Trabalho Prático 1

Métodos de ordenação

João Vitor Soares Santos

Matrícula: 2023002138

20 de Junho de 2024

Introdução

O presente trabalho tem por objetivo exercitar os conhecimentos em algoritmos de ordenação, analisando-os experimentalmente para que se constraste à análise de complexidade teórica vista em sala de aula no decorrer da disciplina de "Estrutura de Dados" , ministrada pelo professor Wagner Meira.

O problema proposto consiste na implementação dos seguintes algoritmos:

- *Bubble Sort*
- *Insertion Sort*
- *Selection Sort*
- *Merge Sort*
- *Quick Sort*
- *Shell Sort*
- *Counting Sort*
- *Bucket Sort*
- *Radix Sort*

A posteriori, a análise dos algoritmos levou em consideração, o tamanho do vetor, o tamanho dos itens em bytes, a configuração inicial do vetor, se ele está ordenado, inversamente ordenado ou não ordenado e também a população de chaves, se elas podem ou não se repetir e seu domínio.

O código criada para fazer as analises experimentais pode ser encontrado no link anexado abaixo:

Sorting Algorithm Analysis

Implementação

Arquitetura do projeto

A arquitetura é definida em trẽs partes principais:

- ./src : O diretório que contém os arquivos fonte
- ./include : O diretório contendo os cabeçalhos necessários
- Makefile: arquivo que contém diretivas usadas para automatizar a compilação

Existem dois comandos principais no arquivo Makefile:

- make all: compila todo o código fonte em uma saída a.out, localizada na pasta ./bin
- make clean: limpa todos os arquivos de saída e objeto nos diretórios ./bin e ./obj

Compilação do projeto

1. Abra o terminal e navegue até a pasta base do projeto.
2. Certifique-se de que o arquivo Makefile está presente na pasta base do projeto.
3. Execute o seguinte comando no terminal para compilar o projeto:

   usuario@pc:~$ make

Programa principal

O programa ao ser executado no terminal precisa ser iniciado com alguns parâmetros:

- -z : o tamanho do array
- -s : a seed geradora de números aleatórios
- -a : o algoritmo a ser executado
- -d: o domínio das chaves
- -i : o estado inicial do array
- -t : o tipo de dados usado no array

Todas as especificações são apresentadas na função *tutorial()*, ela é executada toda vez que parâmetros errados são passados na linha de código. Resumidamente, o programa criado pega os parâmetros apresentados na linha de código, cria um array com o tamanho e tipos especificados, gera as chaves no domínio, define um estado inicial do array e executa o algoritmo de ordenação, tudo de acordo com os parâmetros passados na linha de código, e após isso imprime o tempo de execução do algoritmo de ordenação no cenário solicitado.

Estratégias de Robustez

Como estratégias de robustez, valores não predefinidos não são aceitos como parâmetros, e valores negativos não são aceitos como *size* e *seed,* além disso não é permitido chaves negativas para tipos que não as suportam.

Execução do programa

Após a compilação bem-sucedida, execute o seguinte comando no terminal para rodar o programa:

```
usuario@pc:~$ ./bin/a.out -z 10 - s 3  -a q3i -d rand -i rand -d int64
```

Isso iniciará a execução do programa, além disso os parâmetros passados acima são apenas um exemplo, ainda existindo inúmeras possibilidades.

Configuração usada

Os seguintes código e testes foram feitos nas seguintes configurações:

- Sistema Operacional : Linux (Ubuntu 23.10) 64 bits
- Linguagem de programação implementada: C++
- Compilador utilizado: g++
- Processador : Ryzen 5
- Quantidade de memória RAM : 8 Gigabytes

Tipos de dados

A análise principal levou em conta dois tipos de dados, o tipo de dado primitivo de c++ *long long* que contém 8 bytes, essa levou em conta que mesmo que fosse usada tipos maiores, a análise experimental não seria muito diferente, e em termos práticos itens com muitos bytes são ordenados através de ponteiros.

Aleatoriedade

Todos os vetores gerados para teste foram criados com a mesma *seed*, então a comparações entre os algoritmos foram feitas levando em conta o mesmo vetor. A função *drand48()* da biblioteca *stdlib.h* do gcc *(GNU Compiler Collection)* foi usada para gerar números aleatórios entre zero e um, dessa forma foram criada as funções que populam as chaves no intervalo especificado:

- *initVector1(array, size)*: popula o vetor com chaves aleatórias no intervalo de 0 a size
- *initVector2(array, size)*: popula o vetor com chaves aleatórias no intervalo de -size a size
- *initVector3(array, size)*: popula o vetor com chaves únicas no intervalo de 0 a size

- *initVector4(array, size)*: popula o vetor com chaves únicas no intervalo de -size a size

Domínio das chaves

Todos os testes foram feitos com chaves pertencendo ao seguinte dominio: $D = \{ x \in \mathbb{Z} \mid x >= -size \wedge x <= size \}$

Sendo size o tamanho do vetor sendo testado. Para tal, todos os algoritmos foram implementados assumindo possíveis valores negativos como chaves. Dessa forma, a análise principal levou em conta a função *initVector2()*.

Análise de Complexidade

Tendo em vista os conceitos aprendidos em sala de aula, os algoritmos testados neste trabalho são primordialmente classificados em três classes de acordo com seu comportamento assintótico:

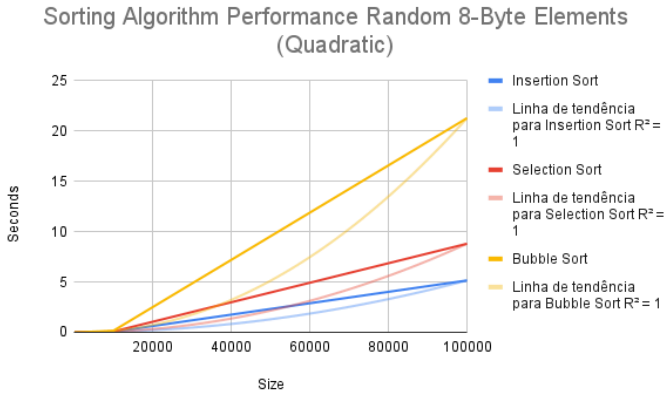
- Quadráticos O(n²) : *Insertion Sort, Bubble Sort* e *Selection Sort*.
- Log-lineares O(n log n) : *Quick Sort, Merge Sort* e *Shell Sort.*
- Lineares ou não-comparativos O(n) : *Counting Sort, Bucket Sort* e *Radix Sort*

Durante o curso, vimos que os algoritmos de ordenação comparativos são Θ(n log n), então os algoritmos de ordenação lineares testados aqui não são comparativos, além disso vale lembrar que embora o *Shell Sort* não possua uma classe assintótica bem definida, seu caso media se aproxima dos log-lineares.

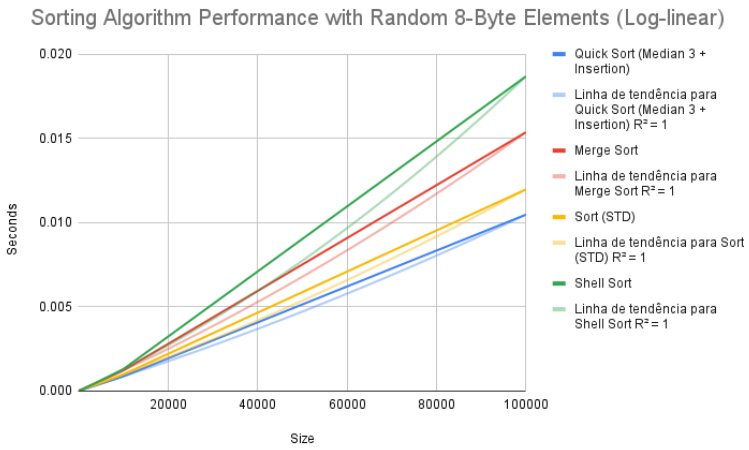
Análise Experimental

Vetores não ordenados

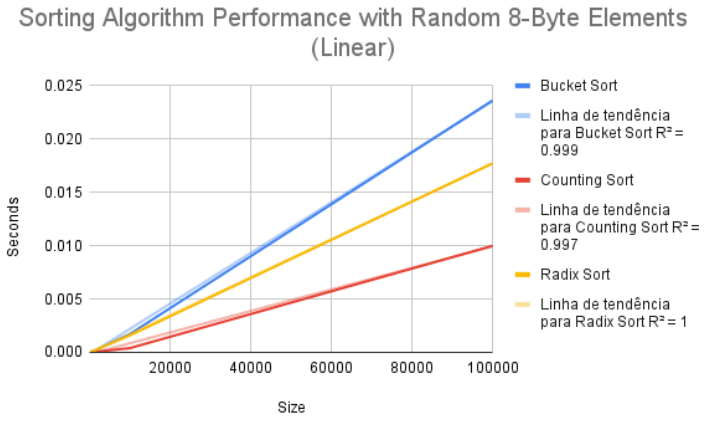
Como primeiro caso, iremos comparar a eficiência dos algoritmos de ordenação em vetores não ordenados.



Na classe quadrática dos algoritmos de ordenação, para arrays não ordenados, em termos de velocidade de execução, *insertion sort* se destaca de seus irmãos *bubble sort* e *selection sort*, pelo fato de ter um comportamento adaptativo, além de ter complexidade assintótica O(n²) para comparações e movimentações de itens no pior caso, e O(n) no melhor caso. Embora o *bubble sort* também seja um algoritmo adaptável, ele fica com a última colocação pelo fato de que suas movimentações de itens são O(n²) enquanto as do *selection sort* são O(n).



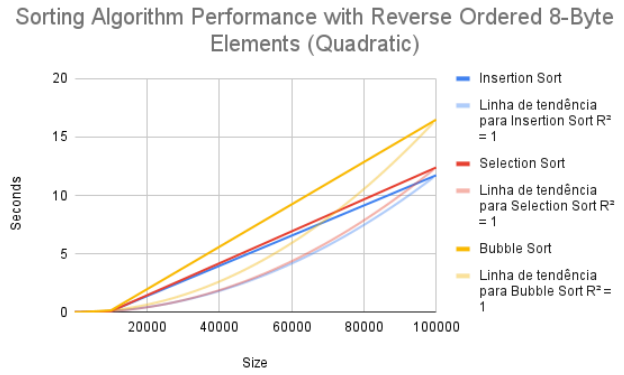
Na classe log-linear dos algoritmos de ordenação, estamos comparando quatro algoritmos, sendo um deles o *std::sort* que pertence ao *gcc (GNU Compiler Collection)*. Além disso, embora o *shell sort* não seja exatamente um algoritmo log-linear (o seu comportamento assintótico varia muito dependendo da sequência de gap) seu caso médio é muito próximo. Dessa forma, fica claro que tanto o *std::sort* quanto o *quick sort,* que não foi implementado com a variação *mediana de 3* com *insertion sort* para vetores menores, levam vantagem em cima do *merge sort,* pelo fato de serem adaptáveis. Outro fato importante é a complexidade espacial do *merge sort*, que é O(n), enquanto seus irmãos são *in-place*(complexidade espacial constante), porém não iremos abordar isso nessa análise. Dito isso, o *merge sort* se destaca por ter o pior caso O(n log n), enquanto o *quick sort* é $O(n^2)$, o que é resolvido pela mediana de 3.



Na classe linear dos algoritmos de ordenação, o *counting sort* se destaca entre os algoritmos de ordenação não-comparativos, ao alcançar um desempenho significativo sacrificando um espaço linear no processo. Da mesma forma, o *bucket sort* utiliza a memória para alcançar desempenho, ao separar seus elementos em *baldes* organizador e juntando-os na etapa final, o algoritmo consegue alcançar velocidade linear quando a quantidade de *baldes* é a igual ao tamanho do array. Diferente de seus dois irmãos, o *radix sort* é o único algoritmo linear in-place testado aqui, sendo O(kn), sendo *k* o tamanho da chave em bits.

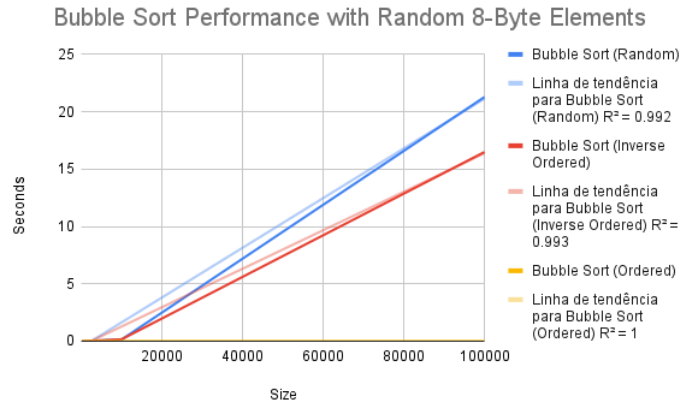
Vetores inversamente ordenados

Em vetores inversamente ordenados alguns algoritmos encontraram seu pior caso, outros entretanto, tiveram resultados surpreendentes

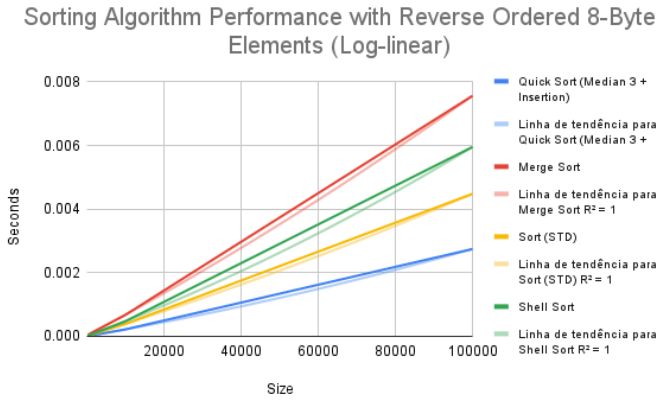


Embora a ordem de classificação dos algoritmos quadráticos não tenha mudado, ao compará-los entre si, entre diferentes estados iniciais do vetor (aleatório, inversamente ordenado e ordenado), conseguimos ver o melhor e o pior caso de cada um.

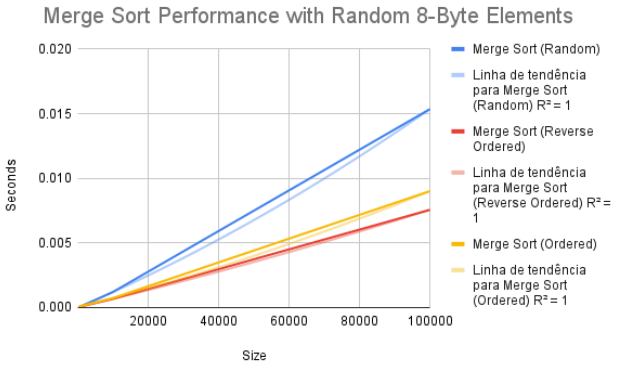
Em vetores inversamente ordenados, o insertion sort encontra seu pior caso, executando $O(n^2)$ comparações e movimentações de itens, e por ser um algoritmo adaptável, como veremos mais tarde, ele tem seu melhor caso em vetores ordenados, executando $O(n)$ comparações e nenhuma movimentação de item.



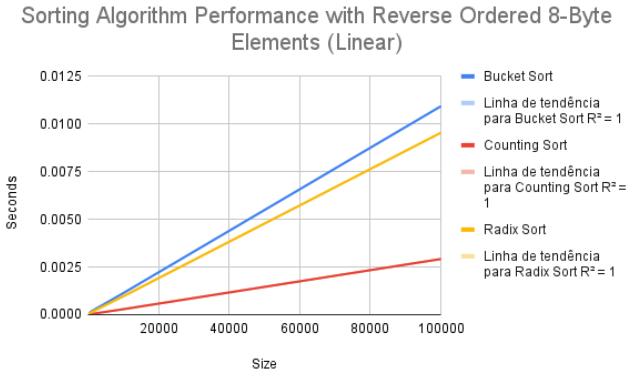
Embora o pior caso do *bubble sort* ocorra em vetores inversamente ordenados, na prática vemos que surpreendentemente ele consegue ser mais rápido do que em vetores aleatórios, mesmo executando $O(n^2)$ movimentações de itens. Isso deve acontecer principalmente pelo efeito *branch prediction*, que é basicamente quando o processador prever os futuros passos do código, pelo fato da comparação entre elementos $antecessor > atual$ sempre ser verdadeira, o processador consegue otimizar esse processo.



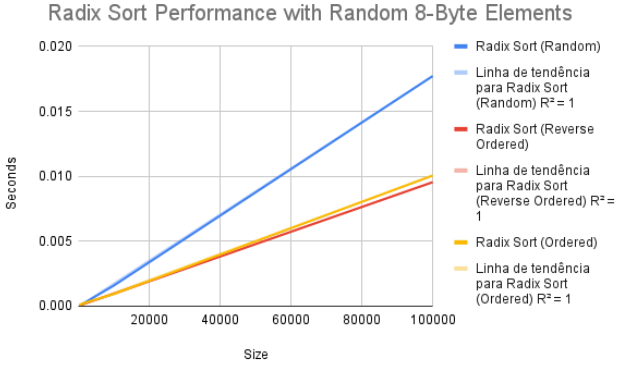
Nos algoritmos log-lineares, enquanto a ordem entre o *quick sort* e o *std::sort* permanece inalterada, em vetores inversamente ordenados, o *shell sort* consegue ultrapassar o *merge sort* em termos de eficiência, e isso se deve ao fato do primeiro ser um algoritmo adaptável, enquanto o segundo não.



Embora o *merge sort* execute o mesmo tanto de operações independente da configuração inicial do array, em vetores ordenados e inversamente ordenados existem menos comparações, o que explica sua melhor eficiência.



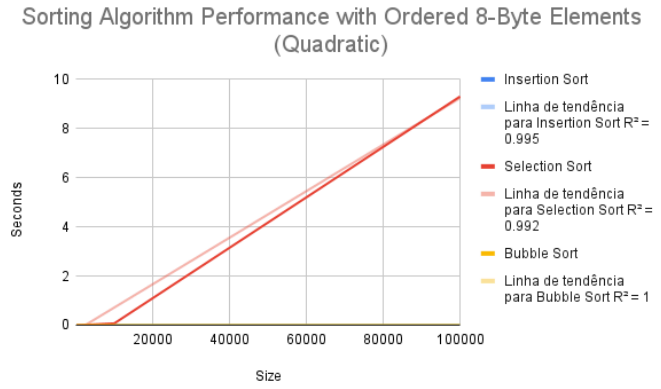
Ainda que o estado inicial do vetor não tenha um efeito aparente nos algoritmos não-comparativos, existe um efeito prático nos princípios de localidade e referência, pelo fato dos elementos de tamanho próximo estarem agrupados, o princípio de localidade espacial consegue tirar muito proveito do array estar ordenado e inversamente ordenado.



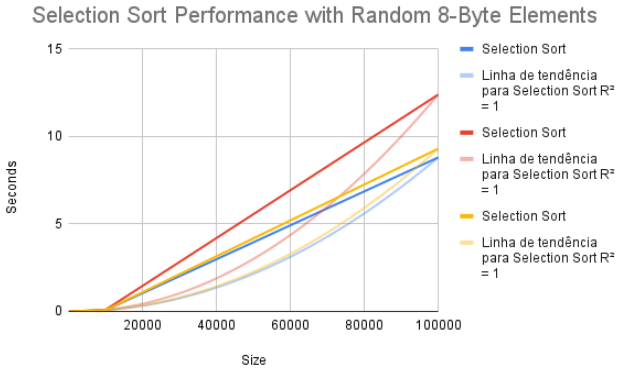
Além disso, mesmo não sendo um algoritmo adaptativo, o radix sort consegue aproximar seu desempenho em arrays inversamente ordenados próximo a eficiência de arrays já ordenados, pelo fato de que n-ésimo maior elemento sempre trocará de posição com o n-ésimo menor elemento, o que o torna muito eficiente, junto também com os efeitos da localidade e referência.

Vetores inversamente ordenados

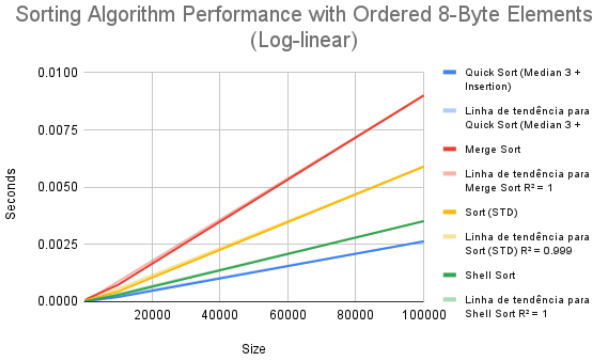
Em vetores ordenados a maioria dos algoritmos encontrou seu melhor caso, pelo fato de serem adaptativos, já outros permaneceram inalterados, pelo fato de não levarem em conta a configuração inicial do array.



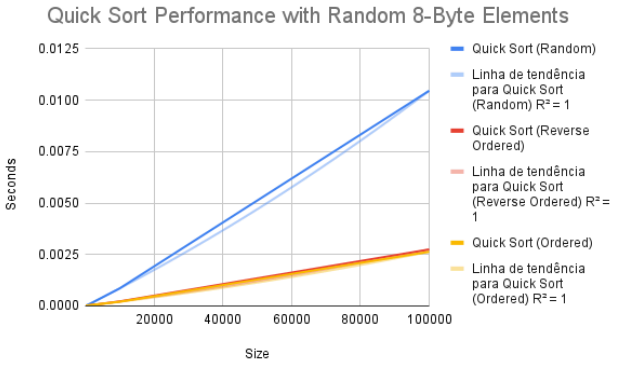
Tanto o *bubble sort* quanto o *insertion sort* encontraram seu melhor caso em vetores ordenados, executando O(n) comparações e nenhuma movimentação de itens, porém esse não foi o caso do *selection sort*, que intrinsecamente, não é adaptativo.



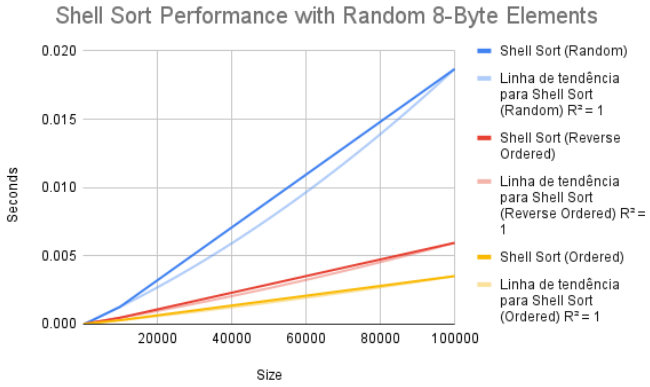
Pelo fato do *selection sort* ser intrinsecamente não adaptativo ele não consegue tirar proveito nenhum do fato de que o array ja esta ordenado, de fato, escolhas melhores para algoritmos quadráticos de ordenação seriam ou o *bubble sort* ou *insertion sort*.



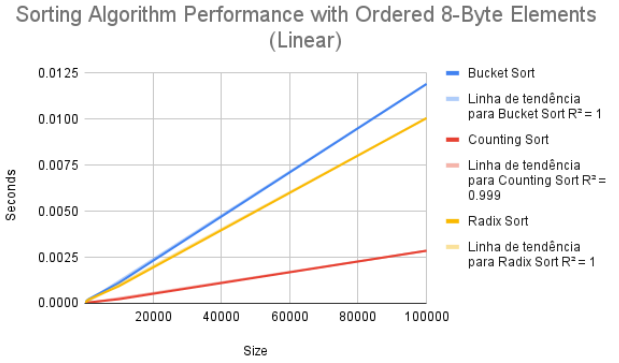
Em vetores ordenados todos os algoritmos log-lineares, com exceção do *merge sort* como vimos na seção anterior, conseguem tirar proveito de vetores ordenados por serem adaptativos, com destaque principal no *quick sort* e no *shell sort*



O *quick sort* consegue alcançar desempenho excelente tanto em vetores inversamente ordenados como em vetores ordenados, no último caso, o quick sort realiza apenas O(n log n) comparações. Em vetores inversamente ordenados, semelhante ao *radix sort*, graças aos princípios de localidade e referência o algoritmo alcança melhor eficiência do que um vetores aleatoriamente ordenado, além do fato de que o n-ésimo maior elemento também é trocado pelo n-ésimo menor elemento.

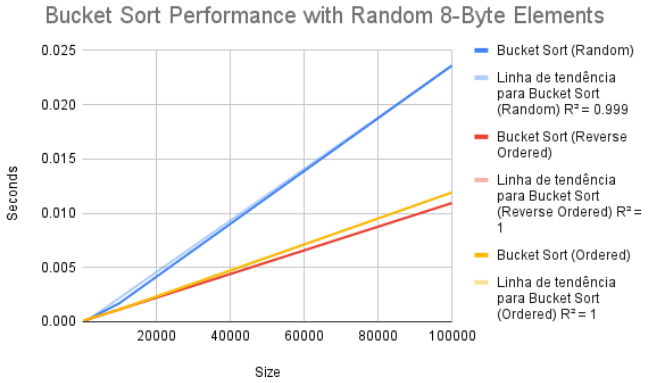


Ademais, o *shell sort* consegue tirar bastante proveito de arrays previamente ordenados, afinal nenhum item é movimentado. Além disso, mesmo sendo uma variação do *insertion sort*, o shell sort lida melhor com o caso de arrays inversamente ordenados por que ele executará menos movimentações, além da sequência escolhida ser outro fator importante nesse caso.



Nos vetores previamente ordenados que o princípio de localidade e referência mais brilha, com destaque nos algoritmos counting sort e bucket sort, que mais usam memória extra para ganhar eficiência.

O *counting sort* se baseia em alocar um vetor auxiliar de tamanho igual ao domínio dos dados, dessa forma tanto em vetores ordenados quanto em inversamente ordenados o principio de localidade é facilitado, por que ao contar quantos elementos x existem, os contadores dos elementos antecessores e sucessores já foram previamente armazenados nos registradores pela arquitetura do computador.



De forma semelhante ao *counting sort,* o *bucket sort* se baseia em alocar vários vetores auxiliares (baldes) cada um tendo um domínio diferente, dessa forma tanto em vetores ordenados quanto em inversamente ordenados o princípio de localidade brilha, por que os baldes possíveis baldes que um elemento pertence já foram previamente salvos nos registradores pelo princípio de localidade, tornando assim o *bucket sort* mais eficiente.

Conclusão

Na classe dos algoritmos de ordenação quadráticos, podemos concluir que tanto o *bubble sort* quanto o *selection sort* devem ter seu uso evitado, o primeiro porque seu caso médio é o pior de todos podendo ter $O(n^2)$ movimentações, enquanto o segundo, por não ser adaptativo, é altamente desfavorável em situações em que o arary já está semi-ordenado, desta forma concluímos que o insertion sort é o mais eficiente entre eles, sendo altamente adaptável e tendo menos comparações e movimentações de itens em casos melhores, entretanto tendo complexidade $O(n^2)$ no pior caso, é recomendável o usar em conjunto com outros algoritmos para arrays menores, como é o caso do *quick sort.*

Em relação aos log-lineares, todos se destacaram em termos de eficiência, porém vale ressaltar alguns pontos importantes, como o fato do *shell sort* ser extremamente volátil em relação a sua sequência de gap, podendo ser muito eficiente em muitos casos e não tão bom em outros. Além disso, o fato do *merge sort* se manter robusto não importando os casos, eles sempre mantém sua complexidade em O(n log n), por bem ou por mal ele não é adaptável, então em caso de arrays variados com itens de magnitude desconhecida existem alternativas melhores, como o campeão dessa categoria, como o *quick sort*. O *quick sort*, se manteve no pódio pelo fato de ser um algoritmo adaptativo, embora tenha pior caso $O(n^2)$ essa situação é resolvida ao uni-lo com a mediana de três e ao insertion sort, outro fato que o destaca é não requerer memória extra como o *merge sort*. Outro ponto a ressaltar é que o *quick sort* desenvolvido superou o *std::sort* em todos os casos testados aqui, algo digno de nota.

Seguindo para os algoritmos lineares os três se destacam em questão de velocidade, porém obviamente sofrem de defeitos que todos os algoritmos não-comparativos carregam: o conhecimento do domínio do array. Dentre eles o radix se destaca o radix, que é extremamente eficiente quando o item possui chaves pequenas, além de não possuir casos extremos de pior caso. Ademais, tanto os algoritmos counting sort e bucket sort se destacam por usar os conceitos de localidade e referência com maestria, sendo extremamente eficientes em todos os casos, porém em contrapartida requerem espaço de memória extra, tendo complexidade de espaço O(n).

Sendo assim, por meio da resolução deste trabalho, foi possível praticar os conceitos relacionados a algoritmos de ordenação e análise de comportamento assintótico, tendo em vista o

Bibliografia

Quicksort. In *Wikipedia, The Free Encyclopedia*. Disponivel em https://en.wikipedia.org/wiki/Quicksort

Radix Sortt. In *Wikipedia, The Free Encyclopedia*. Disponivel em https://pt.wikipedia.org/wiki/Radix_sort

Bucket Sortt. In *Wikipedia, The Free Encyclopedia*. Disponivel em https://pt.wikipedia.org/wiki/Bucket_sort