Universidade Federal de Minas Gerais

DCC UFMG

Trabalho Prático 1 - Estrutura de Dados

Rafael Paniago
Número de Matrícula: 2023028056

April 27, 2024

1 Introdução

O problema abordado nesse trabalho prático se refere à implementação e teste de desempenho de diferentes algoritmos de ordenação de vetores, em comparação com o seu desempenho de acordo com a teoria de complexidade assintótica. Os algoritmos abordados são: Método da bolha, Método da inserção, Método da seleção, Merge Sort, Quick Sort, Shell Sort, Counting Sort, Bucket Sort e Radix Sort.

A solução do problema envolve a implementação desses algoritmos em linguagem C e a execução de testes de benchmark nesses algoritmos em diferentes situações, envolvendo diferentes cargas de trabalho, como dimensão do vetor, tamanho das estruturas de dados presentes em cada uma das células individuais e ordenação inicial do vetor.

2 Método

Os algoritmos foram todos implementados em linguagem C, e foi escrito um script na mesma linguagem que realiza os testes de benchmark. Os algoritmos de ordenação em si foram implementadas utilizando apenas as estruturas de dados presentes na biblioteca padrão da linguagem, entre eles `int`(Variável inteira de 32 bits), `int*`(Ponteiro para variável inteira de 32 bits), `long long int`(Variável inteira de 64 bits), `long long int*`(Ponteiro para variável inteira de 64 bits).

O script, por outro lado, utilizou a função `gettimeofday` da biblioteca `<sys/time.h>` para poder calcular o tempo de execução de cada uma das funções de forma precisa, sem contar o tempo de inicialização de variáveis ou impressão de resultados. O script foi desenvolvido para que o teste seja feito apenas em uma função especificada para um tamanho de vetor, estrutura de dados contida no vetor e ordenação específicas, por isso foi desenvolvido um arquivo bash `runbenchmarks.sh` que executa o script de teste múltiplas vezes, com diferentes cargas de trabalho e salva os resultados em um arquivo .csv. Para vetores ordenados de forma aleatória, o script calcula a média de tempo de 50 testes realizados em vetores aleatórios diferentes em 50 seeds para cada tamanho de vetor, de forma a deixar a amostra de tempo o menos enviesada possível, e garantir a confiabilidade dos resultados.

Com os resultados, foram traçados 12 gráficos diferentes, 2 para cada situação testada, que representam a evolução dos 9 algoritmos em 6 diferentes cenários: Vetor Ordenado de int, Vetor Invertido de int, Vetor Aleatório de int, Vetor Ordenado de long long int, Vetor Invertido de long long int e Vetor Aleatório de long long int, à medida que o tamanho do vetor testado é aumentado.

3 Análise de Complexidade

À seguir, será analisada a complexidade teórica de tempo e espaço dos algoritmos implementados, de forma à comparar a complexidade de cada um com as curvas traçadas nos experimentos, mostrando a sua eficiência na prática. Além disso, a parte mais importante será compreender possíveis trade-offs a serem utilizados entre algoritmos, como por exemplo um algoritmo teoricamente menos eficiente podendo ser usado para ordenar vetores de menor tamanho, etc.

Bubble Sort

- Tempo:
  - Pior caso: $O(n^2)$
  - Melhor caso: $O(n^2)$
- Espaço: $O(1)$

Insertion Sort

- Tempo:
  - Pior caso: $O(n^2)$
  - Melhor caso: $O(n)$
- Espaço: $O(1)$

Selection Sort

- Tempo: $O(n^2)$
- Espaço: $O(1)$

Merge Sort

- Tempo: $O(n \log n)$
- Espaço: $O(n)$

Quick Sort

- Tempo:
  - Pior caso: $O(n^2)$
  - Caso médio: $O(n \log n)$
- Espaço: $O(\log n)$

Shell Sort

- Tempo: Depende da sequência de lacunas
- Espaço: $O(1)$

Counting Sort

- Tempo: $O(n + k)$
- Espaço: $O(n)$

Radix Sort

- Tempo: $O(d \cdot (n + k))$
- Espaço: $O(n)$

Bucket Sort

- Tempo:
  - Pior caso: $O(n^2)$
  - Melhor caso: $O(n + k)$
- Espaço: $O(n + k)$

4 Estratégias de Robustez

Nessa seção, serão listadas as estratégias de robustez empregadas na execução dos testes, de forma a tornar os resultados confiáveis.

- Implementação de geração aleatória de seeds e realização de múltiplos de testes para vetores gerados de mesmo tamanho, calculando a média do tempo gasto no algoritmo, de forma a tornar a amostra o menos enviesada possível.
- Implementação dos algoritmos feita respeitando o encapsulamento, de forma que o código possa ser refatorado sem que mecanismos que dependam da sua utilização tenham o seu funcionamento comprometido
- Utilização da função `gettimeofday`, com o objetivo de impedir que o tempo de impressão de resultados e o tempo de carregamento das variáveis altere os resultados experimentais dos testes.
- Realização de testes para uma larga variedade de tamanhos de vetores, de forma a evitar o underfitting das curvas traçadas sobre os gráficos de desempenho, e tornar o ajuste o mais confiável possível.

5 Análise Experimental

Como dito anteriormente, os algoritmos foram testados em 6 cenários diferentes, mudando a ordenação e a estrutura de dados presente e escalando o tamanho do vetor ao longo do gráfico. À partir do gráfico número 1, que representa o desempenho dos algoritmos ordenando vetores aleatórios de int, vamos ajustar curvas para metrificar a consistência dos resultados de acordo com a teoria. Os gráficos estão presentes nas páginas 5, 6 e 7.

Por razões de facilidade visualização os ajustes não serão colocados em cima do gráfico, mas seus R2 foram calculados para demonstrar a sua veracidade. Cada algoritmo foi ajustado com um curva de acordo com a sua categoria de complexidade assintótica de caso médio:

- $O(n^2)$: Insertion Sort, Bubble Sort, Selection Sort
- $O(nlogn)$: Shell Sort, Merge Sort, Quick Sort
- $O(n+k)$: Radix Sort, Counting Sort, Bucket Sort

Portanto, os algoritmos $O(n^2)$ foram ajustados a curvas quadráticas, os algoritmos $O(nlogn)$ foram ajustados para curvas polinomiais logaritmicas e os algoritmos $O(n+k)$ foram ajustados para funções lineares. A seguir estão os cálculos de seus R2:

$BubbleSort$, R^2 : 0.9425194003413252
$InsertionSort$, R^2 : 0.9395303252393756
$SelectionSort$, R^2 : 0.933707421276803
$MergeSort$, R^2 : 0.9970765410799112
$QuickSort$, R^2 : 0.9532903640349117
$ShellSort$, R^2 : 0.9861399517322342
$CountingSort$, R^2 : 0.975059919310175798
$BucketSort$, R^2 : 0.9665243034083548
$RadixSort$, R^2 : 0.9659738247401355

Os valores de R2 extremamente próximos de 1 demonstram que os testes dos algoritmos chegam muito perto de sua complexidade assintótica teórica, mostrando a veracidade dos testes.

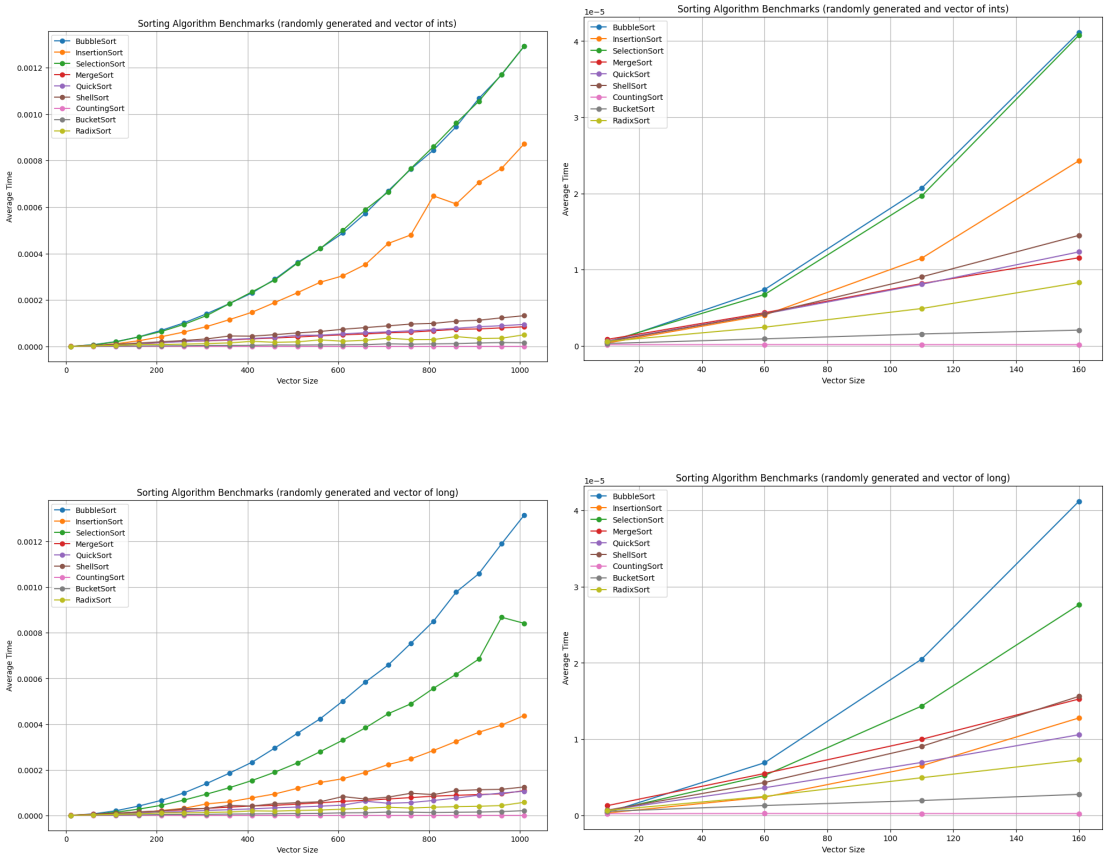

Figure 1: Os 4 gráficos acima representam o comportamento dos algoritmos para vetores não ordenados. É interessante observar que para números grandes de n as trajetórios dos linhas dos algoritmos de cada ordem de complexidade($O(n^2)$, $O(nlogn)$ e $O(n+k)$) tendem a ficar extremamente agrupados e diferentes dos outros grupos. Ao mesmo tempo, podemos observar nos gráficos com menores tamanhos de vetor quais são cada uma das classes de complexidade, e que, interessantemente, para tamanhos pequenos algoritmos teoricamente menos eficientes podem ser preferíveis (para vetores pequenos pode-se usar insertionSort ao invés de QSort, o que é interessante visto que teoricamente QSort é um algoritmo muito mais eficiente. Além disso, podemos ver quais os algoritmos que possuem mais trocas, observando quais algoritmos tendem a crescer mais com o uso de longs ao invés de ints. Outra coisa clara é como todos os testes tendem a demorar mais quando a estrutura de dados o conteúdo de cada célula do vetor muda de int para long, aumentando em média 2 vezes, mas mais para vetores que fazem mais trocas

s

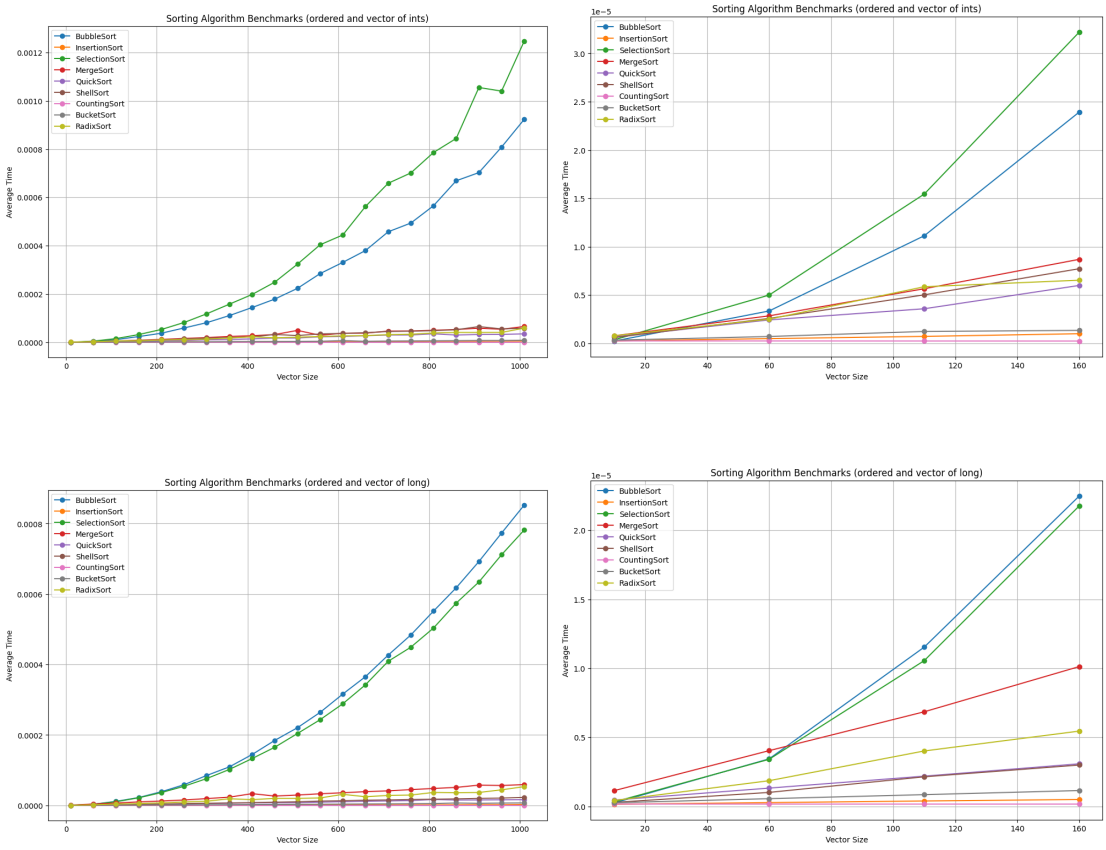

Figure 2: Os 4 gráficos acima representam o comportamento de cada um dos algoritmos para vetores ordenados. É interessante observar que nesse caso, insertionSort é um algoritmo extremamente eficiente, como descrito na teoria. Podemos observar também, como os outros algoritmos quadráticos são muito piores em relação aos outros nesses casos, pois tendem a continuar tendo um comportamento quadrático independenmente da ordenação do vetor. Nesse caso também, podemos ver que o mergeSort e ShellSort não tem seus comportamentos consideravelmente alterados, mostrando que eles tendem a ser da ordem de $O(nlogn)$ independementemente da ordenação do vetor, e que os algoritmos lineares tendem a ter comportamente semelhante também.

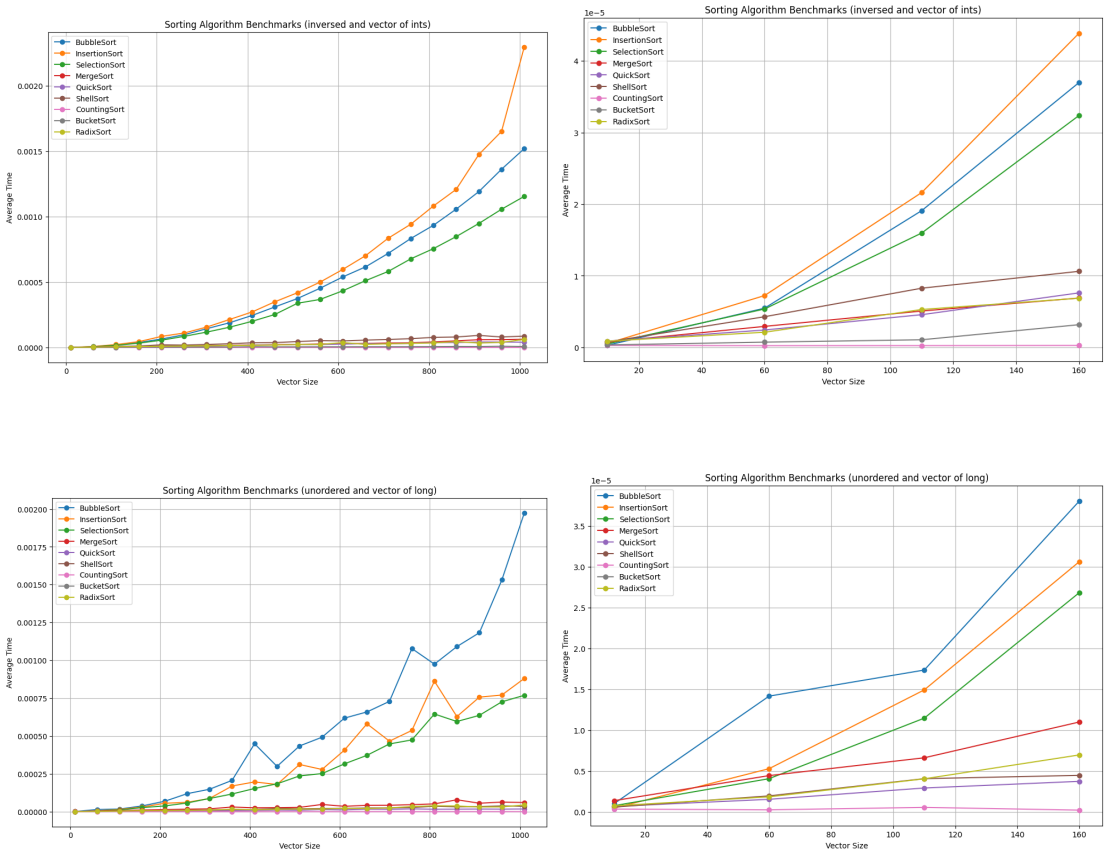

Figure 3: Os 4 gráficos acima representam o comportamento de cada um dos algoritmos para vetores invertidos. A primeira coisa interessante a se observar é como o QuickSort se torna um algoritmo extremamente ineficiente nesse caso, enquanto os outros algoritmos de ordem $O(nlogn)$ continuam tendo um desempenho semelhante, mostrando como ele é um algoritmo melhor que os outros de sua classe para casos médios mas tem um trade-off em relação a casos piores. Além disso, é possível ver como o insertionSort tende a ser o pior algoritmo nesse caso, enquanto os mesmos de sua classe são muito piores que os outros ele é ainda pior que seus pares, sendo que no caso de vetor invertido seu tempo de execução explode. Além disso, podemos observar de novo como o bubbleSort tende a ser pior em relação a seus pares quando suas células tendem a ter estruturas de dados mais pesadas, como long long int, pois ele realiza mais operações de troca, que se tornam ineficientes à que o conteúdo das células aumenta.

6 Conclusões

Através da análise de algoritmos feita, algumas conclusões podem ser tiradas em relação ao funcionamento de algoritmos de ordenação.

1. A primeira e mais importante é que algoritmos, na maioria considerável das vezes, se tratam de trade-offs. Nem sempre um algoritmo que é teoricamente mais eficiente em termos de tempo será a melhor escolha. Para cada situação diferente um algoritmo diferente será o mais adaptado, portanto é extremamente importante para o programador saber qual algoritmo utilizar e quando.

2. Outra conclusão importante de se ressaltar é a grande diferença que a complexidade espacial faz. Na maioria das vezes, os cientistas da computação consideram apenas o número de iterações dos algoritmos para estimar o seu tempo de execução. Mas a utilização de diferentes estruturas de dados impacta e muito a eficiência dos algoritmos.

3. Além disso, é importante que o programador considere também o impacto da complexidade espacial não no tempo de execução mas na própria memória do computador. Apesar da utilização de algoritmos de ordem linear ser muitas vezes preferível, é necessário considerar o impacto da sua execução na memória RAM. A ordenação de vetores muito grandes pode sobrecarregar a memória e comprometer o funcionamento não só do algoritmo como do próprio computador.

4. A teoria de complexidade assintótica de algoritmos é uma ótima forma de nos orientarmos de forma geral em relação à sua execução. Entretanto, existem casos onde algoritmos teoricamente mais eficientes se tornam menos eficientes, portanto é muito importante que o programador não baseie suas escolhas exclusivamente na teoria de complexidade assintótica.

7 Bibliografia

References

Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, 2009.

Robert Sedgewick and Kevin Wayne. Algorithms. *Addison-Wesley*, 2011.