Nota: Gráficos gerados com base na média de 1000 execuções de cada algoritmo, para cada ***n***, de modo a minimizar os picos registados .

Neste trabalho pretendeu-se **analisar e estudar** diversos **algoritmos de ordenamento**. Na primeira página deste relatório são apresentados os nomes dos mesmos, bem como uma análise quanto à sua complexidade. Com base numa simulação de casos de teste distintos (arrays ordenados, inversamente ordenados, aleatoriamente ordenados e com todos os elementos iguais) são apresentados na segunda página deste relatório os gráficos correspondentes.

Tarefa A-Insertion Sort (Não recursivo e Estável):

```
for(i=1;i<num;i++) { //num = numero de elementos
    String temp=array[i];
    for (j=i-1; j>=0 && array[j].compareToIgnoreCase(temp)>0;j--){
        array[j+1]=array[j];
    }
    array[j+1]=temp;
}
```

Melhor caso: (*array completamente ordenado*): ***O(n)***

– Compara apenas 1 vez cada elemento, ou seja, **n** vezes, pois o elemento **i nunca é maior** do que o elemento **i-1**.

Pior caso: (array ordenado pela ordem inversa): ***O(n^2)***

– Compara os **n** elementos, **n vezes**, pois o elemento **i é sempre maior** do que os restantes .

Caso médio: (array aleatoriamente ordenado): ***(O(n) +O(n^2))/2 = O(n^2)***

Tarefa A-Quicksort Melhorado (Recursivo e Não Estável):

Melhor caso: (quando o pivot divide a sequência em duas de igual tamanho): ***O(n log n)***

Pior caso: (quando o pivot é o maior ou menor elemento da série): ***O(n^2)***

```
for(i=low,j=high-1;;){

    while( s[++i].compareToIgnoreCase(pivot)<0);
    while( pivot.compareToIgnoreCase( s[ --j ] )<0 );
    if(i<j)
        swap( s, i, j );
```

– Resulta em arrays com grande número de elementos, e arrays com poucos ou nenhuns (arrays degenerados), o que leva a que não se aproveite a divisão destes em arrays menores, mais rápidos de percorrer e ordenar, retirando assim o príncipio que torna este algoritmo atractivo. É o facto de em sucessivas divisões os arrays serem de igual tamanho, que faz com que N elementos sejam percorridos log n vezes, tornando o algoritmo ***O(n log n).***

Caso médio: (array aleatoriamente ordenado): ***O(n log n)***

Tarefa B-Three-way Radix Sort (Recursivo e Não Estável):

Algoritmo híbrido entre MSD e Quicksort.

Melhor caso: (array completamente ordenado – *Apenas usa o MSD*): ***O(k*n) = O(n);***

-Compara as **k** letras das **n** chaves, e não existe nada para trocar, não usa o Quicksort, apenas usa o MSD.

Teoricamente, o caso médio seria *O(k*n log n) = O(n log n),* porque vai dividindo o array em 3 partes diferentes, comparando os caracteres com o MSD e ordenando com o uso do Quicksort**, no entanto, observamos que continua aproximadamente O(n), pelos valores obtidos e pela curva no gráfico.**

Tarefa B-LSD Sort (Não recursivo e Estável):

Complexidade (todos os casos): *O(n*k) + O(n+r) = O(n*k),* onde **k = tamanho da maior String e *O(n+r)* é o tempo gasto a "completar" as palavras** que têm tamanho diferente do tamanho da maior palavra . No meu caso, completei as palavras com " "(espaços em branco), para que todas as palavras tivessem o mesmo tamanho e o algoritmo funcionasse correctamente.

Este algoritmo apresenta **sempre a mesma complexidade** para todos os casos, estando esta **dependente do tamanho das Strings** a ordenar. Sendo assim, para **alfabetos maiores** (que requerem menos símbolos para representar uma palavra), **a execução será mais rápida**, mas será necessário o **consumo de mais memória** para contar as ocorrências dos caracteres. Com um **alfabeto mais pequeno**, binário por exemplo, será ocupada **menos memória** mas levará **mais tempo a executar**, uma vez que as palavras serão tendencialmente maiores. Uma grande vantagem deste algoritmo em relação a outros, é a sua estabilidade.

1º for – Acerta o tamanho das palavras;
2º for – Coloca a 0 a matriz de contagem;
3º for – Conta as ocorrências de cada caracteres;
4º for – Saber a posição onde vai ficar a palavra, dada pelo array de contagem;

Análise de Resultados:

Através da visualização dos gráficos, constata-se que, **para situações** em que os elementos **estão aleatoriamente inseridos**, ou mesmo em casos em que estão **completamente desordenados**, os algoritmos que operam com **Radixes,** os da Tarefa B, são folgadamente **mais eficientes** do que os outros. Para situações em que os arrays estão **completamente ordenados**, ou **pouco desordenados**, e em casos em que a grande **maioria das chaves são iguais**, o **Insertion Sort** mostra-se bastante **eficiente**, daí ser usado como complemento no algoritmo de Quicksort. O algoritmo **LSD**, como apenas **depende do tamanho das Strings**, vai ser tanto mais eficiente quanto menores forem estas. A grande vantagem deste algoritmo **é a sua estabilidade**, o que o torna bastante atractivo para manipulação de certos dados, em que é importante manter a ordem. A sua eficiência é também bastante atractiva, sendo quase tão bom em termos de eficiência como o **Three-Way Radix Sort**, que se demonstrou o **algoritmo mais rápido**, na **média** de todos os casos testados. O **Quicksort** melhorado, apesar de no caso médio apresentar valores razoáveis, nos casos degenerados tende a ser um algoritmo pouco eficiente, pois tende a ser Quadrático em vez de ser N log N.

Gráficos: