# Trabalho Prático 1

## Estrutura de dados – TW

**Luis Antonio Duarte Sousa - 2023001964**

Departamento de ciência da computação – Universidade Federal De Minas Gerais (UFMG)

05/04/2024 – Belo Horizonte
luisduarte-dev@ufmg.br

## 1. Introdução

O problema proposto consiste na realização de uma análise comparativa, tanto teórica quanto experimental, do desempenho de diversos algoritmos de ordenação em uma variedade de cenários de carga de trabalho. Paralelamente, serão coletados os dados fornecidos pela execução dos testes, tais como **tempo** de execução e tamanho do **objeto**, com o objetivo de examinar a complexidade assintótica dos algoritmos.

Os resultados obtidos a partir dos testes serão apresentados em visualizações gráficas adequadas e utilizados para deduzir conclusões comparativas sobre o ambiente e as circunstâncias mais propícias para a aplicação de cada método de ordenação. Além disso, serão identificadas características distintivas de cada algoritmo.

Os testes serão conduzidos por meio de um código implementado, o qual permitirá a modificação da configuração inicial da ordenação, do número de elementos e do tamanho de cada elemento. Essa flexibilidade será devidamente explicada em relação a todos os algoritmos avaliados, os quais serão avaliados em termos de tempo de execução.

Portanto, resumidamente, a solução se baseia na implementação de todos os algoritmos, utilizando std apensa para gerar numeros aleatórios e marcar tempo de execução. De forma complementar também será mostrado o comportamento de variações/melhorias dos algoritmos.

## 2. Método e implementação

### 2.1. Configurações da máquina

- Sistema Operacional: Windows 11 pro
- Ambiente executado: WSL 2 com Ubuntu 22.04
- Compilador: GCC - 13.2
- RAM (Random Access Memory): 24 GB
- Processador: AMD ryzen 5 7535HS

### 2.2. Implementação

A implementação consiste em um programa em C++ para testar o desempenho de diferentes algoritmos de ordenação em uma variedade de cenários.

### 2.2.1. Estruturas de dados

Todas as estruturas de dados utilizadas foram implementadas com objetivo de realizar testes e possibilitar algumas implementações dos algoritmos.

### 2.2.2. TADs e/ou Classes

```
struct Object {
   int key;
   int content[SIZEBYTES];
   ...
};
```

Estrutura desenvolvida para simular tipos de dados complexos, oferecendo a personalização do espaço em bytes. Esta funcionalidade é alcançada através de um array interno de inteiros, onde cada um representa 4 byte, e pode ser preenchido com um número variável de inteiros que é definido em tempo de compilação.Além disso, a classe conta com a sobrecarga dos operadores,o que facilita sua utilização em algoritmos. A comparação entre instâncias da classe é realizada com base em um índice inteiro(4 bytes).

```
template <typename T>
class CountingPair {
public:
   T first;
   int second;
   ...
};
```

Classe que permite o armazenamento do objeto com sua quantidade de aparições, utilizada para facilitar a implementação de um Counting Sort generalizado que ordena elementos complexos pela sua chave.

```
template<typename T>
class SortedLinkedList : public Container<T> {
   private:
      Cell<T>* head;
      Cell<T>* tail;
      ...
};
```

Classe que implementa uma lista encadeada que os itens já são inseridos ordenados e foi utilizada para implementação dos Buckets, dentro do Bucket Sort.

### 2.2.3. Funções e/ou Métodos

```
void printArray(Object arr[], int n) {
...
}
```

Imprime os elementos atuais do array.

```
void execute(Object array[], int size){

   Object aux[size];

   copy_array(array,aux, size);

   auto start = std::chrono::high_resolution_clock::now();
   sort::BubbleSort(aux, size);
   auto end = std::chrono::high_resolution_clock::now();
   auto duration = std::chrono::duration_cast<std::chrono::microseconds>(end - start);
   std::cout << "Bubble time: " << duration.count() << " us" << std::endl << std::endl;

   copy_array(array,aux, size);
   ...
}
```

Executa todos os algoritmos implementados em cópias do array com intuito de que todos tenham os mesmos elementos de entrada. Além disso, demarca o tempo em microssegundos e imprime em seguida.

### 2.3.4. Cargas de trabalho

As cargas de trabalho escolhidas irão girar em torno de 3 variáveis que são número de elementos do vetor, tamanho do elemento e configuração/distribuição inicial. Os experimentos serão feitos mantendo uma das variaveis e modificando outra para ser visivel a sensibilidade a essas mudanças.O tamanho do elemento terá impacto nos algoritmos que envolvem atribuições, cópias ou trocas de elementos. Por outro lado, o número de elementos afetará a escala do problema, enquanto a configuração inicial demonstrará características únicas de comportamento dos algoritmos de ordenação.

## 3. Análise de complexidade

A complexidade de tempo do código depende de dois fatores: o número de elementos ***N*** do array e a ordenação inicial selecionada para a execução dos algoritmos. Primeiramente, o algoritmo preenche o vetor com números aleatórios, realizando uma operação de **O(1)** para cada um dos **N** elementos do array, resultando em uma complexidade de $O(n)$**.** Em seguida,o array é ordenado por uma função $f(n)$ que utiliza todos os algoritmos de ordenação implementados conforme solicitado. Em geral, $f(n)$ tem complexidade $O(n)$ no melhor caso, quando o array já está ordenado naturalmente e $O(n^2)$ no pior caso. Após a ordenação, é imprimido os tempo para comparação que é **O(1).**

Melhor caso: $O(n) + O(1) + O(n) \ldots = O(n)$
Pior caso**:** $O(n) + O(n^2) + O(n^2) \ldots = O(n^2)$

A complexidade de espaço também depende de dois fatores: o número **N** de elementos e o tamanho **K** do objeto ordenado. A princípio está sendo considerado toda memória usada na análise, ou seja, no main e não nos algortimos. Dito isso, a geração do objeto que será ordenado aloca 4 bytes do index inteiro somado com **4K** bytes decididos para o experimento e multiplicado pelo tamanho **N** do array teremos **N(4K + 4)** bytes, logo temos **O(N4K + 4N**). Após essa operação todas as operações que envolvem são de ordemmenor ou igual a ordem anterior.

Ordem de **θ(NK)**

## 4. Estratégia de robustez

Em primeiro lugar, a função main válida os argumentos de linha de comando de forma defensiva, verificando se os parâmetros fornecidos são válidos e se há valores apropriados para tamanho e tipo de ordenação. Isso ajuda a evitar erros de execução devido a entradas inválidas. Além disso, a estrutura “Object” e as funções relacionadas foram implementadas de forma defensiva. As sobrecargas de operadores foram cuidadosamente definidas para garantir que operações como comparações e atribuições sejam tratadas corretamente, minimizando assim a possibilidade de comportamento inesperado ou indefinido. Para tolerância a falhas, o código inclui verificações para lidar com casos em que a alocação dinâmica de memória pode falhar. Por exemplo, antes de realizar a ordenação em um array, é feita uma cópia do array original para um buffer temporário, garantindo que a operação de ordenação não modifique o estado original dos dados em caso de erro.

## 5. Análise experimental

As análises serão feitas inicialmente de forma individual para classes de complexidade de tempo (ou próximas delas no caso de alguns métodos) de algoritmos para facilitar a visualização gráfica de cada um. Caso contrário, a visualização ficaria prejudicada quando existir uma discrepância grande de tempo. Dito isso, gráficos das dimensões de análise serão gerados com os dados coletados.  O eixo Y sempre representará o tempo em microssegundos, já o eixo X poderá representar o tamanho em bytes do Item ou o número de elementos.
Em geral os experimentos são:

1. **Tempo x Entrada:** Testar um array aleatório  com N elementos, onde N varia de (10000 ,100000) , e o intervalo dos numeros gerados é (1, 10000). O tamanho do item nesses teste é irrelevante para não ter uma sobrecarga desnecessária e será constante de 2 inteiros (8 bytes).

2. **Tempo x Tamanho do item:** Testar um array bem definido de 10 elementos para que todos os algoritmos tenham as mesmas movimentações sempre. Dito isso, será variado o tamanho em bytes do item no intervalo (4004, 400004).
   **Object  array[] = {Object(3), Object(7), Object(4), Object(8), Object(9), Object(2), Object(1), Object(5), Object(3), Object(10)};// Array utilizado**

3. **Configuração x Desempenho:** Nessa etapa os algoritmos serão testados com um numero significante de elementos como 1000 e um tamanho de item razoavel como 40 bytes mas variando as configurações iniciais para demonstrar alguma características de cada um.

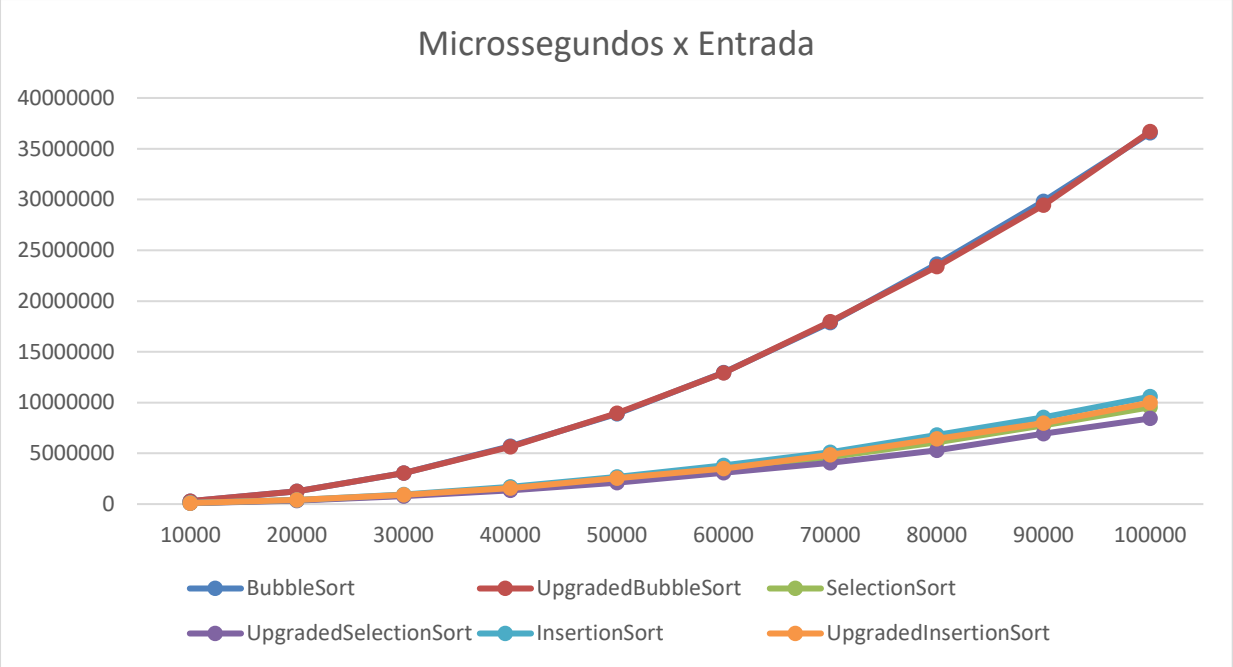



*Figura 1*

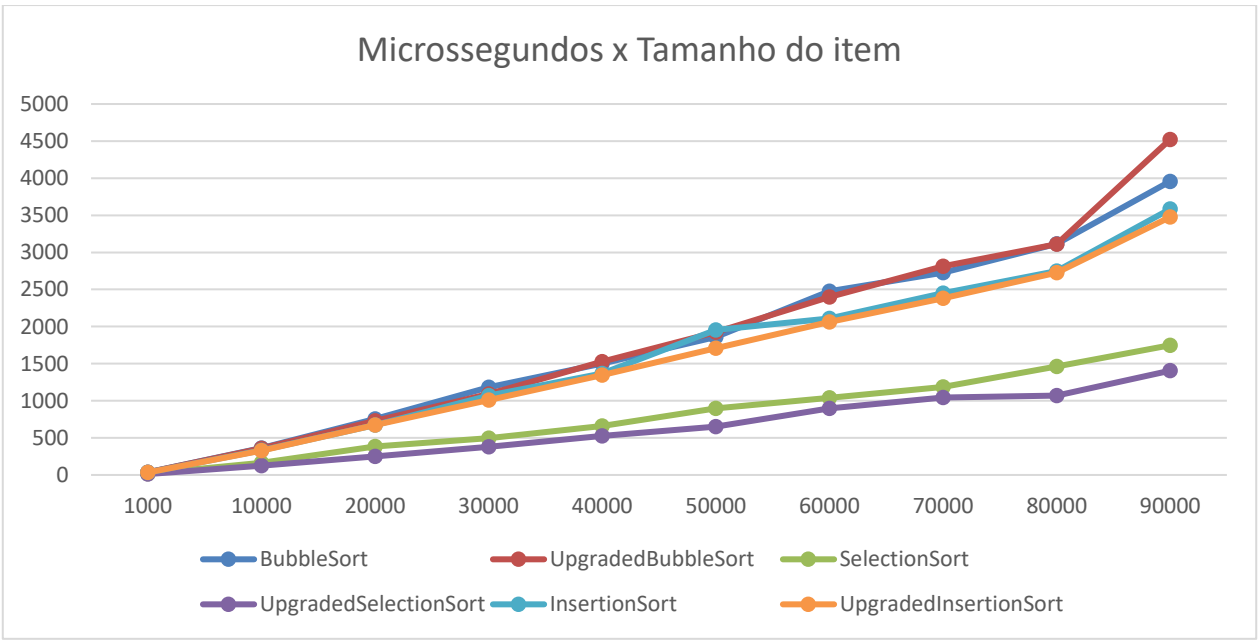



*Figura 2*

Primeiramente vamos explicar o que são essas variações dos algoritmos e quais suas propostas para melhorar algum aspecto dos originais:

- **Upgraded Bubble:** È adicionado um atributo booleano que verifica quando acontece trocas e caso não haja, da um break no loop externo. O objetive dessa mudança é evitar comparações em casos em que a entrada esta completamente ou parcialmente ordenada.
- **Upgraded Selection:** Ao invés de iterar de 0 a (n-1) será iterado de 0 a (n/2 – 1) e cada iteração sera armazenado não somente o minimo mas também o maximo, assim preenchendo ambas as extremidades ao mesmo tempo. O objetivo é aumentar um pouco a velocidade o algoritmo pelo preço de uma variável T max;
- **Upgraded Insertion:** A melhoria adicionada se baseia no posicionamento de um sentinela que fica na posição array[0] e que esse sentinela seja o elemento mínimo do array. O objetivo é tirar completamente as comparações de indices no while interno e manter apenas comparações de chave pelo preço de se encontrar o minimo no array, mas pode ser util caso o minimo já seja conhecido.

Através da figura 1, é evidente o impacto significativo do tamanho do vetor no desempenho dos algoritmos de ordenação com complexidade de tempo quadrática. Isso ocorre devido ao grande número de comparações e movimentações realizadas durante o processo de ordenação. Esses algoritmos compartilham a característica de visitar repetidamente as mesmas posições no vetor em busca de elementos mínimos ou posições corretas, o que resulta em um aumento do tempo de execução à medida que o tamanho do array aumenta. Além disso, como demonstrado pelos dados da figura 2, esses algoritmos são sensíveis linearmente ao tamanho dos elementos sendo ordenados, devido ao grande número de movimentações necessárias. Portanto, esses algoritmos podem se tornar impraticáveis para vetores muito grandes devido ao seu tempo de execução prolongado e deve-se ser cauteloso com o que está sendo ordenado.

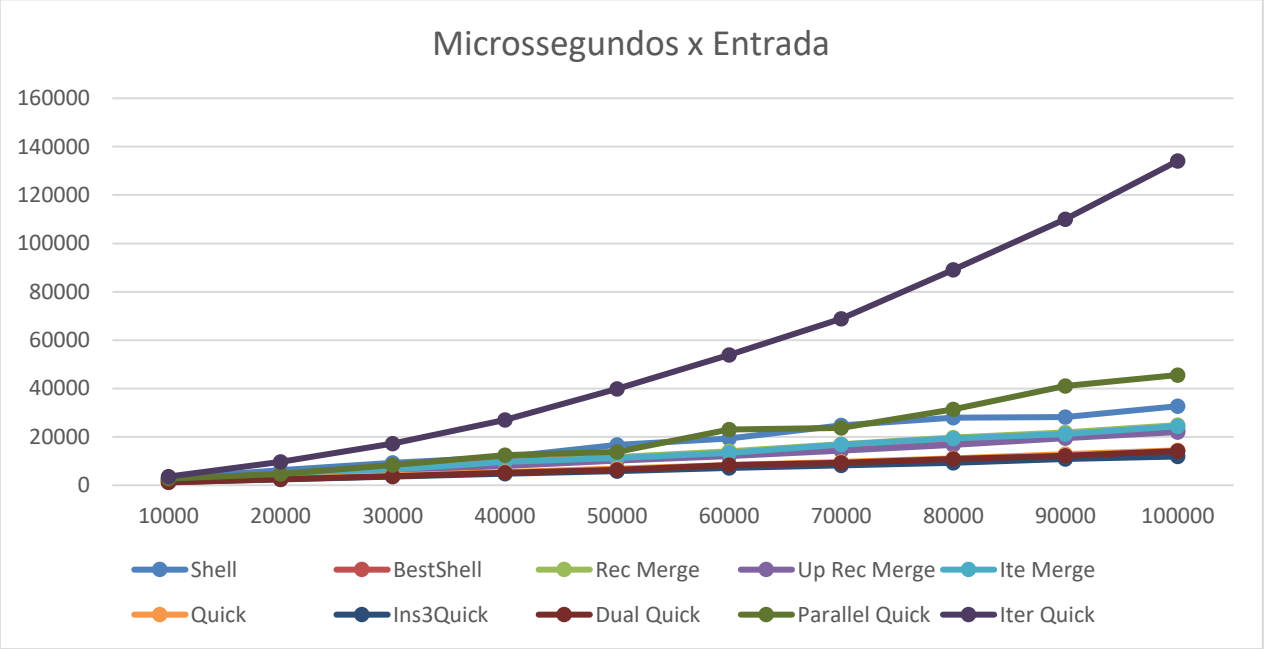



*Figura 3*

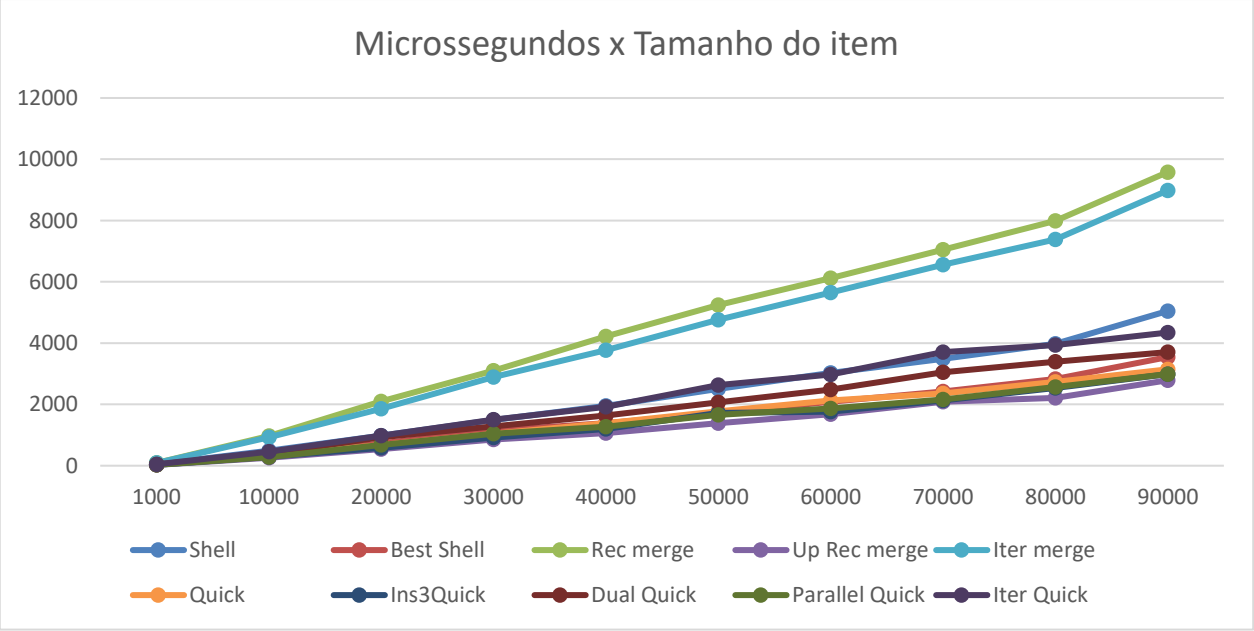



*Figura 4*

Propostas de variações dos algoritmos :

- **Best Shell**: A complexidade e desempenho do shellsort é totalmente dependente da sequencia escolhida para decidir os intervalos de busca no array. Dito isso, eis uma implementação utilizando a melhor sequencia conhecida até hoje que reduz sua complexidade de $n^2$ para n^(4/3). Essa sequencia foi criada por Robert Sedgewick e sua equação é:

  **4^(k) + 3 * 2^(k-1) + 1, prefixed with 1**

- **Upgraded Recursive MergeSort:** Ao invés de ordenar por meio do merge utilizamos insertion sort para tamanhos menores que um **LIMITE** definido.Além disso fazemos uma verificação se ja esta ordenado antes de iniciar um merge.
  **if (arr[mid] > arr[mid + 1]) merge(arr, left, mid, right);**
  O objetivo é reduzir o merges desnecessários e utilizar o insertion para aumentar a velocidade de ordenação;

- **Iterative Merge e Quick:** A implementação não recursiva dos algortimos de divisão e conquista pode ser uma boa escolha em ambientes com pouca memória pois a pilha de execução é cara e complexa. Mas em certos casos pode ser menos eficientes em tempo por sua maior quantidade de comparações.

- **Insertion median Quick:** Mescla a utilização de insertion sort para arrays pequenos e além disso usa a mediana dos elementos primeiro, meio e ultimo. Essa seleção tem intuito de evitar que o algoritmo caia no pior caso que é $n^2$.

- **Dual Quick:** Utiliza dois pivots para ordenção com intuito de melhorar o um pouco o tempo pelo preço de mais complexidade e variaveis auxiliares. Entretanto, não tem uma seleção eficiente de pivot, o que pode levar ao pior caso mais vezes.

- **Parallel Quick:** Usa outras threads para paralelizar o processo de partição do quicksort, usando um **LIMITE** para decidir se cria outra thread ou não. Isso é importante pois caso seja pequeno não vale a pena utilizar outra thread.

A figura 3 revela uma diferença marcante em relação aos algoritmos de complexidade n log n, conhecidos por sua abordagem de divisão e conquista (exceto o Shell Sort). Esses algoritmos demonstram um desempenho significativamente mais eficiente em comparação com os algoritmos de complexidade quadrática. Isso ocorre porque, ao invés de percorrer repetidamente o array como os algoritmos quadráticos, os algoritmos n log n dividem o array inicial em sub-arrays (sejam in-place ou não) e cada sub-arranjo é resolvido de forma separada. Essa abordagem reduz consideravelmente o número de comparações e movimentações de elementos, resultando em um tempo de execução muito mais eficiente. No entanto, é importante notar que, embora esses algoritmos sejam menos sensíveis ao tamanho do array em comparação com os algoritmos quadráticos, ainda há considerações adicionais a serem feitas, especialmente em relação ao uso de memória extra. Por exemplo, o Merge Sort, que é notório por sua eficiência e estabilidade, pode exigir uma quantidade significativa de memória adicional para a mesclagem dos sub-arrays ordenados. Alem disso, é visível na figura 4, que esses algoritmos são bastante linearmente sensiveis ao tamanho do item uma vez que eles fazem muitas movimentações durante o processo de ordenação.

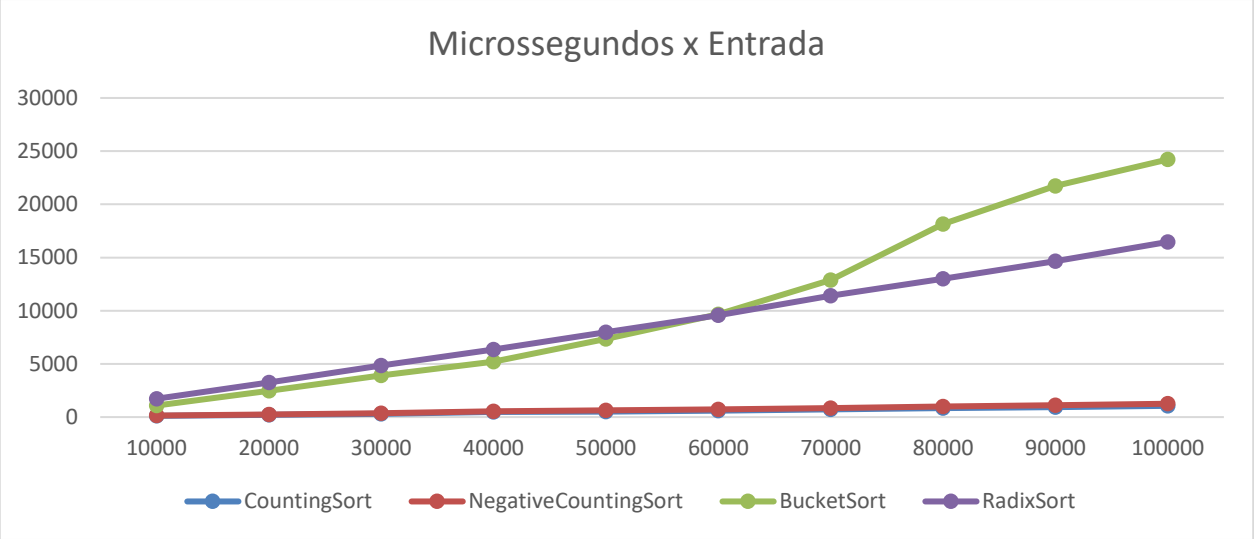



*Figura 5*

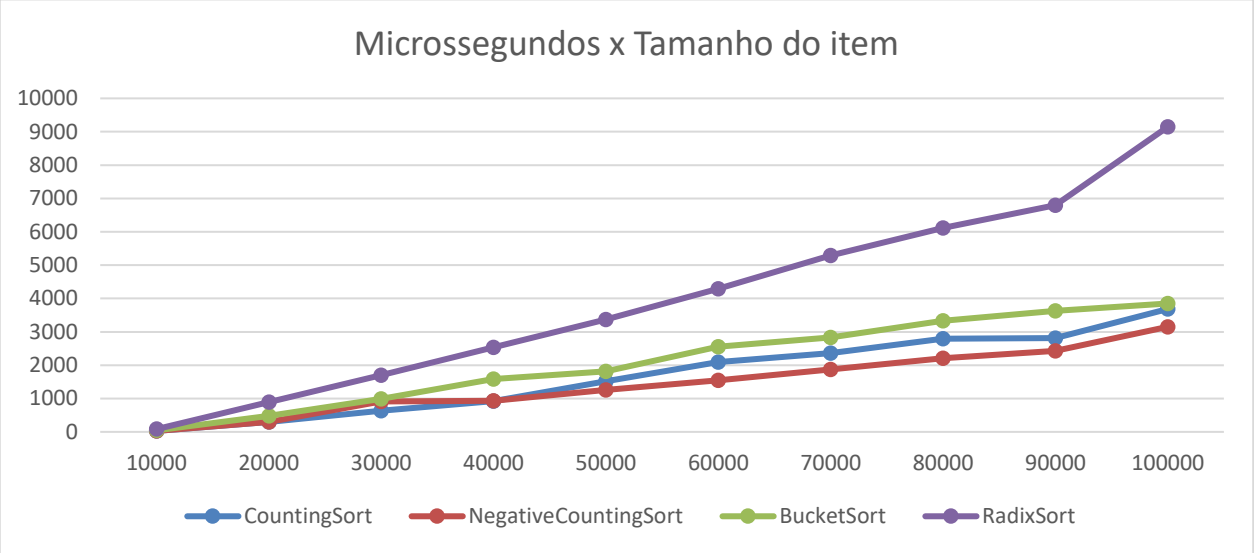



*Figura 6*

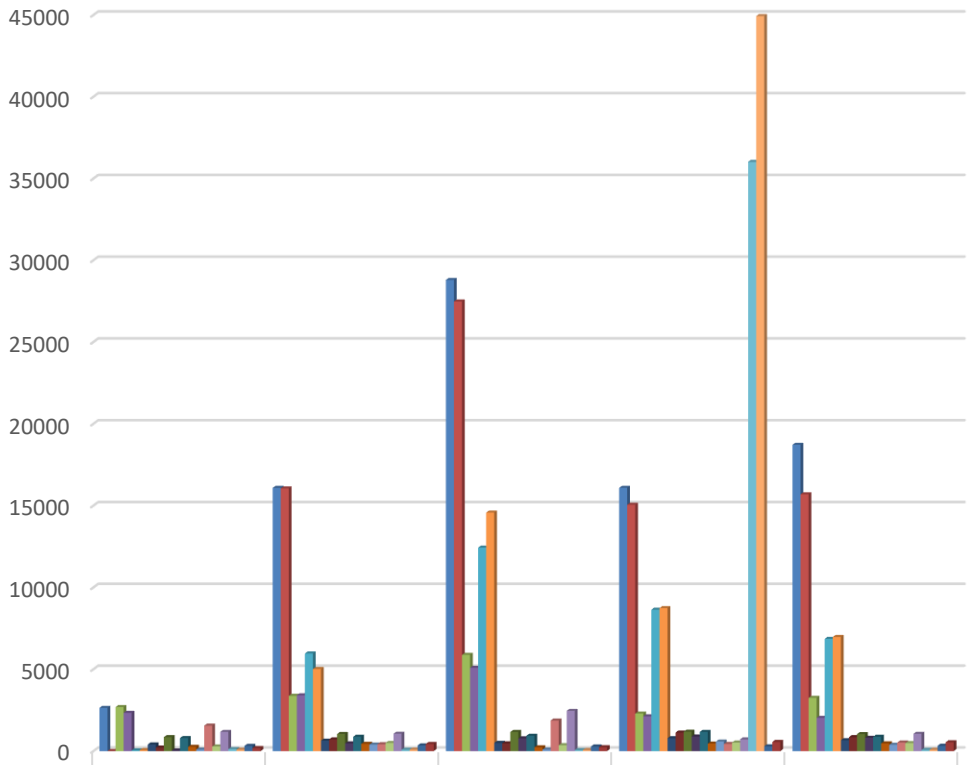
Propostas de variações de algoritmos:

- **Negative Counting:** A proposta dessa implementação é criar um Counting sort que também aceita numeros negativos pelo preço de ter que encontrar, além do maximo, o minimo:

  ```
  template<typename T>
  void sort::NegativeCountingSort(T arr[], int size, const T& max, const T& min) {
  ...
  }
  ```

  É um preço de memória e complexidade pela maior generalização e aplicatibilidade do método de ordenação.

Os métodos de ordenação sem comparação são afetados pelo tamanho do array de diferentes maneiras que os outros. Enquanto o Counting Sort e o Bucket Sort são mais sensíveis à distribuição dos elementos e o intervalo entre o maior e menor elemento, o Radix Sort é mais influenciado ao trabalho de se extrair os bits, então quanto mais numeros “grandes”, mais bits. No entanto, é possível observar, conforme evidenciado na figura 5, que mesmo não tendo comparações ,eles ainda podem ser bastante influenciados pelo tamanho do array por causa das movimentações.Em contra-partida, em questão de memória eles utilizam grande quantidade auxiliar.




| | Ordenado | Parcialmente Ordenado | Inversamente Ordenado | Grande Intervalo de elementos | Muitas repetições de elementos |
|---|---|---|---|---|---|
| Bubble | 2635 | 16082 | 28793 | 16084 | 18714 |
| Upgraded Bubble | 4 | 16051 | 27479 | 15051 | 15691 |
| Selection | 2690 | 3375 | 5878 | 2290 | 3254 |
| Upgraded Selection | 2333 | 3407 | 5087 | 2125 | 2030 |
| Insertion | 49 | 5964 | 12431 | 8640 | 6858 |
| Up Insertion | 60 | 5014 | 14574 | 8732 | 6969 |
| Shell | 402 | 633 | 497 | 784 | 664 |
| Best Shell | 213 | 710 | 468 | 1134 | 851 |
| Rec merge | 850 | 1044 | 1169 | 1183 | 1038 |
| Up Rec merge | 39 | 470 | 777 | 887 | 808 |
| Iter merge | 793 | 874 | 934 | 1171 | 872 |
| Quick | 261 | 441 | 228 | 456 | 469 |
| Ins3Quick | 93 | 395 | 93 | 581 | 385 |
| Dual Quick | 1555 | 417 | 1863 | 435 | 522 |
| Parallel Quick | 280 | 495 | 359 | 519 | 487 |
| Iter Quick | 1171 | 1059 | 2455 | 711 | 1054 |
| Counting | 121 | 84 | 63 | 36016 | 74 |
| Negative Counting | 84 | 86 | 66 | 44933 | 80 |
| Bucket | 318 | 348 | 277 | 291 | 321 |
| Radix | 182 | 433 | 233 | 556 | 538 |

Todos os dados, foram obtidos de compilações diferentes e consequentemente terão arrays gerados diferentes. Dito isso, é justificável casos de “outliers” dentro do conteúdo, mas independentemente de não serem sequenciais em alguns casos, ainda demonstram padrões de crescimento.

### Ordenado:

Para arrays já ordenados, algoritmos como Bubble Sort e Selection Sort são notavelmente menos eficientes em comparação com outros métodos, uma vez que continuam realizando diversas comparações e iterações no array, mesmo quando a ordenação já está concluída. O algoritmo Merge Sort recursivo é outro exemplo, pois executa as mesmas operações independentemente de o array estar ordenado ou não.
Em geral, os algoritmos exibem desempenhos semelhantes, mas algumas abordagens se destacam em determinados cenários. Por exemplo, a melhoria do Bubble Sort ao reconhecer que o array já está ordenado é significativa. O algoritmo híbrido de Merge Sort com Insertion Sort demonstrou um bom desempenho devido à eficiência do Insertion Sort em conjuntos de dados parcialmente ordenados. Da mesma forma, a combinação do algoritmo Quick Sort com o Insertion Sort também mostra eficácia em algumas situações, aproveitando as vantagens do Insertion Sort em arrays menores ou parcialmente ordenados. O algoritmo Insertion Sort em si é reconhecido por sua eficiência, especialmente em conjuntos de dados pequenos.

### Inversamente ordenado:

Para arrays inversamente ordenados, algoritmos de complexidade quadrática, como Bubble Sort e Insertion Sort, tendem a ter um desempenho extremamente desfavorável em seus piores casos e, portanto, não são recomendados. Em contraste, as implementações de algoritmos mais eficientes, como Quick Sort, Shell Sort, Merge Sort e Counting Sort, destacam-se geralmente com os melhores tempos de execução nessas situações. Isso se deve ao fato de que esses algoritmos adotam abordagens mais sofisticadas (não são afetados por essa ordenação especifica) para lidar com arrays desordenados, resultando em melhorias significativas no desempenho, especialmente quando os dados estão inversamente ordenados.

### Parcialmente ordenado:

Para arrays parcialmente ordenados, algoritmos como Shell Sort e insertion podem ser eficientes,pois eles se beneficiam da presença de elementos já ordenados. Entretanto, em geral os melhores desempenhos serão os mesmos algoritmos do inversamente ordenado.

### Grandes intervalos de elementos:

Essa configuração em especial não afeta tanto a maioria dos algoritmos mas nos casos de ordenações sem comparações que dependem do intervalo de valores do array é visivel a extrema sensibilidade e consequente explosão do Counting sort.

### Muitas repetições de elementos:

Não fica tão evidente pelo gráfico, entretanto o Quick Sort pode ter um desempenho pior em arrays com muitos elementos repetidos, especialmente se a escolha do pivô não for otimizada para lidar com essa situação. Isso ocorre porque a partição do Quick Sort pode não ser equilibrada, resultando em divisões desiguais do array e levando a uma complexidade de tempo pior do que o esperado. De formas semelhantes o embora o Merge Sort geralmente tenha um desempenho consistente, a etapa de mesclagem pode se tornar menos eficiente se houver muitos elementos repetidos. Isso ocorre porque, durante a mesclagem, os elementos iguais podem ser trocados de posição desnecessariamente, o que aumenta o tempo de execução do algoritmo.

## 6. Conclusões

Após uma análise abrangente considerando o tamanho do vetor ordenado, o tamanho do objeto ordenado e a configuração inicial dos dados, chegamos a várias conclusões importantes sobre o desempenho dos algoritmos de ordenação.

- Não existe algoritmo melhor, o que existe é um problema e cada algoritmo tem uma capacidade diferente e pode se encaixar para resolver esse problema.

- A eficiência dos algoritmos mais complexos vem acompanhado sempre de um preço e uma pergunta que é : vale a pena e podemos pagar ?

- Algoritmos mais complexos, como Merge Sort e Quick Sort, podem oferecer eficiência em termos de tempo de execução em cenários específicos, mas podem exigir mais recursos computacionais, como memória adicional ou poder de processamento.

- Em situações em que o tempo de execução é crítico e os recursos são limitados, algoritmos mais simples, como Insertion Sort ou Selection Sort, podem ser preferíveis devido à sua simplicidade e menor sobrecarga computacional.

- Ao lidar com objetos de tamanho variável, é importante considerar não apenas o tamanho do objeto em si, mas também como esse tamanho afeta a eficiência dos algoritmos de ordenação que operam nesses objetos.

- A configuração inicial dos dados pode ter um impacto significativo no desempenho dos algoritmos de ordenação. Por exemplo, em situações em que os dados estão parcialmente ordenados ou contêm padrões repetidos, certos algoritmos podem se destacar em relação a outros.

- É fundamental realizar uma análise abrangente das características do problema, dos requisitos de desempenho e das características dos dados antes de selecionar um algoritmo de ordenação. Isso garantirá que a solução escolhida seja a mais adequada para o contexto específico.

Portanto, ao escolher um algoritmo de ordenação para uma determinada aplicação, é importante considerar não apenas o tamanho do vetor, mas também o tamanho do objeto e a configuração inicial dos dados.

## 7. Bibliografia


- Projeto de algoritmos – ZIVIANI

- Algoritmos – CORMEN

- Slides de aula – WAGNER MEIRA

- https://www.w3schools.com (07/05/2024