Trabalho de estruturas de dados

Lucas Silva

2020097243

A seguinte documentação será organizada da seguinte forma:
Seção 1: Introdução.
Seção 2: Análise de complexidade, classes e métodos.
Seção 3: Estratégia de robustez e análise experimental.
Seção 4: Conclusão.

1 Introdução

Nesse trabalho vamos falar sobre a analise experimental e a complexidade dos algoritmos de ordenação dados em sala de aula. Sendo eles:

- Bubble sort
- Selection sort
- Insert sort
- Quick sort
- Merge sort
- Shell sort
- Counting sort
- Bucket sort
- Radix sort

Logo, vamos verificar as cargas de trabalho das entradas, e averiguar se elas estão inversamente ordenadas, ordenadas ou bagunçadas. Verificaremos também a quantidade de elementos a serem ordenados, a complexidade assintótica e métodos implementados nos algoritmos, e por fim, dado um elemento qualquer do vetor verificaremos seu tamanho em bytes. Logo em seguida vamos realizar uma analise experimental, ou seja para cada configuração dos vetores, vamos pegar uma classe de complexidade e vamos gerar um gráfico sobre os algoritmos da determinada classe de complexidade. Assim, vamos concluir sobre cada configuração de vetores, qual é o melhor algoritmo para cada entrada testada no programa.

2 Classes e métodos

2.1 Classe de Carga de Trabalho

Essa classe possui seis métodos, onde tais métodos calculam a carga de trabalho que foi proposta pelo trabalho.

- **qtsiordena()**:

  Esse método tem como objetivo retornar o número de elementos de um vetor, para ordenar o mesmo.

  A complexidade desse método em relação ao tempo é $O(n^2)$, pois ele chama o algoritmo Bubble sort para ordenar e fazer as comparações de onde cada elemento deveria estar, já a complexidade com relação ao espaço é $O(n)$, devido a quantidade de itens que vão ser armazenados na memória.

- **Tamanhoembytes()**:

  Esse método alocamos dinamicamente um ponteiro, e armazenamos o tamanho em bytes de cada item do nosso vetor experimental.

  A complexidade desse algoritmo é $O(n)$ tanto em relação ao tempo quanto em relação ao espaço.

- **Olhatamanhovetor()**:

  Esse método tem como objetivo setar se o vetor é organizado, inversamente ordenado ou bagunçado. Logo ele verifica se os elementos estão na forma sequencial ou na forma inversamente sequencial.

  A análise de complexidade desse algoritmo é $O(n)$ em relação ao tempo e $O(n)$ em relação ao espaço.

- **getIsordenado()**, **getBaguncado()**, **getInversa()**:

  Esses métodos retornam como saída se o vetor é inversamente ordenado, ordenado ou bagunçado. Eles retornam o conteúdo dos atributos *_isOrdenado*, *_baguncado*, *_inversamenteO*, já que os atributos são privados. E possuem complexidade $O(1)$ tanto em relação ao tempo quanto em relação ao espaço.

2.2 Classe Objeto

Essa classe armazena as chaves para serem utilizadas nos algoritmos de ordenação e tem os seguintes métodos:

- **getChave():**

  Este método tem como objetivo retornar o valor do atributo *_chave*.

- **SetChave():**

  Este método tem como objetivo definir na variável chave o valor passado por parâmetro.

- **incrementa1(), decrementa1(), incrementarn():**

  Estes três métodos são usados no algoritmo *Counting Sort* para aumentar 1, diminuir 1 e aumentar n, respectivamente. Como o atributo *_chave* é um atributo privado, para podermos incrementar e decrementar no algoritmo, precisamos de um método para acessar esse atributo e fazer as alterações necessárias.

2.3 Classe Sort

Essa classe contêm os algoritmos de ordenação que foram propostos pelo trabalho, cada método é um algoritmo ou um método auxiliar para alguns algoritmos, como o particionar() do Quick sort.

- **Bubble sort()**:

  Esse método é o algoritmo de ordenação *bubble sort*, na qual sua complexidade é $O(n^2)$ no pior caso. Já no melhor caso é $O(n)$.

- **Selection sort()**:

  Esse método é o algoritmo de ordenação *selection sort*. Sua complexidade é $O(n^2)$ no pior caso e também no melhor caso.

- **Insert Sort()**:

  Esse método é o algoritmo de ordenação *insert sort*. Sua complexidade é $O(n)$ no melhor caso, e no pior caso é $O(n^2)$.

- **particionar()**:

  Esse método é para auxiliar o algoritmo *quick sort*, e serve para fazer as partição do vetor.

- **ordena()**:

  Esse método é serve para chamar a partição, e fazem as ordenações do **algoritmo**.

- **Quick sort()**:

  Esse método chama a ordenação onde começa a rodar o algoritmo do *quick sort*. Sua complexidade é $O(nlog(n)$, já que o pivô desse algoritmo é sempre a média, tanto no pior caso quanto no melhor caso.

- **Merge()**:

  Esse método serve para combinar duas listas ordenadas em uma única lista ordenada,

  serve para o mergesort.

- **Mergesort()**:

  Esse método serve para particionar as listas, realizando chamadas recursivas. A complexidade deste método é $O(nlong(n))$.

- **Counting sort()**:

  Esse método serve para fazer a chamada do algoritmos *counting sort*. A complexidade desse algoritmo $O(n)$, tanto no pior quanto no melhor caso.

- **Maxtam()**:

  Esse método serve para contar o número de baldes que vão ser usados para fazer o *bucket sort*.

- **Bucketsort():**

  Esse método serve para fazer a chamada do algoritmo de ordenação *bucket sort*. A complexidade desse algoritmo é $O(n + k)$, tanto no pior quanto no melhor caso.

- **Shell sort()**:

  Esse método serve para fazer a chamada do algoritmo de ordenação *shell sort*. A complexidade desse algoritmo é $O(nlog(n))$ no melhor caso, e no pior caso é $O(n)$.

- **radixs()**:

  Esse método serve para fazer a chamada do algoritmo de ordenação *radix sort*.A complexidade desse algoritmo é $O(d * (n + k))$ no pior caso, e se d for constante, o melhor caso é $O(n)$.

2.4 Método

- **Obterobjeto()**:

  Esse método serve para obter um objeto com chaves aleatórias. A complexidade desse método é $O(n)$ no pior caso e no melhor caso.

- **Ajustavetor()**:

  Esse método serve para passar o conteúdo do vetor passado por parâmetro para um vetor auxiliar. Sua complexidade é $O(n)$ no pior caso e no melhor caso.

- **voidprint()**:

  Esse método imprimi as chaves dos objetos dos vetores. Sua complexidade é $O(n)$ no pior e no melhor caso.

- **ObterobjOrd()**:

  Esse método obtém chaves ordenadas para os objetos dos vetores. Sua complexidade é $O(n)$ no pior é no melhor caso.

- **ObterobjInv()**:

  Esse método obtém chaves inversamente ordenadas para os objetos do vetores. Sua complexidade é $O(n)$ no pior e no melhor caso.

- **printsituation()**:

  Esse método ele define qual é o melhor algoritmo de acordo com o tamanho de cada vetor. Sua complexidade é $O(n)$ no pior e no melhor caso.

3 Análise experimental e estratégia de robustez

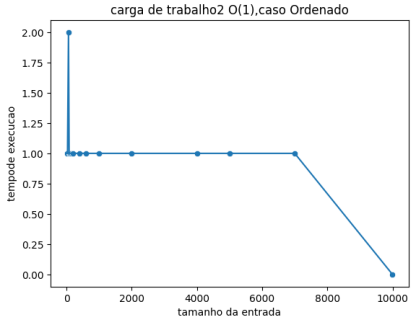
3.1 Estratégia de robustez

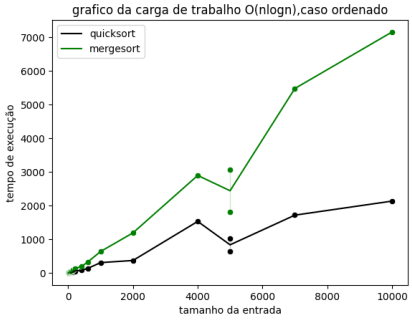
As estratégias de robustez utilizadas, foram encapsulamento de atributos em algumas classes e alguns loops para o usuário do programa não colocar valores fora do [0, ultima posição do vetor], logo vai haver uma repetição ate ele colocar uma entrada valida. Também foram utilizados alguns *if's* para verificar se os arquivos foram abertos como deveriam.

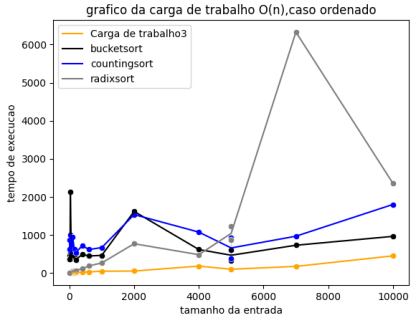
Podemos concluir que não tivemos muitas alterações de estrategia de robustez pelo programa ser bem linear e objetivo.

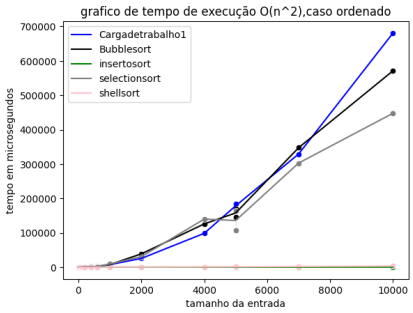
3.2 Análise experimental

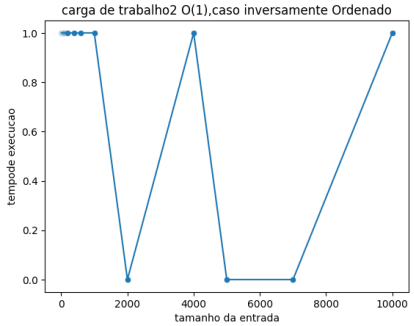
Na análise experimental temos o tempo dos algoritmos, onde os gráficos foram gerados em relação a tamanho do vetor, no tempo de microssegundos, e logo geramos um gráfico para cada classe assintótica de algoritmos, para cada configuração de vetor, vamos estar gerando um gráfico para cada classe assintótica dos algoritmos, logo vamos verificar as comparações

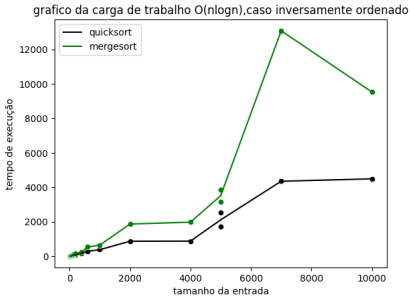












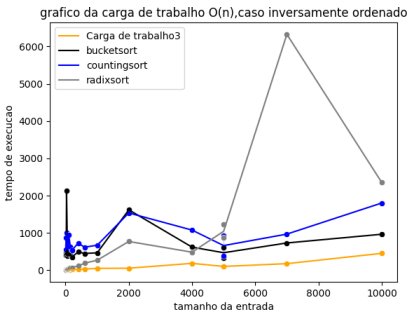


grafico da carga de trabalho O(n),caso inversamente ordenado
Carga de trabalho3
bucketsort
countingsort
radixsort
tempo de execucao
tamanho da entrada

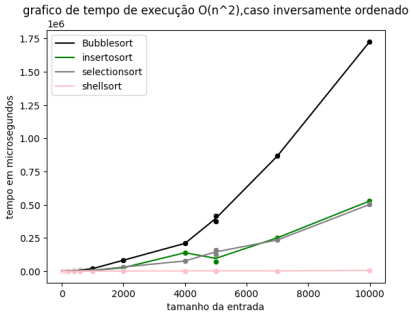
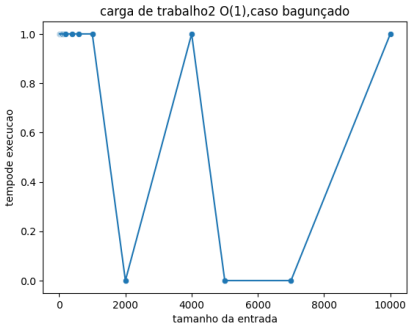


grafico de tempo de execução O(n^2),caso inversamente ordenado
1e6
Bubblesort
insertosort
selectionsort
shellsort
tempo em microsegundos
tamanho da entrada


carga de trabalho2 O(1),caso bagunçado
tempode execucao
tamanho da entrada

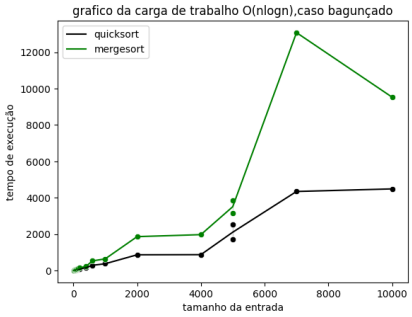


grafico da carga de trabalho O(nlogn),caso bagunçado
quicksort
mergesort
tempo de execução
tamanho da entrada

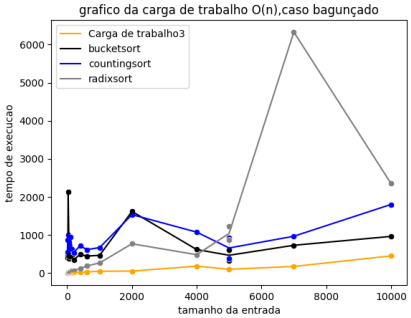


grafico da carga de trabalho O(n),caso bagunçado
Carga de trabalho3
bucketsort
countingsort
radixsort
tempo de execucao
tamanho da entrada

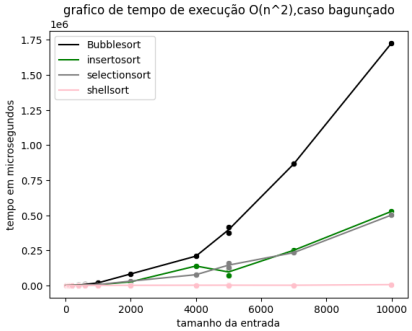


grafico de tempo de execução O(n^2),caso bagunçado
1e6
Bubblesort
insertosort
selectionsort
shellsort
tempo em microsegundos
tamanho da entrada

3.3 Analise

Bom a analise sera feita a partir do tempo, com relação a cada entrada logo vamos ver qual é o melhor algoritmo em relação ao tempo, para cada entrada que foram utilizadas nos testes.

caso bagunçado:[h] Quando as entradas são pequenas o melhor algoritmo é o insertsort em questão de custo computacional ,quando o vetor começa a aumentar a partir de tamanho 30 o melhor algoritmo é o shellsort em questão de custo computacional, para uma entrada a partir de 150 temos que o radixsort foi bem mais otimizado do que outros algoritmos com relação ao custo computacional,quando aumentamos a entrada do algoritmo para 5000 o bucketsort passou a ser o melhor algoritmo em relação ao custo computacional.

caso ordenado:[h] Quando o algoritmo esta totalmente ordenado em todas as entradas de vetores o melhor algoritmo foi o insertsort, justamente porque o insertsort minimiza o numero de operações necessária e consequentemente o que é bom para o custo computacional.

caso inversamente ordenado[h] para 4 itens no vetor o melhor algoritmo acabou sendo selectionsort juntamente com shellsort em questão de tempo e custo computacional, para o tamanho 10 tivemos 4 algoritmos que tiveram baixo custo computacional, insertsort, quicksort e selectionsort e o shellsort, e na medida que o tamanho da entradas foram aumentando, os melhores algoritmos ficavam variando entre quicksort e o shellsort em questão de custo computacional, ate chegar em um valor muito grande que o bucketsort acaba sendo o melhor algoritmo de ordenação para custo computacional.

4 Conclusão:

Bom apos implementação dos algoritmos e das cargas de trabalho a meta era saber o melhor algoritmo mais eficiente para cada tipo de entrada e para cada configuração do vetor entre ordenado,inversamente ordenado,e bagunçado e assim conseguimos concluir qual o melhor algoritmo para cada caso. Apos concluirmos a Analise esse trabalho teve foco em duas partes como a analise experimental e o programa. A analise experimental com os algoritmos de ordenação propostos em sala de aula foram, bom conseguimos concluir que bom para um algoritmo de ordenação ser melhor existem vários fatores que podem influencia como as cargas de trabalhas propostas como: se o algoritmo e ´ordenado ou não, o tamanho dos vetores também é algo que influencia, e o tamanho dos conteúdos que tem nos algoritmos e o seu tamanho. Já no programa um dos maiores desafios foram testar os casos no programa

já que tínhamos que cobrir muitos casos para podermos gerar uma analise e vê ate onde cada algoritmo é eficiente. Entretanto o maior desafio desse projeto foi realizar a analise dos algoritmos por causa dos vários fatores para concluirmos algo com relação a analise experimental.

link para o codigo: https://github.com/minipatch/tp1