Métodos de ordenação

Aluno: Wilson Gomes Nogueira Junior

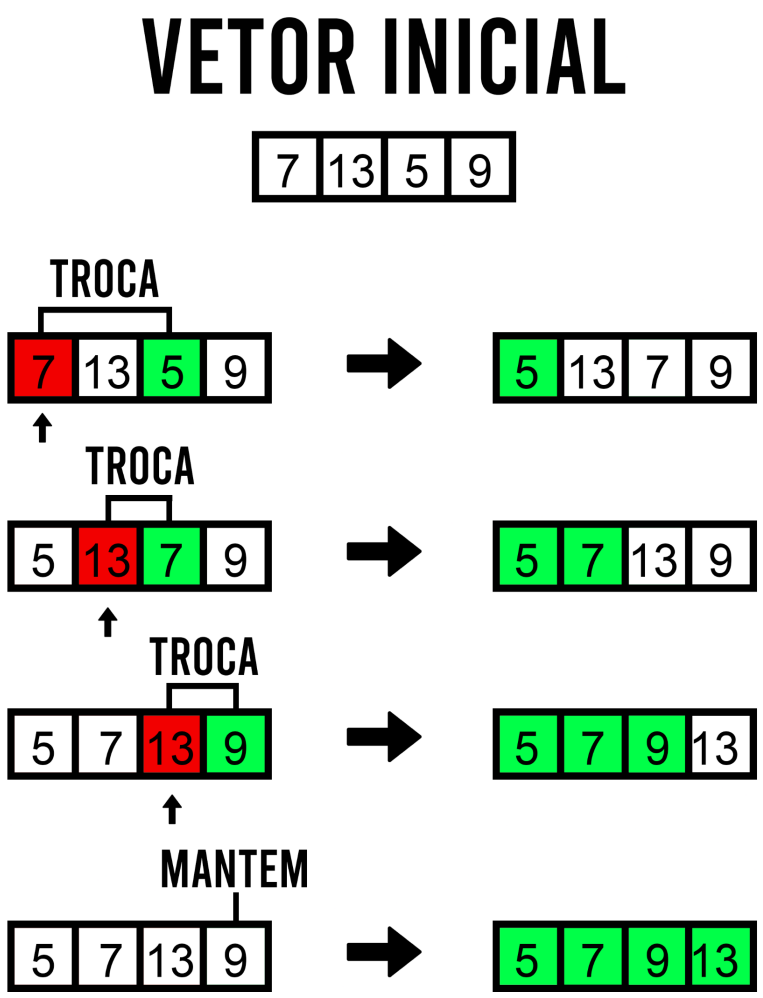
Selection Sort

O algoritmo de ordenação por seleção (Selection sort) funciona da seguinte maneira:

1. Encontre o menor elemento em um array não ordenado e coloque-o na primeira posição.
2. Encontre o próximo menor elemento e coloque-o na segunda posição.
3. Repita esse processo até que todos os elementos estejam ordenados.

Complexidade $O(n^2)$

O Selection Sort é simples e fácil de entender, mas é ineficiente para grandes conjuntos de dados devido à sua complexidade de tempo $O(n^2)$. Existem outros algoritmos de ordenação mais eficientes, como o quicksort e o mergesort.

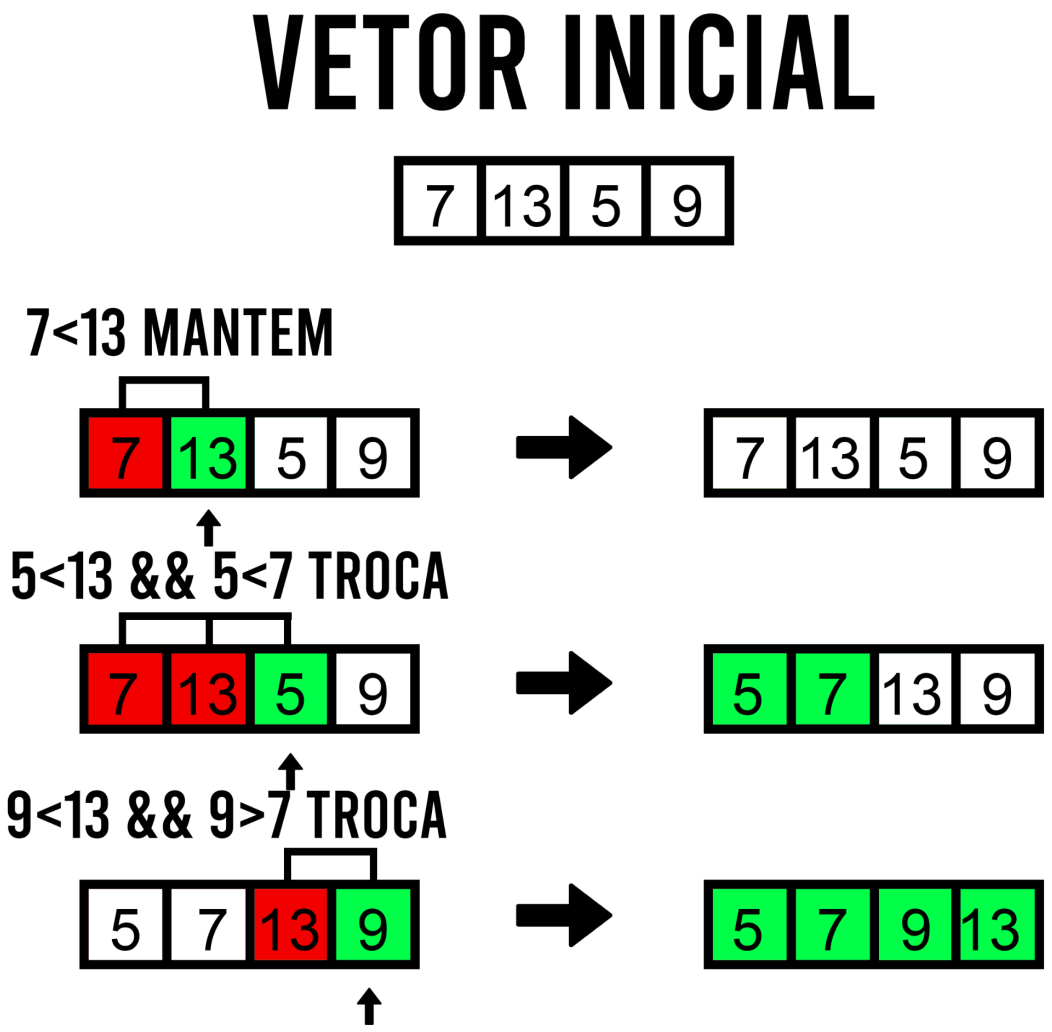


Insertion Sort

O algoritmo de ordenação Insertion Sort funciona comparando cada elemento de uma lista ou array com os elementos à sua esquerda, e os colocando em ordem crescente ou decrescente, dependendo da configuração escolhida. O algoritmo começa comparando o segundo elemento da lista com o primeiro, e, se necessário, os troca de posição. Ele então compara o terceiro elemento com os dois primeiros, e os coloca na posição correta, e assim por diante, até que toda a lista esteja ordenada.
Este algoritmo é mais eficiente com listas pequenas e já quase ordenadas.

Complexidade $O(n^2)$

A complexidade do algoritmo de ordenação por inserção (Insertion Sort) é $O(n^2)$ em sua pior e média caso. No melhor caso, quando o array já está ordenado, a complexidade é O(n) porque o algoritmo não precisa fazer qualquer troca.



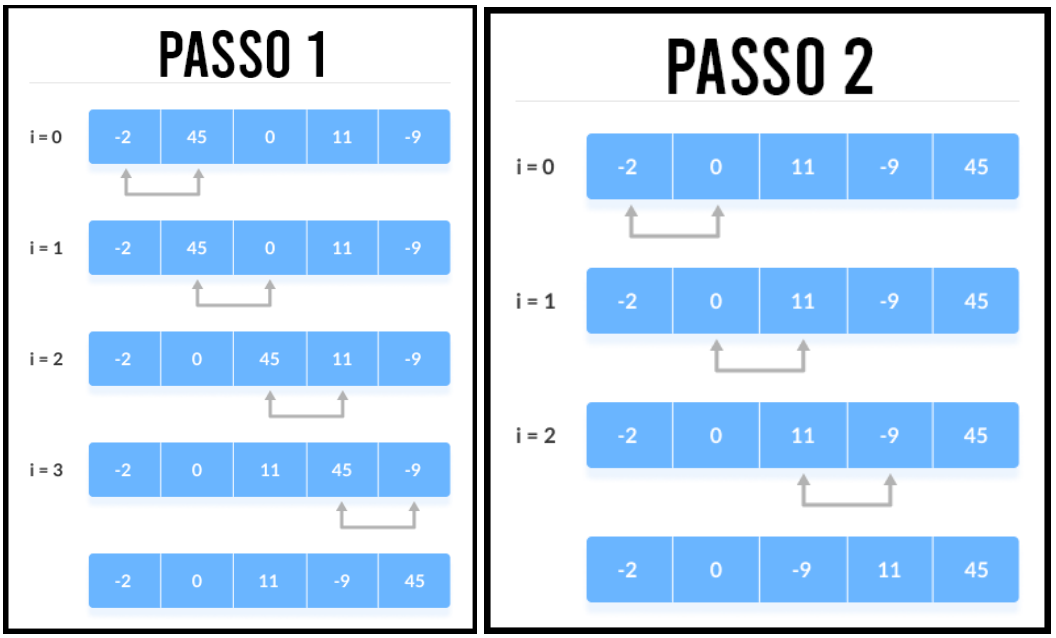
Bubble Sort

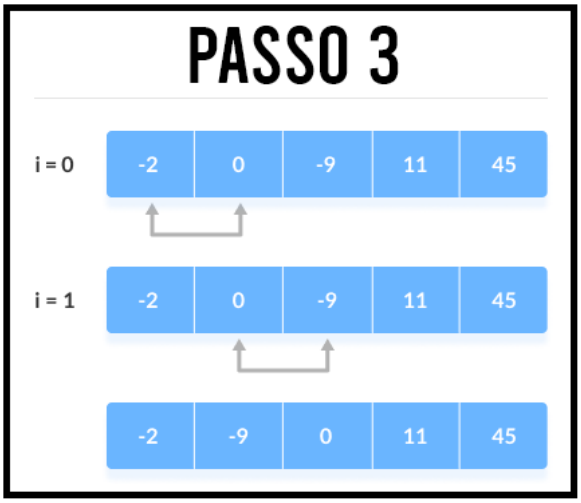
O algoritmo do Bubble Sort funciona da seguinte maneira:

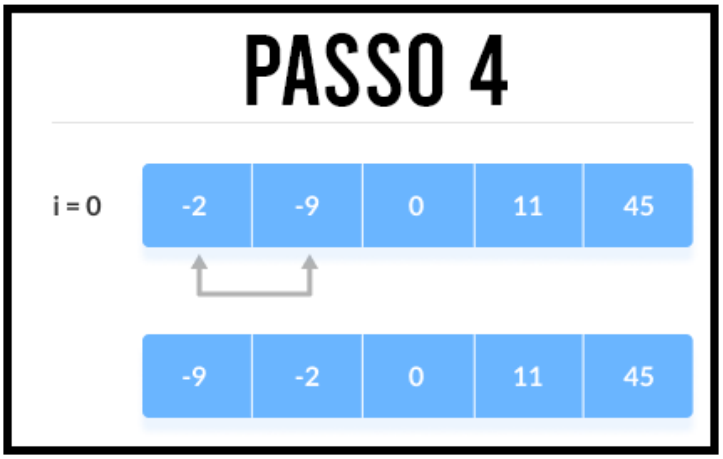
1. Comece no primeiro elemento do array e compare-o com o elemento seguinte.
2. Se o elemento atual for maior do que o elemento seguinte, troque-os de lugar.
3. Avance para o próximo par de elementos e repita o processo de comparação e troca.
4. Repita esse processo até que nenhuma troca seja necessária, o que significa que o array está ordenado.

Complexidade $O(n^2)$

A complexidade do algoritmo de ordenação por bolhas (Bubble Sort) é $O(n^2)$ em seu pior e melhor caso.É importante notar que o Bubble sort é um algoritmo ineficiente e não é recomendado para grandes conjuntos de dados.







Fonte: https://www.programiz.com/dsa/bubble-sort

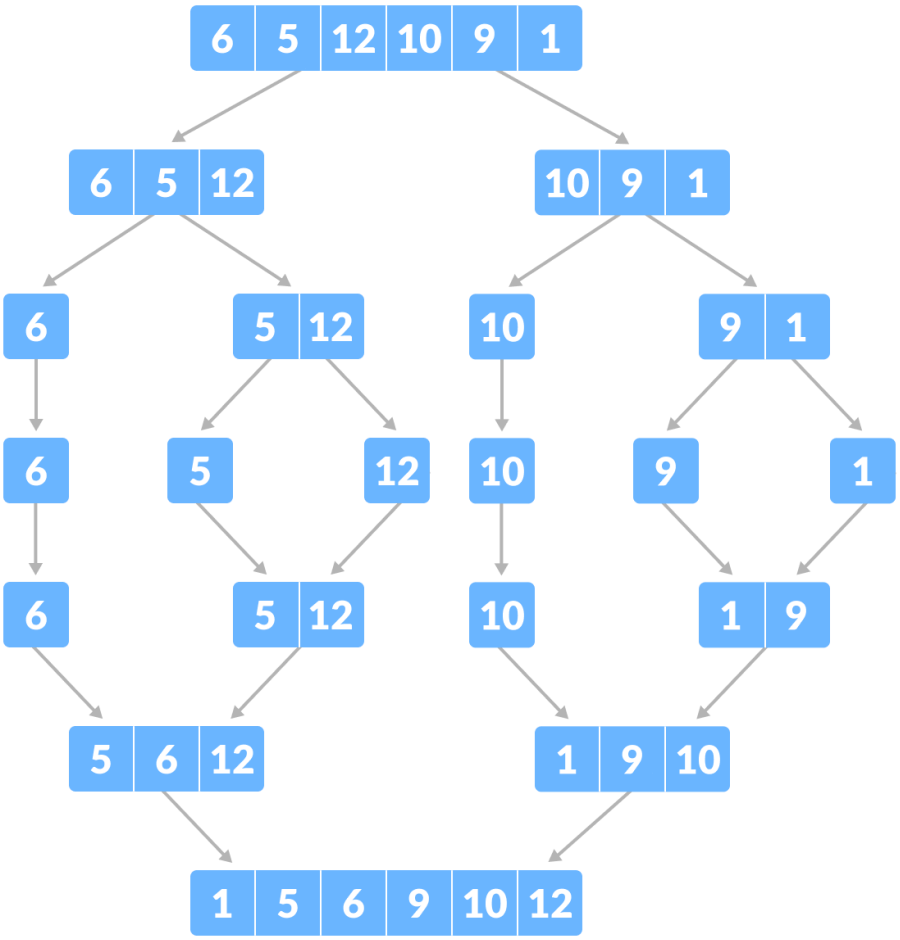
Merge Sort

O algoritmo de ordenação por intercalação (Merge Sort) funciona da seguinte maneira:

1. Divida o array em duas metades.
2. Ordene cada metade recursivamente usando o algoritmo de ordenação por intercalação.
3. Intercale as duas metades ordenadas para formar o array final ordenado.

Complexidade $O(n \log n)$

O algoritmo é baseado em dividir o problema em subproblemas menores e resolvê-los individualmente. Ele é eficiente para grandes conjuntos de dados, devido à sua complexidade de tempo O(n log n).



Fonte: https://www.programiz.com/dsa/merge-sort

Quick Sort

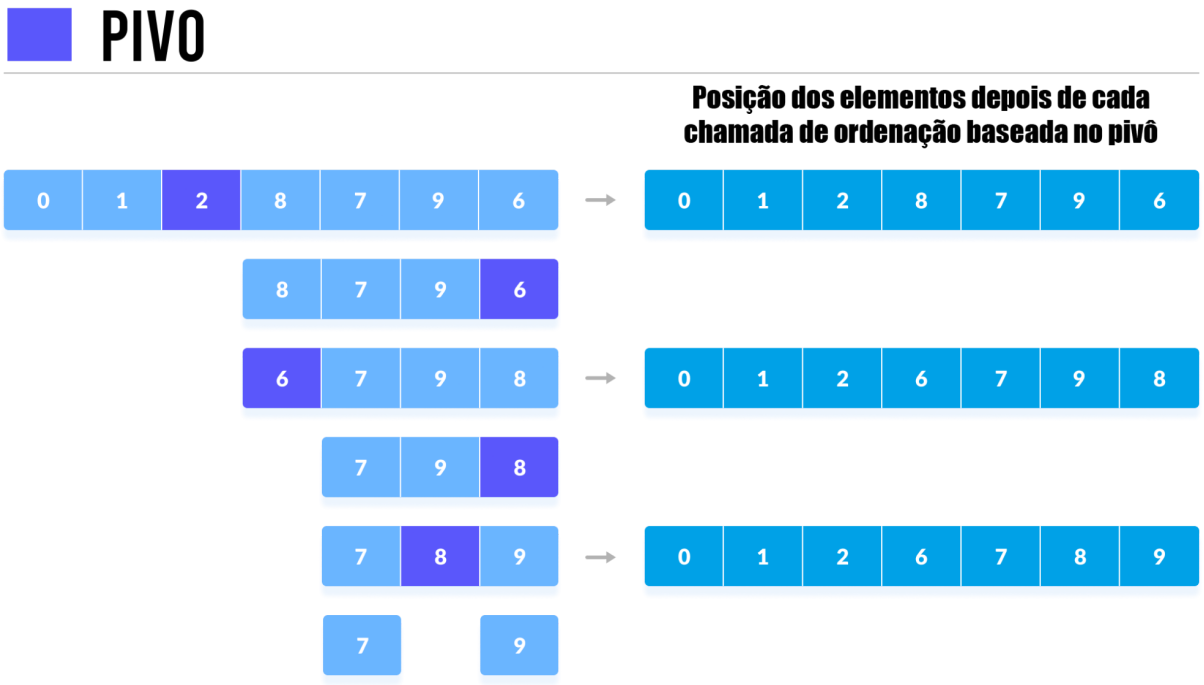
O algoritmo de ordenação rápida (Quick Sort) funciona da seguinte maneira:

1. Escolha um elemento do array como "pivô" (geralmente o elemento do meio).
2. Reorganize os elementos do array de forma que todos os elementos menores que o pivô fiquem à esquerda dele e todos os elementos maiores fiquem à direita dele.
3. Aplique o algoritmo recursivamente às duas sublistas (esquerda e direita) do pivô.

O algoritmo de ordenação rápida é baseado no conceito de dividir para conquistar. Ele seleciona um elemento do array como pivô e reorganiza os elementos do array de acordo com esse pivô.

Complexidade $O(n \log n)$

A complexidade do algoritmo é O(n log n) em sua média caso, e O(n^2) no pior caso, quando ocorre o desbalanceamento entre os subproblemas gerados pela escolha inadequada do pivô.



Fonte: https://www.programiz.com/dsa/quick-sort

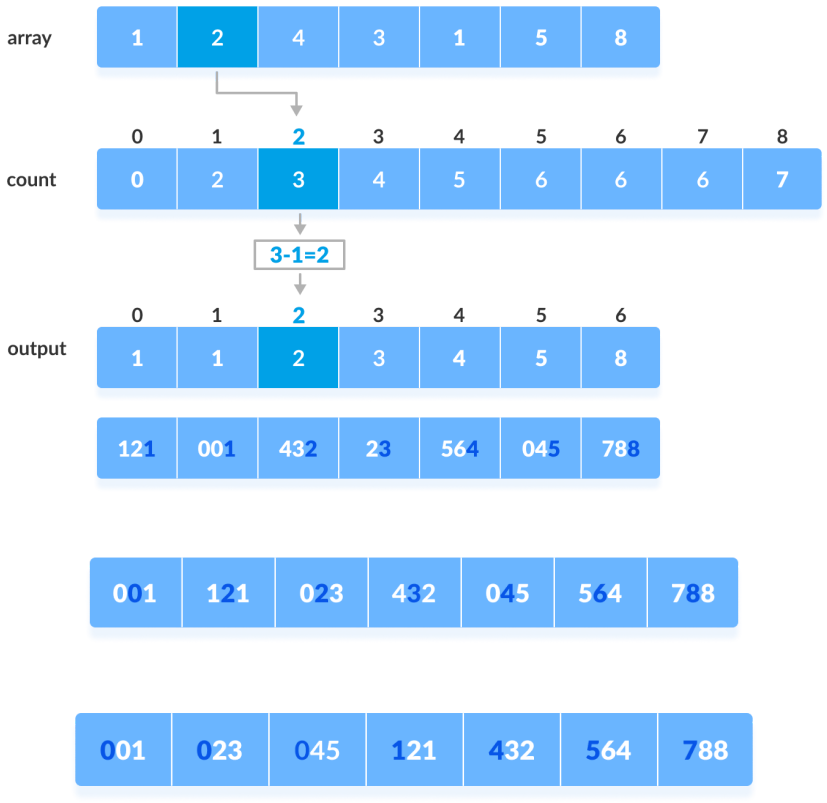
Radix Sort

O algoritmo de ordenação Radix Sort funciona da seguinte maneira:

1. Ache o maior elemento do vetor e pegue a quantidade de dígitos dele, esse valor será o número de dígitos que serão analisados em todos os elementos.
2. Comece ordenando os elementos do array com base no dígito menos significativo (a unidade, por exemplo).
3. Em seguida, ordene os elementos com base no próximo dígito mais significativo (as dezenas, por exemplo).
4. Repita esse processo até que todos os dígitos tenham sido considerados e o array esteja ordenado.

Complexidade $O(n\,k)$

A complexidade do algoritmo de ordenação Radix Sort é O(nk) em seu pior e melhor caso, onde n é o número de elementos no array e k é o número de dígitos dos elementos. Isso significa que o tempo necessário para ordenar um array de tamanho n aumenta linearmente em relação ao número de elementos no array e linearmente em relação ao número de dígitos dos elementos.



Fonte: https://www.programiz.com/dsa/radix-sort

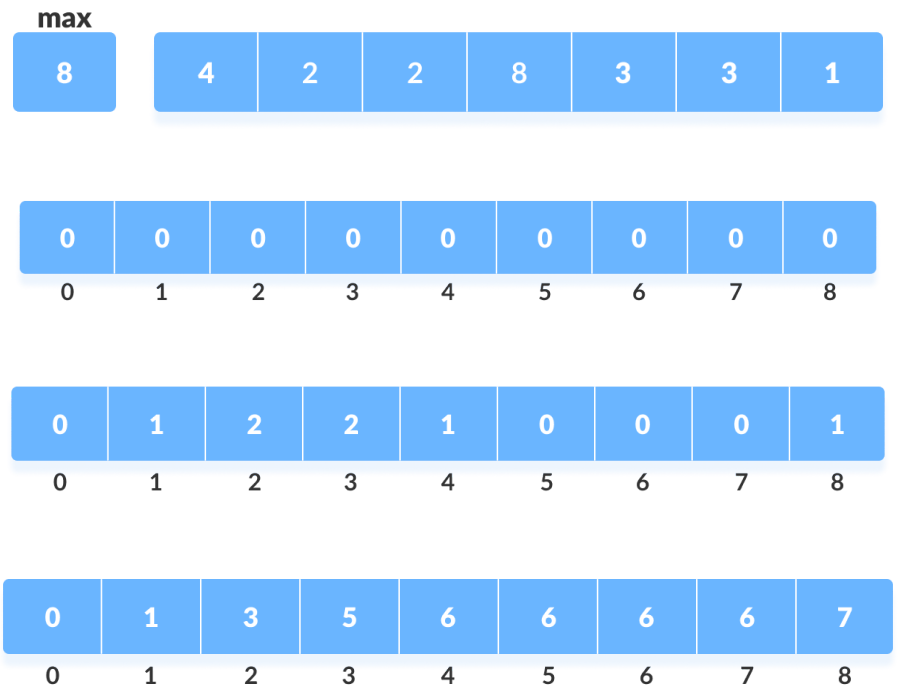
Counting Sort

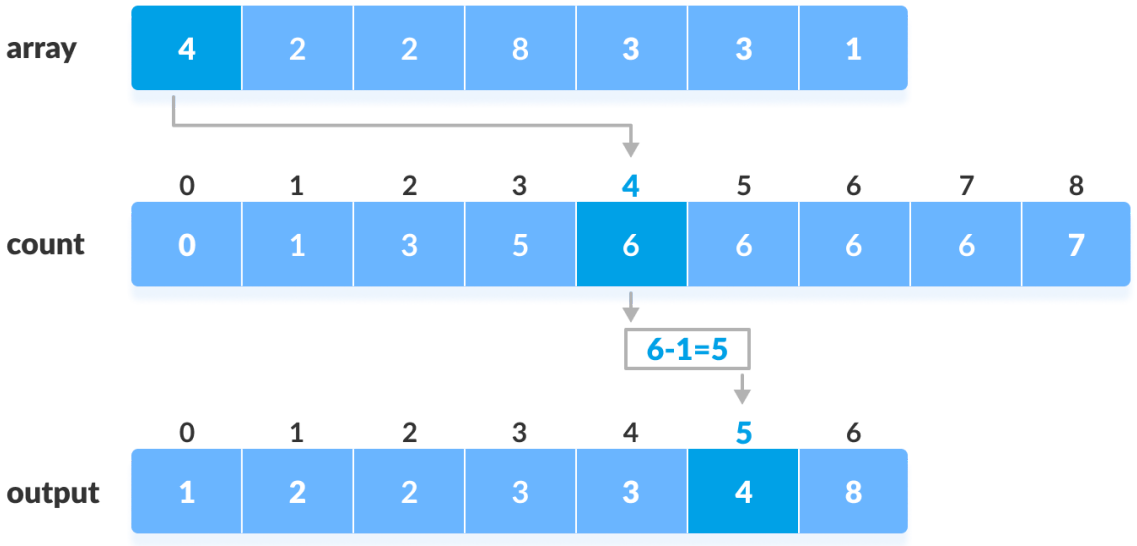
O algoritmo de ordenação Counting Sort funciona da seguinte maneira:

1. Crie um array auxiliar "count" com o tamanho máximo do elemento a ser ordenado.
2. Inicialize o array "count" com zeros.
3. Faça a contagem de cada elemento no array a ser ordenado.
4. Faça a soma acumulativa do array "count".
5. Percorra o array a ser ordenado e coloque cada elemento em sua posição correta no array de saída.

Complexidade $O(n+k)$

A complexidade do algoritmo de ordenação Counting Sort é O(n+k), onde n é o número de elementos no array e k é o intervalo de valores dos elementos. Isso significa que o tempo necessário para ordenar um array de tamanho n é proporcional ao número de elementos no array e ao intervalo de valores dos elementos.





Fonte: https://www.programiz.com/dsa/counting-sort

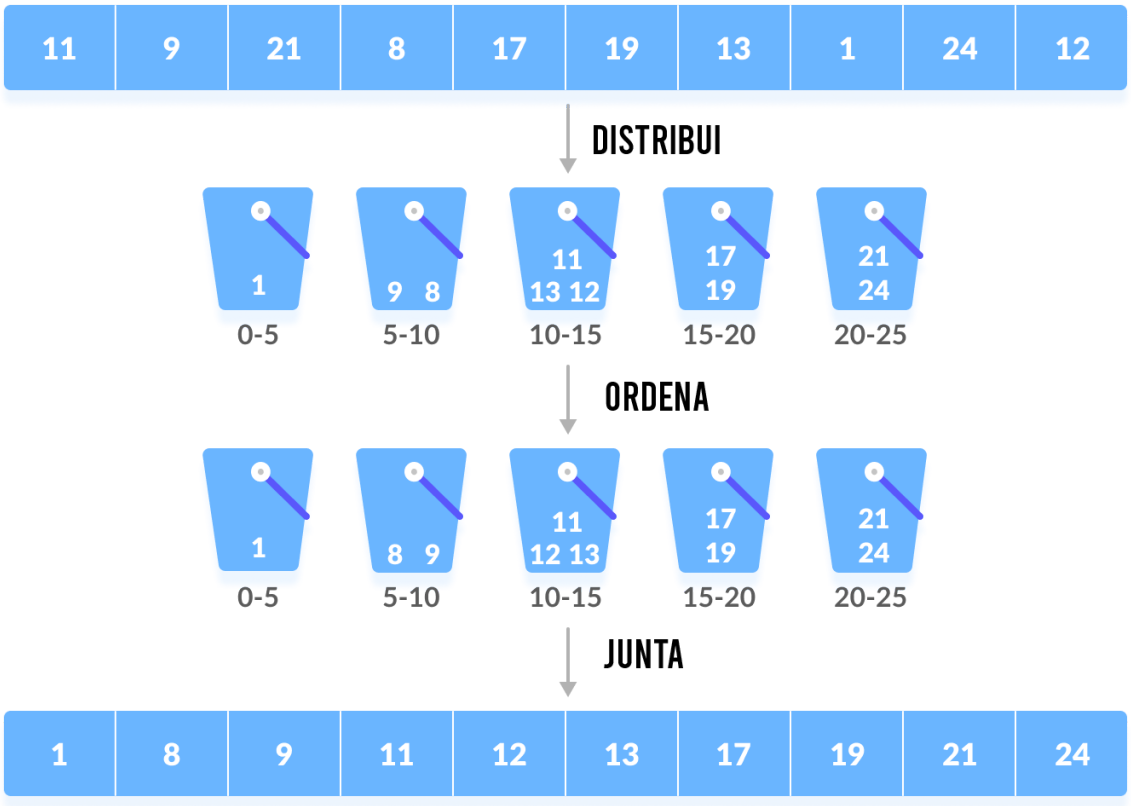
Bucket Sort

O algoritmo de ordenação Bucket Sort funciona da seguinte maneira:

1. Crie uma série de "baldes" para armazenar elementos.
2. Distribua os elementos do array a ser ordenado entre os baldes, de acordo com sua posição relativa ao limite inferior e superior do intervalo de valores dos elementos.
3. Ordena cada baldes individualmente, usando um algoritmo de ordenação interno (como Insertion Sort, por exemplo).
4. Concatene os baldes ordenados para formar o array ordenado.

Complexidade $O(n+k)$

A complexidade do algoritmo de ordenação Bucket Sort é O(n+k) no melhor e no pior caso, onde n é o número de elementos no array e k é o número de baldes.



Fonte: https://www.programiz.com/dsa/bucket-sort

Comparação gráfica dos métodos de ordenação

Comparação feita baseada nos testes exigidos no trabalho, realizando a média de tempo de ordenação de 40 vetores de 8 tamanhos diferentes, começando em 50, e, cada tamanho é o tamanho anterior multiplicado por 5. Logo, temos os seguintes tamanhos de vetores:

50	250	1250	6250	31250	156250	781250	3906250

Validador de ordenação

Para garantir que os algoritmos de ordenação de fato estão funcionando, criei uma função para verificar se os elementos dos vetores estão ordenados. Caso o vetor não esteja ordenado, aparecerá uma mensagem no terminal “Falha na ordenação.”. Todos os algoritmos passaram nesse teste.

```c
int ValidaOrdenacao(int * vetor, int tamanhoVetor)
{
    int i;
    for (i = 0; i < tamanhoVetor - 1; i++)
    {
        if (vetor[i] > vetor[i + 1])
        {
            return 0;
        }
    }
    return 1;
}
```

Marcação de tempo

Para marcar o tempo, a forma mais simples que achei dentro das limitações da linguagem C foi utilizando a biblioteca time.h, utilizando a função clock(), que retorna o horário atual. Criei uma variável ínicio que recebe o horário atual, logo antes de chamar a função que eu queria marcar o tempo. Também criei uma variável fim, que recebia o tempo atual logo após que a função que eu queria marcar o tempo terminava sua execução. Para finalizar, criei uma variável tempoExecucao que recebe a diferença de fim por início. Segue o código

```
// Incia contagem de tempo de execução
clock_t inicio = clock();
funcaoQueQueroSaberOTempo();
// Finaliza contagem de tempo de execução
clock_t fim = clock();
tempoExecucao += (double)(fim - inicio) / CLOCKS_PER_SEC;
```

Máquina utilizada nos testes

A máquina utilizada nos testes possui as seguintes configurações
-Processador: AMD Ryzen 5 5600X 6-Core, 12 Threads, cache 35MB, 3.70 GHz
-Memória RAM: 16Gb, 3200Mhz, DDR4
-Disco rígido/SSD: SSD M.2 Leitura de 560mb/s
-Sistema operacional: Windows 10

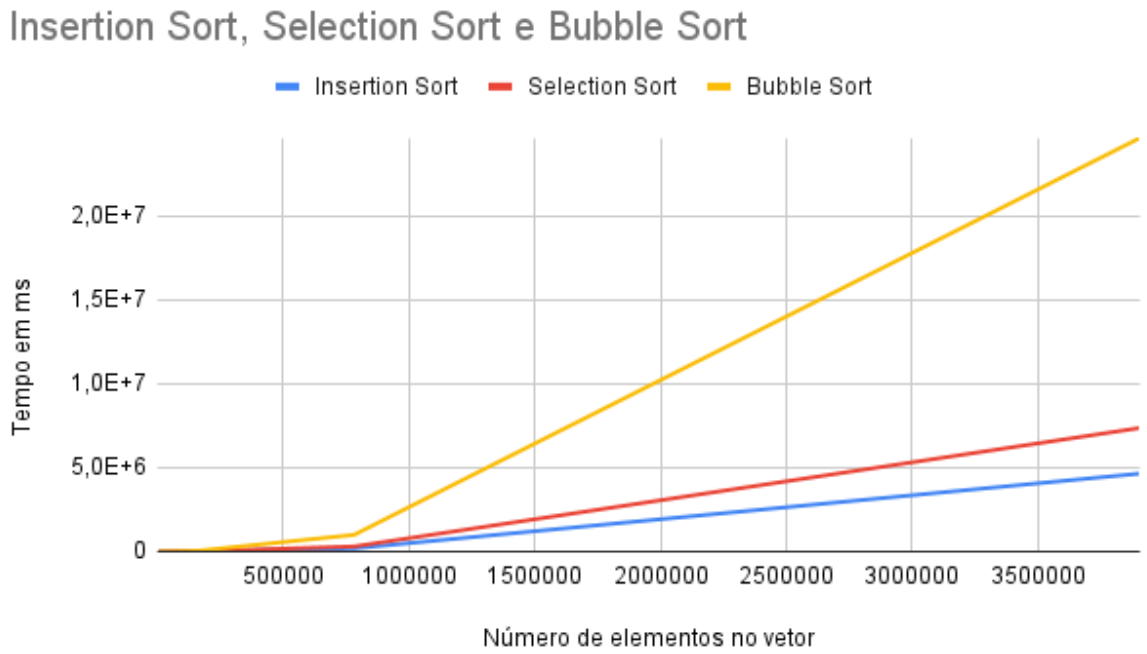
Gerador de testes

Para fazer a criação dos vetores eu utilizei a função GerarVetor, que recebe um vetor e o tamanho dele como parâmetro. Dentro desse método, utilizei a função rand() e a srand() da biblioteca stdlib.h , sendo a função srand para escolher a semente de criação dos números(evitando que sempre fosse criado o mesmo vetor) e a rand para de fato gerar os números para o vetor. E para garantir que sempre seria uma semente diferente, coloquei como parâmetro para srand a função time(NULL), da biblioteca time.h, que sempre irá pegar o horário atual da máquina e passar como parâmetro para a srand, ficando a seguinte expressão srand(time(NULL)).

Limitações da rand()

A função rand irá retornar valores inteiros no intervalo de 0 e RAND_MAX, de forma distribuída entre esse intervalo, ou seja, a chance de retornar um número próximo de 0 é a mesma de retornar um valor perto de RAND_MAX, tornando assim, os elementos dos vetores mais aleatórios possíveis. Entretanto, a constante RAND_MAX é setado por padrão na biblioteca stdlib.h, com o valor 32767. Fui perceber isso apenas no último dia restante para entregar o trabalho, então não tive tempo de tentar contornar essa limitação. Mas mesmo assim tentei bastante, procurei até mesmo o source code da função rand mas não consegui encontrá-la. Infelizmente a linguagem C tem algumas limitações e não tive tempo de tentar corrigi-las. Sendo assim, pode ser que os valores mostrados a seguir nos gráficos e tabelas tenham algum tipo de alteração, como o bucket sort, mas irei tentar explicar o motivo pela qual a alteração ocorreu em cada método que tiver resultados inesperados.

Complexidade $O(n^2)$



Esse gráfico é o baseado nas seguintes amostras coletadas no teste:

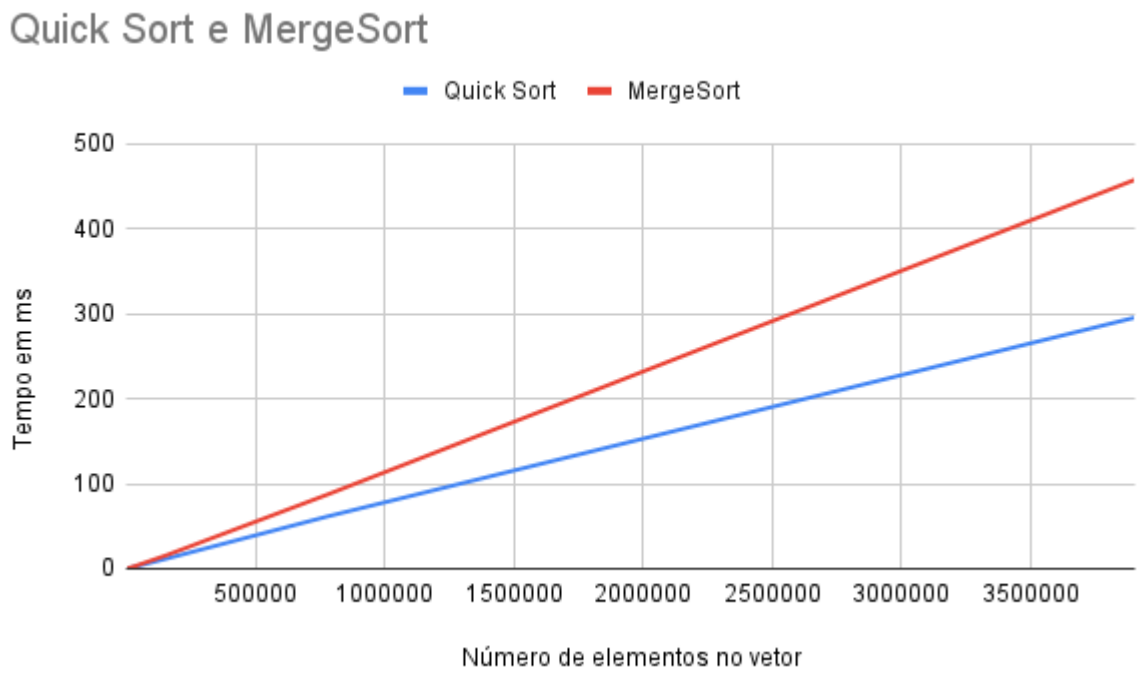
Número de elementos no vetor	50	250	1250	6250	31250	156250	781250	3906250
Insertion Sort	0ms	0ms	0ms	12ms	301ms	7370ms	185594ms	4643518ms
Selection Sort	0ms	0ms	1ms	20ms	489ms	12205ms	298239ms	7362356ms
Bubble Sort	0ms	0ms	0ms	41ms	1072ms	38552ms	990546ms	24649106ms

As células com 0ms são vetores pequenos que foram ordenados de forma tão rápida que não foi possível registrar seu tempo.

Análise

Quando N é um número pequeno é evidente a proximidade de tempo de execução entre os métodos Selection Sort e o Insertion Sort. Entretanto, quando N > 156250, é notável a diferença entre as performances dos métodos de ordenação, sendo que, em caso de vetores com grandes quantidades de elementos, o Insertion Sort terá um desempenho melhor. Já o Bubble Sort obteve o pior desempenho em relação aos outros algoritmos de ordenação de complexidade $O(n^2)$, com o tempo de execução sempre maior a partir de N > 1250, em relação aos outros métodos. Nenhuma alteração causada pela limitação da rand(), já que esses métodos comparam todos os valores, independente se estão próximos ou não.

Complexidade $O(n \log(n))$



Esse gráfico é o baseado nas seguintes amostras coletadas no teste:

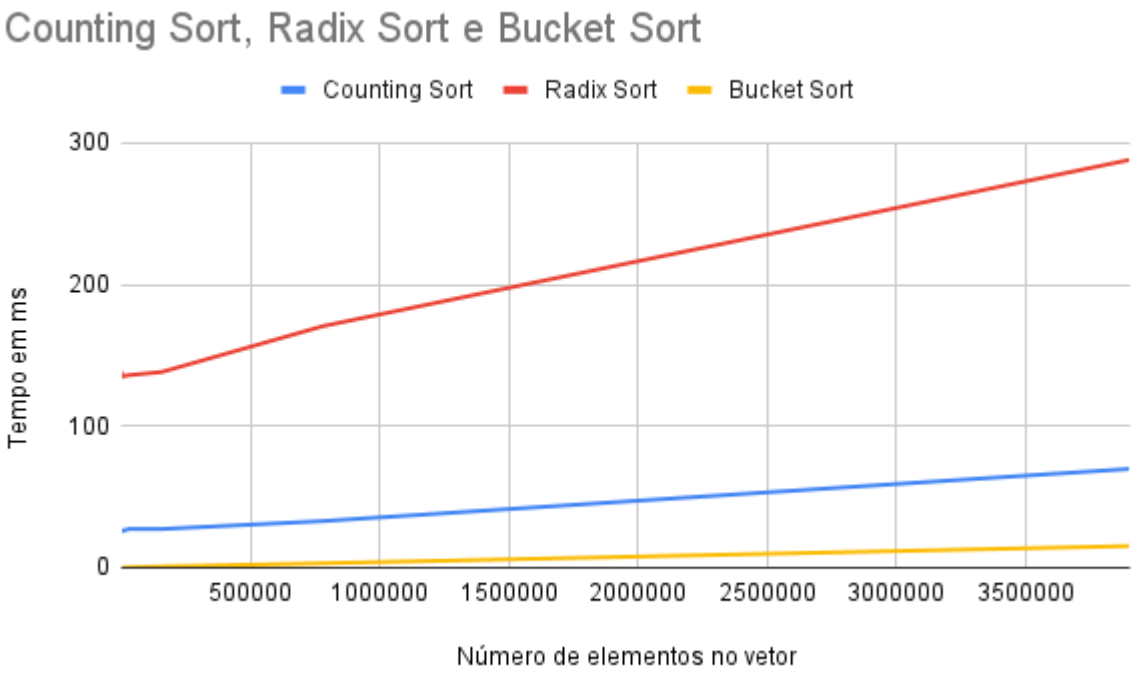
Número de elementos no vetor	50	250	1250	6250	31250	156250	781250	3906250
Quick Sort	0ms	0ms	0,025ms	0,35ms	2,15ms	11,8ms	61,6ms	295,475ms
MergeSort	0ms	0ms	0,125ms	0,55ms	2,725ms	15,825ms	87,45ms	457,675ms

As células com 0ms são vetores pequenos que foram ordenados de forma tão rápida que não foi possível registrar seu tempo.

Análise

A expectativa seria que o Merge Sort ficasse com um desempenho melhor que o Quick Sort, já que em quesito de rapidez ele é melhor. Entretanto, por causa da dificuldade mencionada no tópico “Limitações da rand()”, a quantidade de valores repetidos nos vetores foi muito alta. Como o Merge Sort é um método de ordenação estável, ou seja, ele garante a preservação da ordem original dos elementos que têm o mesmo valor na lista a ser ordenada, ele teve mais trabalho do que o Quick Sort que é instável, que não garante que essa ordem seja mantida. Por esse motivo o Quick Sort teve um melhor desempenho em no teste.

Complexidade *O*(N)



Esse gráfico é o baseado nas seguintes amostras coletadas no teste:

Número de elementos no vetor	50	250	1250	6250	31250	156250	781250	3906250
Counting Sort	26,9ms	26,1ms	26,8ms	26,0ms	27,3ms	27,3ms	32,9ms	69,7ms
Radix Sort	134,4ms	133,2ms	137,2ms	135,2ms	135,975ms	138,125	170,55ms	287,5ms
Bucket Sort	0,05ms	0,05ms	0,05ms	0,05ms	0,3ms	0,85ms	3,1ms	15,5ms

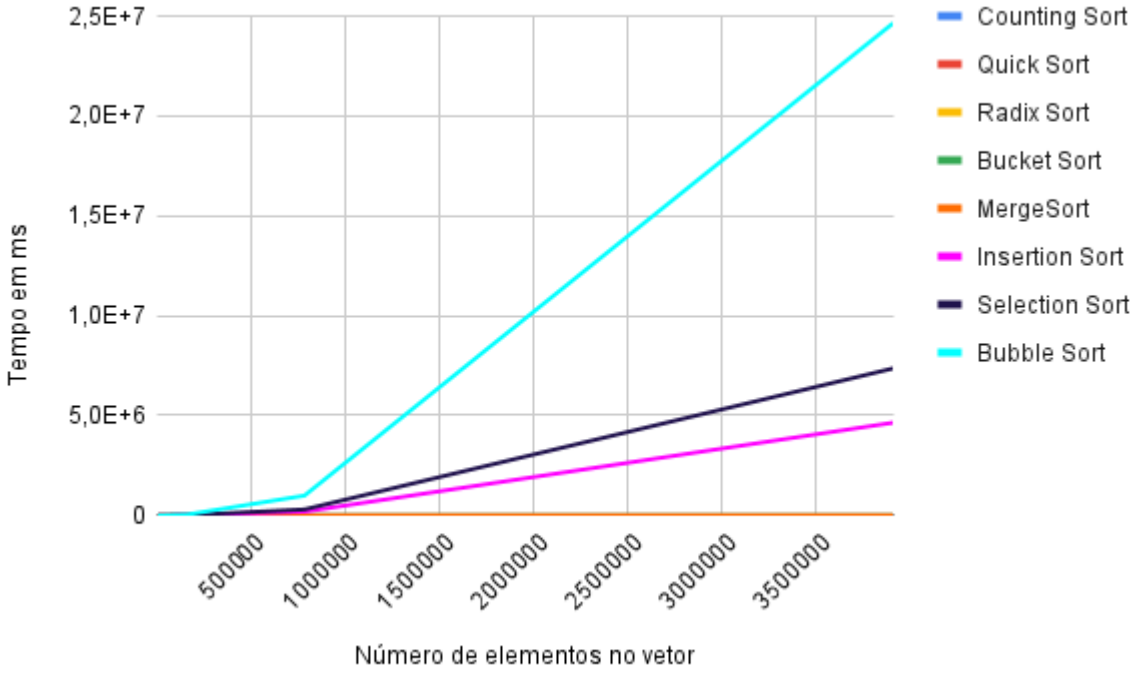
As células com 0ms são vetores pequenos que foram ordenados de forma tão rápida que não foi possível registrar seu tempo.

Análise

A rapidez do Bucket Sort chama muita atenção neste gráfico, e isso tem uma explicação, já que na teoria não era esperado esse desempenho. Novamente, por causa da limitação citada no tópico “Limitações da rand()”, em toda criação de vetor aleatório há uma grande quantidade de números em um mesmo intervalo. Por exemplo, quando estamos ordenando um vetor com 3.906.250 elementos e que contêm apenas número até 32.767, existirão uma grande quantidade de números no intervalo de 1500 < x < 2000. Sendo assim, o número de baldes é muito pequeno, já que apesar de ser um vetor grande, o intervalo de números é muito pequeno, então o número de baldes necessários para colocar todos os elementos é pequeno. Como a complexidade do Bucket Sort é O(N k), onde k é o número de baldes, sempre teremos um valor pequeno multiplicando o N. Por isso esse ótimo desempenho do método. Mas vale lembrar que ele teve um ótimo desempenho nesse contexto, em

outras, possivelmente não seria um método tão bom. Possivelmente o Counting Sort também obteve um bom desempenho devido ao pequeno intervalo de valores contidos no vetor.

Todos os métodos



Esse gráfico é o baseado nas seguintes amostras coletadas no teste:

Número de elementos no vetor	50	250	1250	6250	31250	156250	781250	3906250
Counting Sort	26,925	26,175	26,8	26,025	27,375	27,3	32,9	69,75
Quick Sort	0	0	0,025	0,35	2,15	11,8	61,6	295,475
Radix Sort	134,45	133,225	137,2	135,2	135,975	138,125	170,55	287,975
Bucket Sort	0,05	0,05	0,05	0,05	0,3	0,85	3,1	15,275
MergeSort	0	0	0,125	0,55	2,725	15,825	87,45	457,675
Insertion Sort	0	0	0	12	301	7370	185594	4643518
Selection Sort	0	0	1	20	489	12205	298.239	7362356
Bubble Sort	0	0	0	41	1072	38552	990546	24649106

As células com 0ms são vetores pequenos que foram ordenados de forma tão rápida que não foi possível registrar seu tempo.
Os valores estão todos na unidade de milissegundos.

Análise

Como já era de se esperar, os métodos de ordenação com complexidade $O(N^2)$ tiveram um desempenho péssimo comparados com os outros tipos de algoritmo, que não são nem possíveis de enxergar no gráfico. Para uma análise mais detalhada dos algoritmos de complexidade O(N) e O(n log n), sugiro a leitura dos tópicos anteriores.