Documentação do Trabalho Prático 2 da disciplina Estrutura de Dados

Ariel Augusto dos Santos

[1]Departamento de Ciência da Computação – Universidade Federal de Minas Gerais (UFMG)

Belo Horizonte – MG – Brasil

ariel.santos@dcc.ufmg.br

INTRODUÇÃO

Esta documentação tem como objetivo descrever e analisar os algoritmos de ordenação implementados para este Trabalho Prático, nomeadamente:

- InsertionSort;
- MergeSort;
- QuickSort;
- e CycleSort.

Na seção 3 cada implementação será apresentada, explicitando sua base teórica, custos e complexidade.

COMPILAÇÃO E EXECUÇÃO

O projeto segue o arquivo `projeto com makefile.zip` padrão provido, com a adição de uma pasta `/test` e um arquivo `CMakeLists.txt` que contém pontos de entrada para *targets* do **CMake** para o benchmark e teste das funções. O projeto **CMake** depende da biblioteca *Google Benchmark* [1] estar localizada na pasta `/benchmark` para compilação. Como não são estritamente necessários ou especificados por este Trabalho, o projeto **CMake** e seus executáveis são entregues sem nenhuma garantia.

O programa principal, compilado por **make** (`make all`), deve ser executado conforme especificado na definição do trabalho: `./bin/run.out [arquivo] [nº de linhas]`.

O executável de testes, gerado pelo CMake, pode ser invocado por `./cmake-build-debug/TP2Test [arquivo] [nº de linhas]` e não retorna nenhum output, apenas um *status* de retorno indicando se alguma *assertion* de teste foi violada.

O executável de benchmark `./cmake-build-debug/TP2Benchmark` deve ser utilizado de acordo com a documentação do projeto *Google Benchmark*.

IMPLEMENTAÇÃO

Para facilitar o desenvolvimento dos algoritmos de ordenação e a manipulação das listas de tuplas `(Nome da base, Distância)`, foram criadas as estruturas `Array` e `BaseDistância`, onde `Array` pode conter N `BaseDistancia`s e acessá-las em ordem arbitrária através de índices (através de um operador `[]` sobrecarregado). Outro método muito utilizado no projeto é `Array::view(inicio, fim)`, que retorna um novo `Array` que compartilha uma região da memória de `BaseDistancia` do `Array` original a partir de `inicio` até `fim` com o objetivo de simplificar a passagem de argumentos entre funções. A classe `BaseDistância` é convertida implicitamente para o seu valor de distância, permitindo que o código das funções de ordenação tome proveito dos operadores já existentes para tipos triviais numéricos, e possui o operador `<<` definido para impressão dos resultados da ordenação, assim como `Array`.

Após encontrar alguns problemas envolvendo *underflow* do tipo utilizado para indexação dos `Array`s, também é utilizado `std::ptrdiff_t`. Como não é deixado claro na definição do Trabalho a faixa de valores que a distância das bases pode tomar, o tipo utilizado para armazenament é `long long`.

ALGORITMOS

Nesta seção analisamos teoricamente os algoritmos de ordenação implementados para o Trabalho.

INSERTION

O mais simples dos algoritmos implementados, simplesmente *insere* cada elemento em sua posição correta, ordenando a lista de `0` a `N` de maneira crescente. Para isso são necessários dois laços e O(1) memória auxiliar.

Apesar de possuir complexidade de tempo $O(N^2)$ nos piores e médios casos, ele performa bem em listas já ordenadas, com o melhor caso $O(N)$. É uma ordenação estável.

MERGE

O mergesort é um algoritmo mais eficiente de ordenação, baseado em comparações e o método dividir-e-conquistar e inventado em 1945 por John von Neumann. O método consiste de duas partes: subdividir recursivamente a lista de elementos em duas partes e então reagrupar as listas já ordenadas. Como listas de tamanho 1 são trivialmente ordenadas, a complexidade do algoritmo reside no método para mesclar (portanto o nome, *merge* em inglês) duas listas ordenadas. A implementação do algoritmo neste Trabalho foi baseada em uma aproximação *top-down* e requer $O(n)$ memória auxiliar para armazenar o novo `Array` resultante de cada mescla.

É possível analisar o algoritmo a partir da relação de recorrência:

$$T(n) = 1, \text{para } n = 1$$

$$T(n) = T(\lfloor \frac{n}{2} \rfloor) + T(\lceil \frac{n}{2} \rceil) + \theta(n), \text{se } n > 1$$

que nos dá um limite assintótico $T(n) = \Theta(n \log(n))$ para o pior e médio caso. Como o algoritmo não tira proveito de listas já ordenadas, seu melhor caso tem o mesmo limite inferior, $T(n) = \Omega(n \log(n))$.

QUICKSORT

QUICKSORT “CRLS”

O quicksort é outro algoritmo baseado em dividir-e-conquistar, porém difere do mergesort pois utiliza o particionamento para subdividir a lista de elementos de acordo com um valor pivô. Após a escolha do pivô entrada é dividida em duas partições com os elementos menores e maiores que ele, e então cada partição é ordenada recursivamente. Como o particionamento pode ser feito *in-place* e cada recursão necessita apenas armazenar um pointeiro e o tamanho da sua partição do `Array` sob a qual está operando ele requer $O(\log(n))$ memória adicional no caso médio e $O(n)$ no pior caso, porém em ambos os casos as constantes envolvidas são relativamente pequenas já que armazenamos apenas ponteiros e índices.

Apesar de existirem implementações específicas do quicksort que são estáveis, nenhuma das duas neste Trabalho é. Para garantir a estabilidade do algoritmo seria necessário a utilização de memória auxiliar $O(n)$ em todos os casos, ele não iria operar *in-place*, e seria necessário a implementação de uma lista encadeada.

O quicksort é sensível à ordenação inicial das entradas, além do algoritmo para escolha dos pivôs. Podemos representar o pior caso, onde criamos repetidamente partições de tamanho $n - 1$, através da seguinte relação de recorrência:

$$T(n) = T(n-1) + n, \text{que possui forma fechada } T(n) = \frac{1}{2}n(n-1) = O(n^2)$$

No melhor caso, nossas partições são perfeitamente balanceadas em todas as recursões:

$$T(n) = T(\lfloor \frac{n}{2} \rfloor) + T(\lceil \frac{n}{2} \rceil) + n$$

e portanto, pelo teorema mestre: $T(n) \sim n \log(n)$ correspondendo também à performance média sob entradas aleatórias, já que aí podemos considerar que *em média* cada partição também terá em torno de $\frac{n}{2}$ elementos.

A implementação do quicksort simples neste trabalho segue à primeira dada no livro *"Introduction to Algorithms"*[2] de *Thomas H. Cormen*, *Charles E. Leiserson*, *Ronald L. Rivest*, e *Clifford Stein* para o algoritmo, escolhendo simplesmente o último elemento como pivô e realizando duas chamadas recursivas no final da função.

QUICKSORT UTILIZANDO PIVOT ADAPTATIVO

Como a escolha do pivô tem grande impacto sobre a performance do quicksort, definindo o tamanho da partição atual e subsequentes e determinando as constantes em $O(n \log(n))$, a alteração realizada para o trabalho foca na escolha deste valor.

A função `sorting::quick_med3(Array&)` faz uma chamada a `prepare(Array&)` antes de selecionar o pivô a partir do último elemento como o quicksort anterior. Esta função é baseada no *paper "Engineering a Sort Function"*[3] que utiliza a mediana de amostras aleatórias como a mediana de regra três e a nona de John W. Tukey para estimar a mediana dos elementos da entrada e então deposita o elemento escolhido na última posição da lista, além de ordenar relativamente o restante das amostras.

Os valores de n para os quais vale a pena (isto é, onde o custo é proporcionalmente menor que a $\%$ de aceleração no tempo) calcular a mediana de regra três e a nona de John W. Tukey foram retirados diretamente do *paper*, que determinou os valores experimentalmente. Os efeitos mensurados no tempo de execução do quicksort modificado serão discutidos na seção 5.2.

CYCLE

O algoritmo extra avulso escolhido para implementação foi o cyclesort, descrito em 1990 no *paper "Cycle-Sort: A Linear Sorting Method"*[4]. Em específico, foi implementada uma versão recursiva modificada do algoritmo `special_cycle_sort`, que opera *in-place* porém ao invés da complexidade $O(n)$ reivindicada no *paper*, é $\Theta(n^2)$ em todos os casos. Isso se dá porque, na pesquisa original, os dados aderem certas restrições como a cardinalidade das chaves dos elementos e não-ocorrência de repetições.

Apesar da complexidade $\Theta(n^2)$, o algoritmo é interessante pela forma como trata a sequência de permutações necessárias para ordenação da entrada como uma fatorização em ciclos, que são ordenados separadamente. Dessa forma é garantido que serão realizadas ao máximo n trocas, o mínimo possível para um algoritmo de ordenação *in-place*.

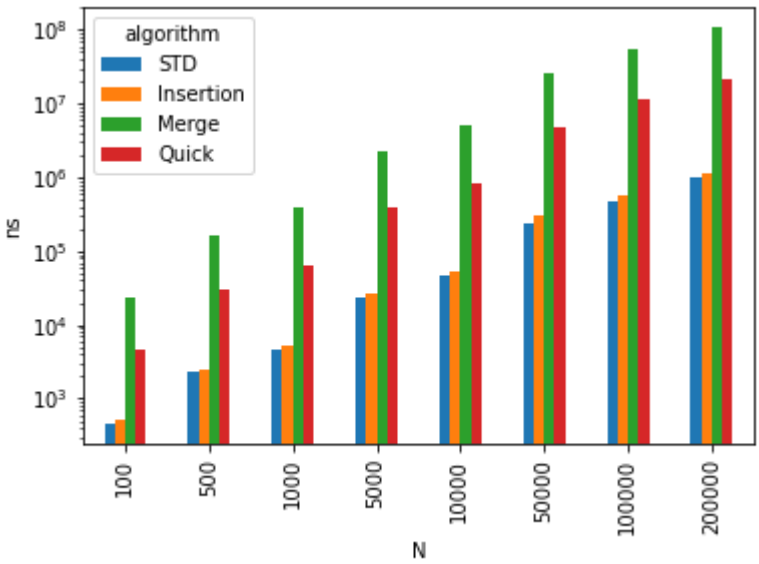
Possíveis aplicações são: coletores de memória, compiladores, ambientes de execução em que escritas à memória são muito custosas, e o jogo *Mastermind* (Senha no Brasil).

BENCHMARKS

Observação: por razões que o autor não foi capaz de determinar, os resultados empíricos obtidos através da biblioteca *Google Benchmark* vão contra todas as expectativas baseadas na complexidade assintótica teórica dos algoritmos. Possíveis causas, consideradas mas não confirmadas, são: *bias* nas listas de entradas, otimizações do compilador, efeitos de cache (frio vs quente), erros de programação, etc. Por esta razão a função `std::sort` também foi inclusa nos benchmarks (como "STD"), para fornecer um baseline "profissional".

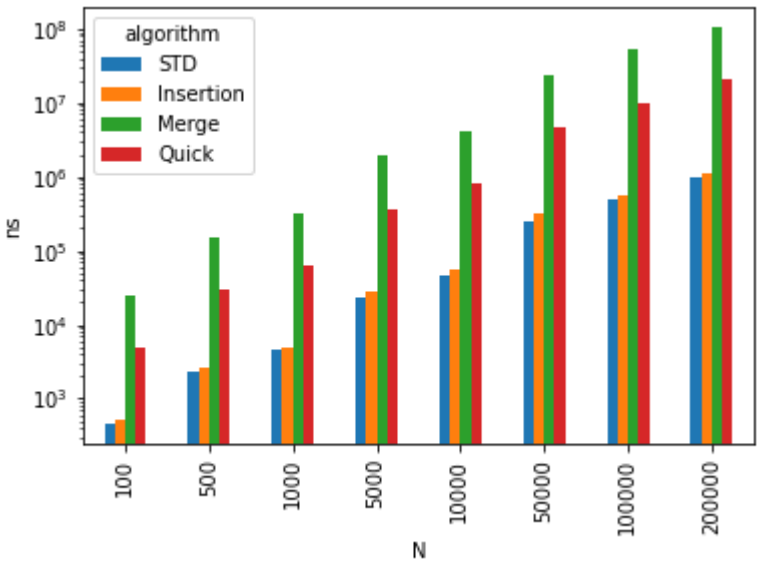
ALGORITMOS PRINCIPAIS VS DADOS ALEATÓRIOS

Em escala logarítmica:



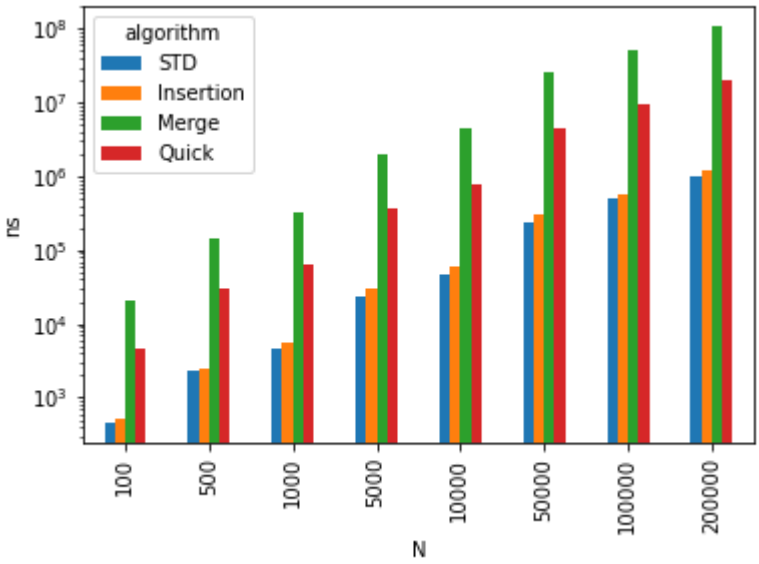
ALGORITMOS PRINCIPAIS VS DADOS CRESCENTES

Em escala logarítmica:



ALGORITMOS PRINCIPAIS VS DADOS DECRESCENTES

Em escala logarítmica:

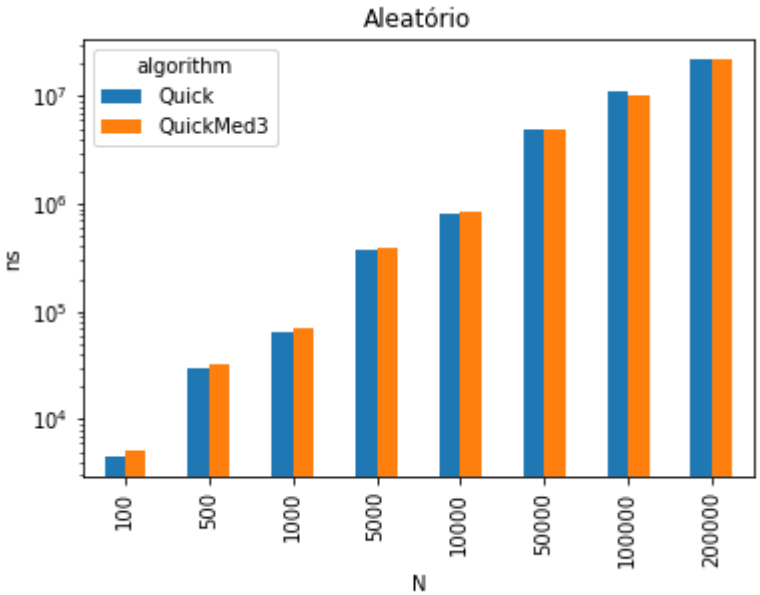


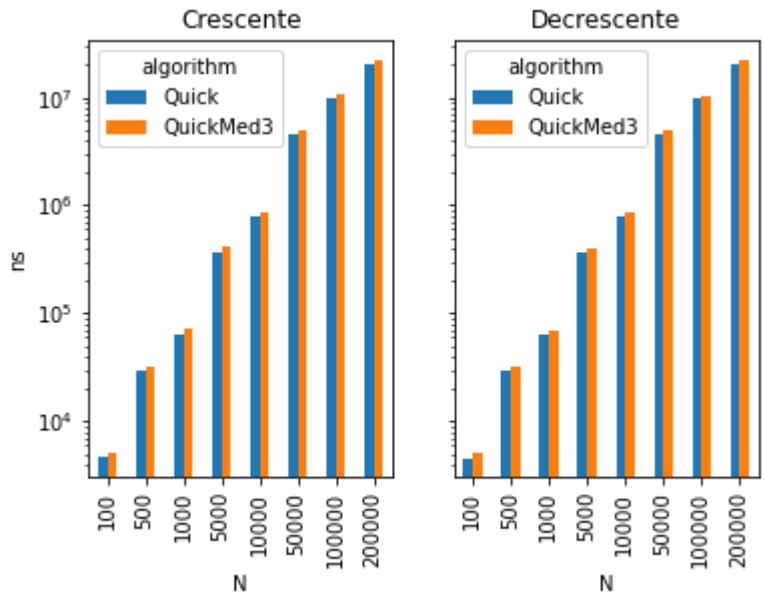
ANÁLISE

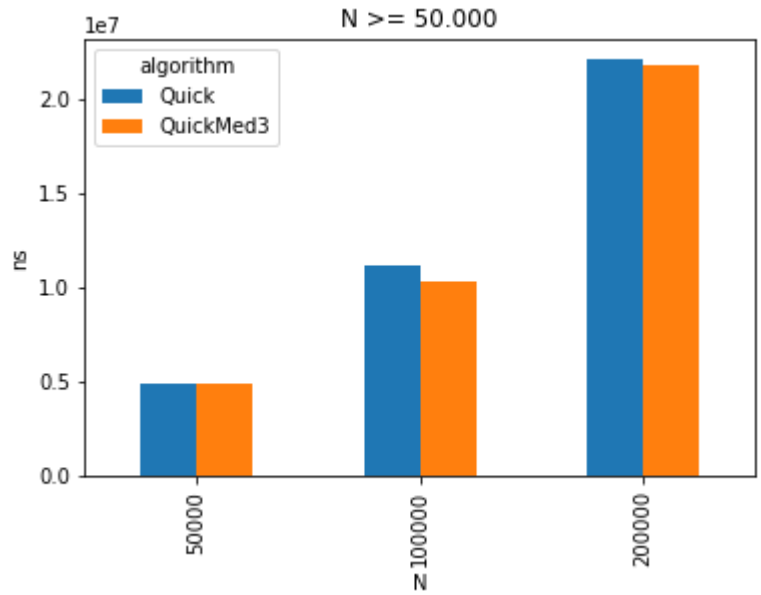
Como mencionado anteriormente, os resultados foram uma completa surpresa. O `sorting::insertion` performou melhor que todos os outros algoritmos implementados para o trabalho, por uma grande margem. Vale observar também que, apesar de mais lento em todos os casos, ele se manteve a menos de uma ordem de magnitude da performance do `std::sort` provido pela biblioteca-padrão `<algorithm>`, cuja complexidade é dada como $O(n\log(n))[5].

QUICKSORT "CLRS" VS QUICKSORTMED3

Ao contrário das expectativas, o `sorting::quick_med3` não se mostrou mais rápido que o `sorting::quick` "*naive*". Causas prováveis são: custos de manipulação das estruturas de dados, distribuição dos dados de teste e claro, erros de programação. Apesar disso, talvez em n muito grandes (isto é, > 50000) o QuicksortMed3 execute tão bem ou melhor que o Quicksort básico, como em $N = 50000, 100000, 200000$.

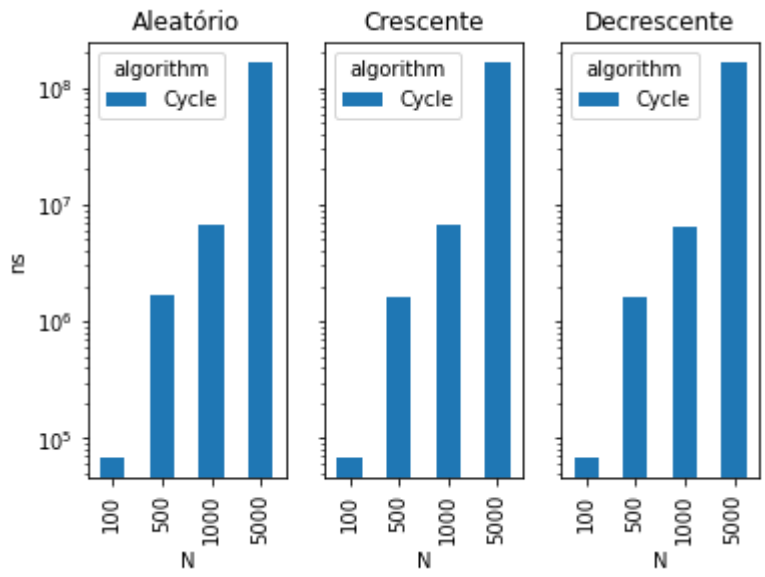






CYCLE

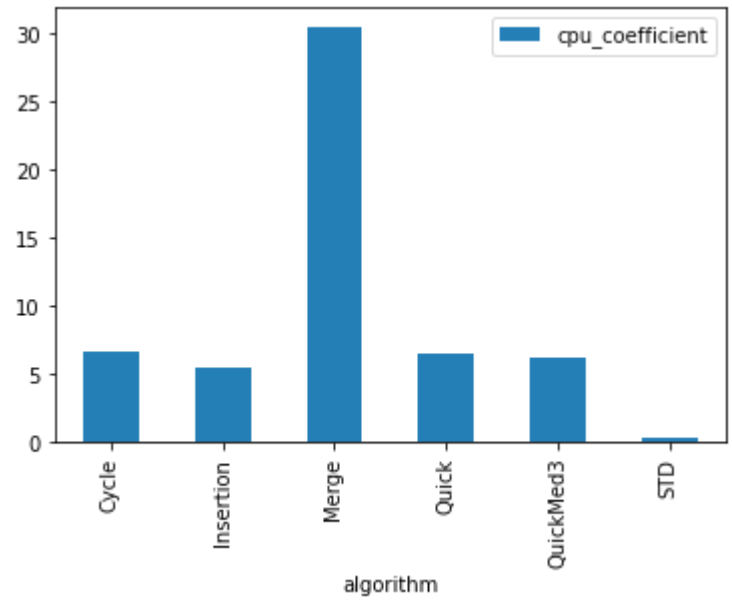
Como descrito anteriormente, o algoritmo `sorting::cycle` é lento.

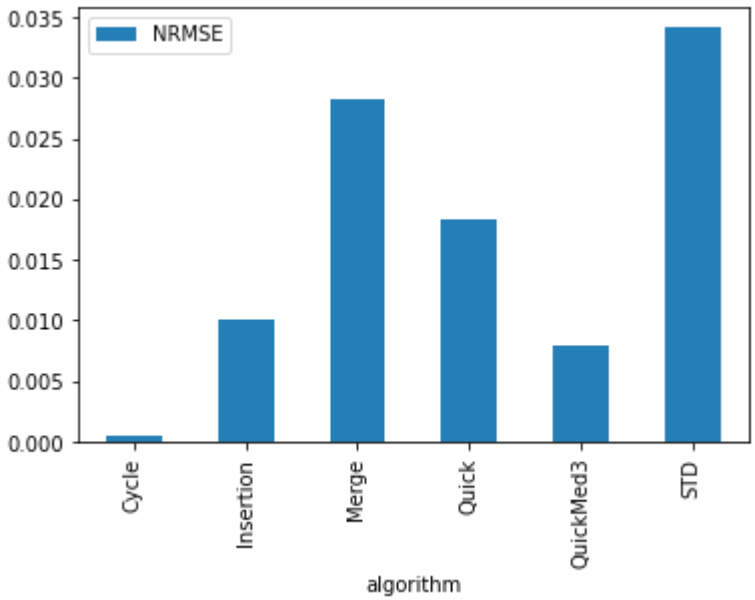


APROXIMAÇÃO DE COEFICIENTES BIG-O

A biblioteca utilizada para benchmark, Google Benchmark, possui funcionalidades para calcular o coeficiente para o termo de maior ordem da complexidade assintótica de uma função[6], juntamente com seu RMSE normalizado.

Sumarizando esses resultados:





E a complexidade que foi passada para a bibliotecar estimar:

Algoritmo	Complexidade Assintótica
Insertion	n
Merge	$n \log(n)$
Quick	$n \log(n)$
QuickMed3	$n \log(n)$
Cycle	n^2

Algoritmo	Complexidade Assintótica
STD	$n \log(n)$

REFERÊNCIAS

1. “Google Benchmark”
   Documentação e código-fonte: https://github.com/google/benchmark
   Acessado em: 07/03/2021 ↩
2. “Introduction to Algorithms, Third Edition”
   Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest & Clifford Stein
   Cópia física, 3ª edição ↩
3. “Engineering a Sort Function”
   Jon L. Bentley & M. Douglas McIlroy
   Disponível em: https://cs.fit.edu/~pkc/classes/writing/papers/bentley93engineering.pdf
   Acessado em: 07/03/2021 ↩
4. “Cycle-Sort: A Linear Sorting Method”
   Bruce K. Haddon
   Disponível em: https://academic.oup.com/comjnl/article/33/4/365/377624
   Acessado em: 07/03/2021 ↩
5. “std::sort - cppreference.com”
   Disponível em: https://en.cppreference.com/w/cpp/algorithm/sort
   Acessado em: 07/03/2021 ↩
6. “Calculating Asymptotic Complexity (Big O)”
   Disponível em: https://github.com/google/benchmark#asymptotic-complexity
   Acessado em: 07/03/2021 ↩