_Toc353561678)

[Arquitetura do código 6](#_Toc353561679)

[Sistema de classes 6](#_Toc353561680)

[Variáveis escolhidas 6](#_Toc353561681)

[Métodos e Funções 6](#_Toc353561682)

[Módulos 6](#_Toc353561683)

[Desempenho computacional teoricamente previsto de cada algoritmo 6](#_Toc353561684)

[Projeto da experiência que permite obter os resultados para posterior análise experimental 6](#_Toc353561685)

[Gráﬁcos comparativos do desempenho computacional teoricamente esperado com os resultados experimentais 6](#_Toc353561686)

[Análise dos resultados experimentais obtidos 6](#_Toc353561687)

[HeapSort 7](#_Toc353561688)

[Arquitetura do código 7](#_Toc353561689)

[Sistema de classes 7](#_Toc353561690)

[Variáveis escolhidas 7](#_Toc353561691)

[Métodos e Funções 7](#_Toc353561692)

[Módulos 7](#_Toc353561693)

[Desempenho computacional teoricamente previsto de cada algoritmo 7](#_Toc353561694)

[Projeto da experiência que permite obter os resultados para posterior análise experimental 7](#_Toc353561695)

[Gráﬁcos comparativos do desempenho computacional teoricamente esperado com os resultados experimentais 7](#_Toc353561696)

[Análise dos resultados experimentais obtidos 7](#_Toc353561697)

[MergeSort 8](#_Toc353561698)

[Arquitetura do código 8](#_Toc353561699)

[Sistema de classes 8](#_Toc353561700)

[Variáveis escolhidas 8](#_Toc353561701)

[Métodos e Funções 8](#_Toc353561702)

[Módulos 8](#_Toc353561703)

[Desempenho computacional teoricamente previsto de cada algoritmo 8](#_Toc353561704)

[Projeto da experiência que permite obter os resultados para posterior análise experimental 8](#_Toc353561705)

[Gráﬁcos comparativos do desempenho computacional teoricamente esperado com os resultados experimentais 8](#_Toc353561706)

[Análise dos resultados experimentais obtidos 8](#_Toc353561707)

[QuickSort 9](#_Toc353561708)

[Arquitetura do código 9](#_Toc353561709)

[Sistema de classes 9](#_Toc353561710)

[Variáveis escolhidas 9](#_Toc353561711)

[Métodos e Funções 9](#_Toc353561712)

[Módulos 9](#_Toc353561713)

[Desempenho computacional teoricamente previsto de cada algoritmo 9](#_Toc353561714)

[Projeto da experiência que permite obter os resultados para posterior análise experimental 9](#_Toc353561715)

[Gráﬁcos comparativos do desempenho computacional teoricamente esperado com os resultados experimentais 9](#_Toc353561716)

[Análise dos resultados experimentais obtidos 9](#_Toc353561717)

[Bibliografia 10](#_Toc353561718)

# Introdução

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

* Insertion-sort; ( Feito)
* Bubble-sort; ( Feito)
* Heapsort; ( Feito)
* Mergesort; ( Feito)
* Quicksort. ( Feito)

Este algoritmos de ordenação estão programados em linguagem java seguindo o pseudo-código subjacente presente no livro da disciplina.

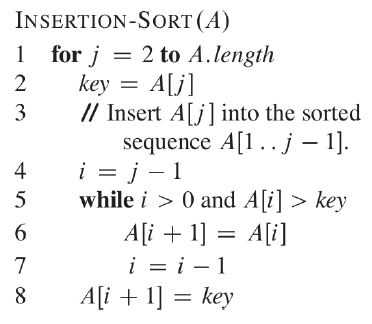
Em relação ao algoritmos iremos mostrar ao longo do relatório o estudo dos mesmo, comparando o desempenho experimental computacional com as previsões teóricas.

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 vistas 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 analise experimental, comparação dos dados previsto com os dados obtidos e por fim a analise dos mesmos.

# Insertion-Sort

## Arquitetura do código



## Sistema de classes

No Insertion-Sort usamos unicamente um método "*sort*" dentro da class " *public sealed class Insertion*", o qual contem o código em C# do Insertion-Sort.

## Variáveis escolhidas

A variável escolhida para a construção do Insertion-Sort foi a seguinte:

**int []A** = Array que contém os algarismo que vão sofrer a ordenação através do Insertion-Sort.

## 

## 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 que 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.

## Módulos

## Desempenho computacional teoricamente previsto de cada algoritmo

A analise do desempenho computacional dos algoritmos é um dos parâmetros mais importantes, esta analise é feita a partir dos tempos de execução os quais dependem da entrada, do numero de elementos a processar e por fim a arquitetura computacional.

Analise Prevista(Em falta)

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

## Gráﬁcos comparativos do desempenho computacional teoricamente esperado com os resultados experimentais

## Análise dos resultados experimentais obtidos

# Bubble-Sort

## Arquitetura do código

## Sistema de classes

## Variáveis escolhidas

## Métodos e Funções

## Módulos

## Desempenho computacional teoricamente previsto de cada algoritmo

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

## Gráﬁcos comparativos do desempenho computacional teoricamente esperado com os resultados experimentais

## Análise dos resultados experimentais obtidos

# Heap-Sort

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 arvore.

## Arquitetura do código

(Em falta)

## Sistema de classes

O código do algoritmo Heap-Sort é constituído apenas por uma classe e esta por sua vez é constituído por 4 métodos que são:

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

## Variáveis escolhidas

A variáveis escolhidas para a construção do Heap-Sort foi a seguinte:

* **\_heapsize =** representa os elementos que a irão ser referidos na tabela A**;**
* **Left(int i) =** esta variável retorna "i \* 2 +1" com o objetivo de definir o no descendente esquerdo**;**
* **Right(int i) =** esta variável retorna "left(i) +1" com o objetivo de definir o no descendente direito**;**
* **Parent =** esta variável retorna "(i-1) / 2" com o objetivo de definir o no progenitor**;**
* **int[] A =**  array que contém os elementos para serem organizados**;**
* **int i**;
* **int l =** Nó esquerdo**(i);**
* **int r =** Nó direito**(i).**

## 

## 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 a esta condição o largest assume a variável l(left) senão for valido 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.**

## 

## Módulos

## Desempenho computacional teoricamente previsto de cada algoritmo

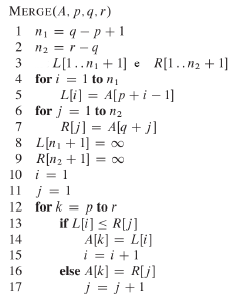
## Projeto da experiência que permite obter os resultados para posterior análise experimental

## Gráﬁcos comparativos do desempenho computacional teoricamente esperado com os resultados experimentais

## Análise dos resultados experimentais obtidos

# MergeSort

## Arquitetura do código



## Sistema de classes

No Merge usamos unicamente um método "*Merge*" dentro da class " *public sealed class Merge*", o qual contem o código em C# do Merge.

## Variáveis escolhidas

As variáveis escolhidas para a construção do Merge foram as seguintes:

**int []A** = Array que contém os algarismo que vão sofrer a ordenação através do Insertion-Sort;

int p = primeiro elemento do array ;

int q = segundo elemento do array;

int r = comprimento do array A.length;

## 

## Métodos e Funções

Na class Merge usamos 2 métodos , o método "**Sort(int[] A, int p, int q, int r)"** e reutilizamos o método "sort" da class Insertion-Sort.

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 objectivo

## Módulos

## Desempenho computacional teoricamente previsto de cada algoritmo

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

## Gráﬁcos comparativos do desempenho computacional teoricamente esperado com os resultados experimentais

## Análise dos resultados experimentais obtidos

# QuickSort

## Arquitetura do código

## Sistema de classes

## Variáveis escolhidas

## Métodos e Funções

## Módulos

## Desempenho computacional teoricamente previsto de cada algoritmo

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

## Gráﬁcos comparativos do desempenho computacional teoricamente esperado com os resultados experimentais

## Análise dos resultados experimentais obtidos

# Bibliografia