CURSO DE CIÊNCIA DE DADOS

ISADORA ALVES TEIXEIRA PREVITALLE

RELATÓRIO DA ANÁLISE DE ALGORITMO DE ORDENAÇÃO:

Adamantina- SP
2023

CURSO DE CIÊNCIA DE DADOS

LISTA DE FIGURA

CURSO DE CIÊNCIA DE DADOS

SUMÁRIO

Adamantina- SP
2023

CURSO DE CIÊNCIA DE DADOS

1 INTRODUÇÃO

Este artigo de pesquisa tem como objetivo apresentar um trabalho prático realizado para explorar a Análise de Algoritmos na perspectiva de Algoritmos de Ordenação. Durante o projeto, um conjunto de algoritmos de ordenação foi implementado e submetido a uma Análise Exploratória para calcular o tempo de execução de cada algoritmo implementado. Além disso, foi realizada uma Análise Assintótica detalhada para destacar o melhor caso, caso médio e pior caso dos algoritmos implementados e comparar os resultados obtidos com os resultados obtidos na Análise Experimental. Para a obtenção da Análise Assintótica, foram utilizados materiais de aula e pesquisas na internet. Os resultados obtidos demonstraram a eficácia de diferentes algoritmos de ordenação em diferentes cenários, fornecendo insights valiosos para a escolha do algoritmo mais apropriado em determinados casos. Este trabalho prático visa contribuir para o avanço do conhecimento em Análise de Algoritmos e reforçar conhecimentos obtidos durante as aulas.

2 ALGORITMOS DE ORDENAÇÃO

2.1 Bubble Sort (versão original sem melhorias)

O algoritmo bubble sort é um método simples de ordenação. Ele percorre

CURSO DE CIÊNCIA DE DADOS

repetidamente uma lista de valores, comparando pares adjacentes e os trocando se necessário, até que a lista esteja completamente ordenada.

O nome "bubble" vem da forma como os elementos maiores "flutuam" (ou "borbulham") para o topo da lista durante esse processo.

O algoritmo começa comparando o primeiro elemento com o segundo. Se o primeiro elemento for maior do que o segundo, eles são trocados. A comparação é então feita entre o segundo e o terceiro elemento, e assim por diante até o final da lista.

Isso constitui a primeira passagem. Na segunda passagem, o algoritmo começa novamente do início da lista e repete todo o processo até que a lista esteja completamente ordenada.

Embora seja um algoritmo de classificação simples, o bubble sort é geralmente menos eficiente do que outros métodos mais sofisticados de ordenação, especialmente para listas grandes ou em cenários de tempo crítico.

O algoritmo trabalha com estruturas de dados do tipo arrays e listas ligadas, possuindo como complexidade de pior e melhor caso $O(n^2)$.

Em python o algoritmo tem o seguinte código:

```
# Geração de 100 variáveis aleatórias
import random
lista = [random.randint(0, 99) for i in range(100)]
# Algoritmo de bubble sort
def bubble_sort(lista):
 n = len(lista)
 for i in range(n-1):
```

CURSO DE CIÊNCIA DE DADOS

```
  for j in range(0, n-i-1):
   if lista[j] > lista[j+1]:
    lista[j], lista[j+1] = lista[j+1], lista[j]
 return lista
# Teste do algoritmo
print("Lista original: ", lista)
lista_ordenada = bubble_sort(lista)
print("Lista ordenada: ", lista_ordenada)
```

2.2 Bubble Sort (versão que verifica se o vetor já está ordenado)

O Bubble Sort é um algoritmo de ordenação simples, que percorre o vetor várias vezes e compara dois elementos consecutivos. A cada passagem do algoritmo, os elementos adjacentes que estão fora da ordem são trocados até que todo o vetor esteja ordenado.

Na versão que verifica se o vetor já está ordenado, a verificação ocorre antes de começar as trocas. Se o vetor já estiver ordenado, o algoritmo encerra a execução sem precisar fazer nenhuma troca. Isso ocorre porque, durante uma passagem completa do Bubble Sort, sempre há pelo menos um elemento que troca de posição com outro elemento, mesmo que o vetor esteja ordenado.

Essa verificação prévia pode tornar o algoritmo mais eficiente em alguns casos, pois evita a realização de operações desnecessárias. No entanto, se o vetor não estiver ordenado, a eficiência do Bubble Sort ainda será $O(n^2)$, ou seja, seu desempenho piora significativamente para vetores maiores. Por isso, outros algoritmos de

CURSO DE CIÊNCIA DE DADOS

ordenação mais complexos são preferíveis para lidar com grandes volumes de dados. O algoritmo trabalha com estruturas de dados do tipo arrays e listas ligadas, possuindo como complexidade de pior caso $O(n^2)$ e melhor caso $O(n)$.

Em python o algoritmo tem o seguinte código:

```
import random
# gerando uma lista com 100 variáveis aleatórias
lista = []
for i in range(100):
 lista.append(random.randint(0, 100))
print("Lista desordenada: ", lista)
# implementando o algoritmo bubble sort melhorado
n = len(lista)
trocou = True
while trocou:
 trocou = False
for i in range(n-1):
 if lista[i] > lista[i+1]:
  lista[i], lista[i+1] = lista[i+1], lista[i]
  trocou = True
  n -= 1
print("Lista ordenada: ", lista)
```

2.3 Insertion Sort

CURSO DE CIÊNCIA DE DADOS

O algoritmo de ordenação por inserção (insertion sort) é um método simples e eficiente de organizar elementos em uma lista ou array. Ele funciona percorrendo a lista e "inserindo" cada novo elemento na posição correta da sequência ordenada.

O processo começa com o segundo elemento, que é comparado com o primeiro. Se o segundo for menor que o primeiro, eles são trocados de posição. Em seguida, o terceiro elemento é comparado com os dois primeiros, e assim por diante, até que todos os elementos sejam processados e a lista esteja completamente ordenada.

O algoritmo é chamado de "sort" (ordenação) porque ele classifica a lista de forma crescente ou decrescente, dependendo da implementação utilizada. O insertion sort é considerado um dos mais simples e intuitivos algoritmos de ordenação, sendo amplamente utilizado em programas de computador e outras aplicações que requerem a organização de dados.

O algoritmo trabalha com estruturas de dados do tipo arrays e listas ligadas, possuindo como complexidade de pior caso $O(n^2)$ e melhor caso $O(n)$.

Em python o algoritmo tem o seguinte código:

```
import random
def insertion_sort(lst):
 for i in range(1, len(lst)):
  key = lst[i]
  j = i - 1
  while j >= 0 and lst[j] > key:
   lst[j + 1] = lst[j]
   j -= 1
```

CURSO DE CIÊNCIA DE DADOS

```
    lst[j + 1] = key
    return lst
lst = [random.randint(1, 1000) for i in range(100)]
print("Lista desordenada:")
print(lst)
print("\nLista ordenada:")
print(insertion_sort(lst))
```

2.4 Shell Sort

O algoritmo Shell Sort também conhecido por “ordenação por incrementos diminutivos” é uma classificação de complexidade quadrática, que utiliza um método de inserção direta criado por Donald Shell. Ele permite a troca de registros distantes um do outro, com a metodologia de dividir um grande vetor ou uma lista em valores menores para aplicar o método da ordenação por inserção a fim de obter uma sub lista estruturada.

O que torna o Shell Sort diferente do algoritmo de ordenação por inserção é que ele usa uma sequência de lacunas (gaps) para determinar o tamanho das sub-listas. A ideia é começar com uma lacuna grande e, em seguida, reduzi-la gradualmente até que ela seja igual a 1. Essa abordagem permite que os elementos se movam mais rapidamente em direção à sua posição final na lista ordenada.

O algoritmo trabalha com estruturas de dados do tipo arrays e listas ligadas, possuindo como complexidade de pior caso dado por O(n) e melhor caso a classe de comportamento assintótico dado por $O(n^2)$.

CURSO DE CIÊNCIA DE DADOS

Em python o algoritmo tem o seguinte código para ordenação de uma lista de 10 elementos aleatórios:

```
import random
import time
Lista = random.sample(range(11), 10)
inicio = time.time()
def shell_sort(arr):
    n = len(arr)
    gap = n // 2
    while gap > 0:
        for i in range(gap, n):
            temp = arr[i]
            j = i
            while j >= gap and arr[j - gap] > temp:
                arr[j] = arr[j - gap]
                j -= gap
            arr[j] = temp
        gap //= 2
    return arr
 Ordenada = shell_sort(Lista)
fim = time.time()
print(Lista)
    #exibindo o tempo para ordenar a lista
```

```
print("Tempo de execução em segundos é {:.2f}".format(fim-inicio))
```

2.5 Selection Sort

O algoritmo Selection Sort é uma ordenação por seleção baseado em se colocar sempre o menor valor de um vetor para sua primeira posição ao percorrê-lo, comparando os elementos a partir de sua direita para realizar a determinação do valor mínimo e assim efetuar a troca do mesmo para o índice 0. Essa instrução se repete até o último índice do vetor a fim de encerrar o processo até que a lista esteja ordenada.

Sua ideia consiste em realizar essa ordenação com base na seleção de cada interação com os menores itens possíveis, colocando-os da esquerda para direita.

Uma de suas principais vantagens está na fácil implementação do algoritmo, ocupando baixo percentual de memória e, uma de suas principais desvantagens é o fato de demandar maior tempo de execução para listas com grandes números de itens.

A ordenação trabalha com estruturas de dados do tipo arrays e listas ligadas, possuindo como complexidade de pior e melhor caso a classe de comportamento assintótico dado por $O(n^2)$.

Em python o algoritmo tem o seguinte código para ordenar uma lista aleatória de 10 elementos:

```
import random
import time
random_list = random.sample(range(11), 10)
inicio = time.time()
```

CURSO DE CIÊNCIA DE DADOS

```
def selection_sort(arr):
    n = len(arr)
    for i in range(n):
        min_idx = i
        for j in range(i + 1, n):
            if arr[j] < arr[min_idx]:
                min_idx = j
        arr[i], arr[min_idx] = arr[min_idx], arr[i]

    return arr
sorted_list = selection_sort(random_list)
fim = time.time()
print(sorted_list)
print("Tempo de execução em segundos é {:.2f}".format(fim-inicio))
```

2.6 Quicksort (com pivô sendo o primeiro elemento da lista - partição)

O algoritmo quicksort é um método de ordenação inventado por Charles Antony Richard Hoare pertencente à categoria de ordenação por troca que consiste em rearranjar chaves de modo que as mesmas que forem menores precedem as chaves maiores ordenando-as consecutivamente em duas sub listas até que a lista completa se encontre ordenada.

Esse método utiliza o paradigma de programação dividir para conquistar, cujo, é uma abordagem recursiva em que a entrada de dados é ramificada múltiplas vezes a fim

de quebrar o problema maior em menores.

Quando o quicksort escolhe um pivô no início ou no final do array o mesmo recebe como parâmetro dois índices do array que será particionada, então escolhe-se um índice i e percorre-se o array usando outro índice j realizando trocas, quando necessário, a fim de que todos os elementos menores ou iguais ao pivô fiquem antes do índice i e os elementos i + 1, ou j - 1, sejam maiores que o pivô .

O algoritmo trabalha com estruturas de dados do tipo arrays e listas ligadas, possuindo como complexidade de pior caso um comportamento dado por $O(n^2)$ e melhor caso um comportamento dado por O(n log n).

Em python o algoritmo tem o seguinte código:

```
def quick_sort(a, ini=0, fim=None):
  fim = fim if fim is not None else len(a)
  if ini < fim:
     pp = particao(a, ini, fim)
     quick_sort(a, ini, pp)
     quick_sort(a, pp + 1, fim)
  return a
def particao(a, ini, fim):
  pivo = a[fim - 1]
  for i in range(ini, fim):
     if a[i] <= pivo:
        a[i], a[ini] = a[ini], a[i]
        ini += 1
```

CURSO DE CIÊNCIA DE DADOS

```
  return ini - 1
a = [8, 5, 12, 55, 3, 7, 82, 44, 35, 25, 41, 29, 17]
print(a)
print(quick_sort(a))
```

2.7 Quicksort (com pivô sendo o primeiro central da lista - partição)

O Quicksort é um algoritmo de ordenação de alta eficiência que utiliza a estratégia de divisão e conquista. A ideia básica do algoritmo é escolher um elemento da lista (o pivô) e particionar a lista em duas sub-listas, uma contendo todos os elementos menores que o pivô e outra contendo todos os elementos maiores ou iguais ao pivô. Em seguida, o algoritmo é aplicado recursivamente a cada sub-lista.

Basicamente não há diferenças entre com pivô sendo o primeiro da lista e com pivô sendo sendo primeiro central da lista, pois são o mesmo algoritmo. Entretanto, a eficiência é maior em alguns casos, quando a variação com pivô sendo o primeiro central da lista. Sua complexidade no melhor caso e pior caso sendo O(n log n).

Para implementá-lo, escolha um elemento que será o pivô, no caso, o primeiro elemento central, divida a lista em duas partes, uma com elementos menores ou iguais ao pivô, e a outra com elementos maiores, e aplique o algoritmo, mais partições criadas, até que a lista esteja completamente ordenada, como no exemplo abaixo:

```
import time
def partition(arr, low, high):
    pivot = arr[low]
    i = low + 1
```

CURSO DE CIÊNCIA DE DADOS

```
    j = high
    while True:
        while i <= j and arr[i] <= pivot:
            i += 1
        while i <= j and arr[j] >= pivot:
            j -= 1
        if i <= j:
            arr[i], arr[j] = arr[j], arr[i]
        else:
            break
        arr[low], arr[j] = arr[j], arr[low]
    return j
def quick_sort(arr, low, high):
    if low < high:
        pivot = partition(arr, low, high)
        quick_sort(arr, low, pivot - 1)
        quick_sort(arr, pivot + 1, high)
    return arr
arr = [random.randint(1, 10) for _ in range(10)] # lista de 10 elementos aleatórios
start_time = time.time()
sorted_arr = quick_sort(arr, 0, len(arr) - 1)
end_time = time.time()
print(f"Tempo de execução: {end_time - start_time:.5f} segundos")
```

```
print(sorted_arr)
```

2.8 Merge Sort

O Merge Sort é um algoritmo de ordenação que divide a lista em duas metades, ordena cada metade recursivamente e depois mescla as duas metades em uma lista ordenada. Ele utiliza a técnica "Dividir para Conquistar" para ordenar a lista.

O algoritmo funciona da seguinte maneira: divide a lista em duas metades, com base no índice do meio, será ordenada cada metade recursivamente, aplicando o Merge Sort. Mescle as duas metades em uma única lista ordenada, comparando o primeiro elemento de cada sub-lista e adicionando o menor à lista resultante, depois repita o processo até que todas as metades estejam mescladas e ordenadas.

A complexidade de tempo do MergeSort é O(n log n) no pior, no melhor e no caso médio, tornando-o um dos algoritmos de ordenação mais eficientes em termos de tempo de execução. No entanto, o Merge Sort requer espaço adicional para armazenar as duas sub-listas enquanto elas estão sendo ordenadas, o que pode torná-lo menos eficiente em termos de espaço de memória do que outros algoritmos de ordenação. É amplamente utilizado em algoritmos de ordenação devido à sua eficiência e estabilidade. Ele é considerado um algoritmo estável, o que significa que preserva a ordem relativa dos elementos iguais na lista ordenada.

Em python o algoritmo tem o seguinte código:

```
import random
import time
```

CURSO DE CIÊNCIA DE DADOS

```
def merge_sort(arr):
    if len(arr) <= 1:
        return arr
    else:
        mid = len(arr) // 2
        left = arr[:mid]
        right = arr[mid:]
        left = merge_sort(left)
        right = merge_sort(right)
        return merge(left, right)
def merge(left, right):
    result = []
    i = 0
    j = 0
    while i < len(left) and j < len(right):
        if left[i] < right[j]:
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1
    result += left[i:]
```

CURSO DE CIÊNCIA DE DADOS

```
    result += right[j:]
    return result
# Gerar lista aleatória de 1000 elementos
my_list = [random.randint(0, 10) for i in range(10)]
# Medir o tempo de execução do algoritmo
start_time = time.time()
# Ordenar a lista usando merge_sort
sorted_list = merge_sort(my_list)
# Calcular o tempo de execução
elapsed_time = time.time() - start_time
# Imprimir a lista ordenada e o tempo de execução
print("Lista ordenada:", sorted_list)
print("Tempo de execução:", elapsed_time, "segundos")
```

2.9 HeapSort

O Heapsort é um algoritmo de classificação (ou ordenação) em que os elementos de uma lista são organizados em uma estrutura de dados chamada "heap" (em português, "monte"), e em seguida removidos da heap e colocados na lista ordenada. O heap é uma estrutura de dados em forma de árvore binária que obedece a uma propriedade especial: cada nó é maior (ou menor) que seus filhos.

CURSO DE CIÊNCIA DE DADOS

O processo de ordenação começa com a criação de um heap a partir da lista desordenada, em que cada elemento é inserido na estrutura de dados em sua posição apropriada, seguindo a propriedade especial da heap. Em seguida, o elemento raiz (o maior ou o menor, dependendo do tipo de heap utilizado) é removido e colocado na lista ordenada. Esse processo é repetido até que todos os elementos da heap tenham sido removidos e colocados na lista ordenada.

O Heapsort é considerado um algoritmo eficiente, com uma complexidade de tempo de O(n log n) no melhor e também no pior caso, em que n é o número de elementos na lista a ser ordenada. No entanto, seu uso prático é limitado em relação a outros algoritmos de classificação mais simples, como o Bubble Sort e o Insertion Sort, devido à sua complexidade e à necessidade de implementar uma estrutura de dados adicional. Como mostrado a seguir:

```
import time

def heapify(arr, n, i):
    largest = i
    left = 2 * i + 1
    right = 2 * i + 2
    if left < n and arr[i] < arr[left]:
        largest = left
    if right < n and arr[largest] < arr[right]:
        largest = right
    if largest != i:
        arr[i], arr[largest] = arr[largest], arr[i]
```

```
        heapify(arr, n, largest)
def heap_sort(arr):
    n = len(arr)

    for i in range(n // 2 - 1, -1, -1):
        heapify(arr, n, i)
    for i in range(n - 1, 0, -1):
        arr[0], arr[i] = arr[i], arr[0]
        heapify(arr, i, 0)
    return arr
import random
arr = [random.randint(1, 10) for i in range(10)] # lista de 10 elementos aleatórios
start_time = time.time()
sorted_arr = heap_sort(arr)
end_time = time.time()
print(f"Tempo de execução: {end_time - start_time:.5f} segundos")
print(sorted_arr)
```

3 RELATO DAS ATIVIDADES DESENVOLVIDAS

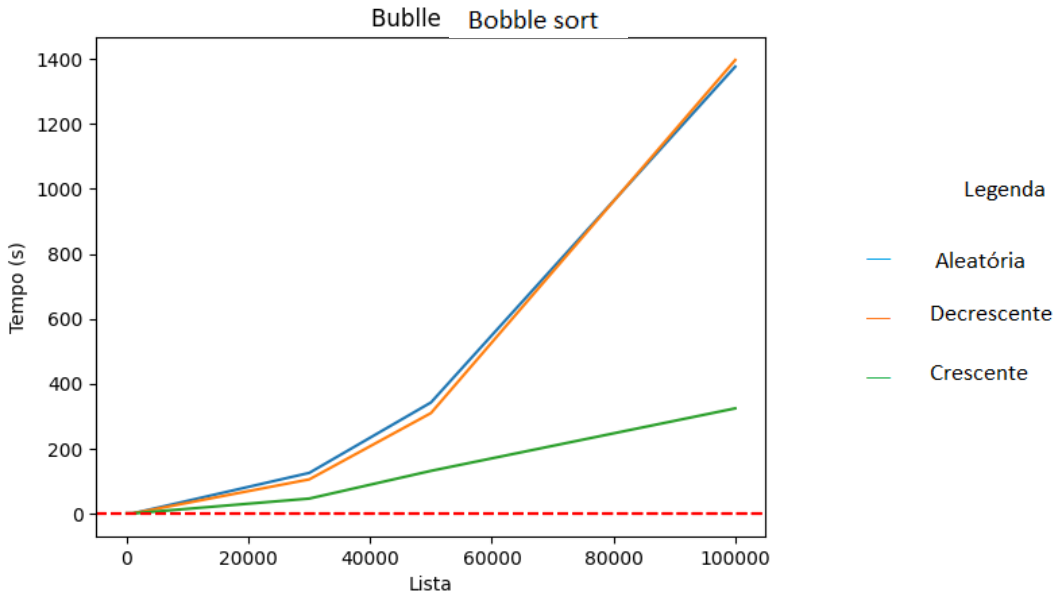
Foi realizado uma análise exploratória implementado um conjunto de algoritmos de ordenação para entradas de listas com tamanhos de 10, 1000, 30000, 50000 e 100000 consecutivamente organizadas em ordem aleatória, decrescente e ordenada.

CURSO DE CIÊNCIA DE DADOS

No algoritmo de ordenação Bubble sort (versão original sem melhorias) é possível observar de acordo com o gráfico gerado contendo os resultados de sua análise exploratória, que sua execução demanda mais tempo para valores decrescentes, cujo ultrapassam 1400 segundos de execução para entradas maiores que 100000, ou seja esse é seu pior caso.

É possível analisar também que o menor tempo de execução que o algoritmo teve para ordenar uma lista foi para realizar essa tarefa em listas crescentes demandando um tempo de execução em torno de 400 segundos para entradas de 100000 elementos, sendo este seu melhor caso.

De acordo com sua complexidade esse algoritmo possui como pior e melhor caso O(n2).

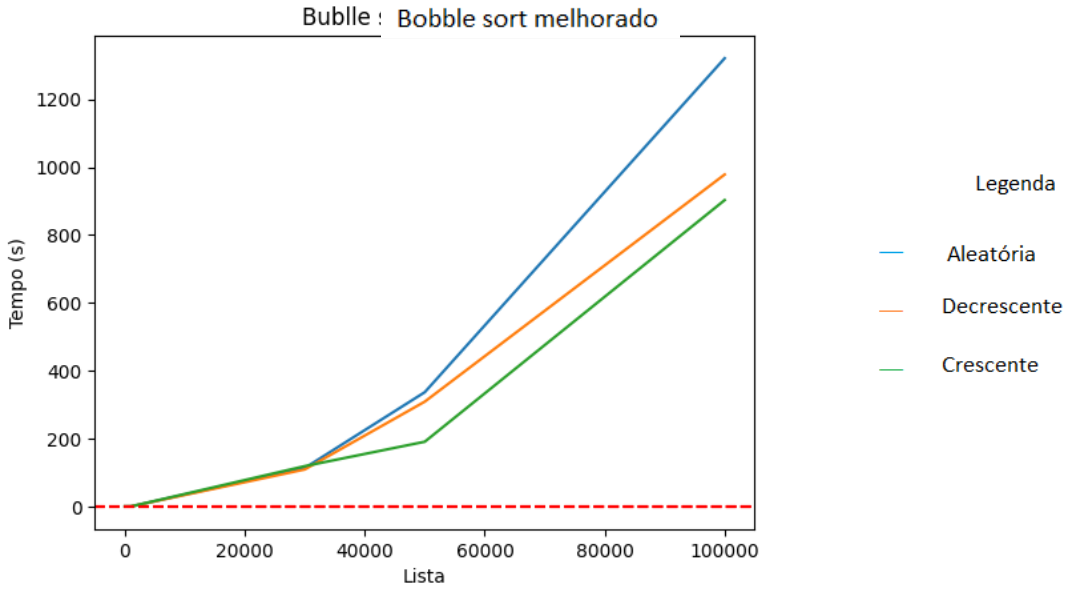


CURSO DE CIÊNCIA DE DADOS

No algoritmo de ordenação Bubble sort (versão que verifica se o vetor já está ordenado) é possível observar de acordo com o gráfico gerado contendo os resultados de sua análise exploratória que sua execução demanda mais tempo para valores aleatórios, que ultrapassam 1200 segundos de execução para entradas maiores que 100000, ou seja esse é seu pior caso.

É possível analisar também que o menor tempo de execução que o algoritmo teve para ordenar uma lista foi para realizar essa tarefa em listas crescentes demandando um tempo de execução em torno de 800 segundos para entradas de 100000 elementos, sendo este seu melhor caso.

De acordo com sua complexidade esse algoritmo possui como pior caso $O(n^2)$ e melhor caso $O(n)$.

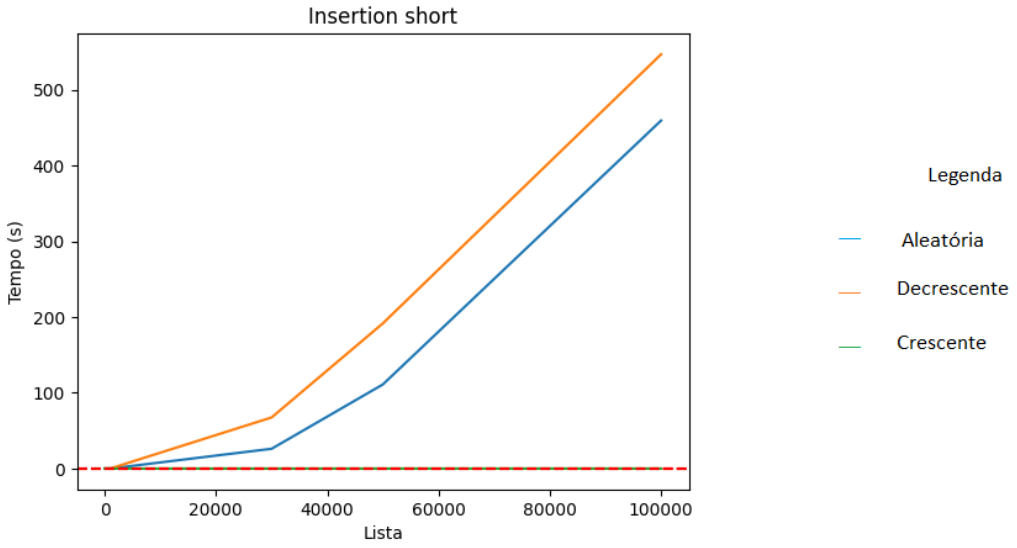


CURSO DE CIÊNCIA DE DADOS

No algoritmo de ordenação Insertion Sort é possível observar de acordo com o gráfico gerado contendo os resultados de sua análise exploratória que sua execução demanda mais tempo para valores decrescentes, que ultrapassam 500 segundos de execução para entradas maiores que 100000, ou seja, esse é seu pior caso.

É possível analisar também que o menor tempo de execução que o algoritmo teve para ordenar uma lista foi para realizar essa tarefa em listas crescentes demandando um tempo de execução em torno de 0 segundos para entradas de 100000 elementos, sendo este seu melhor caso.

De acordo com sua complexidade esse algoritmo possui como pior caso $O(n^2)$ e melhor caso $O(n)$.



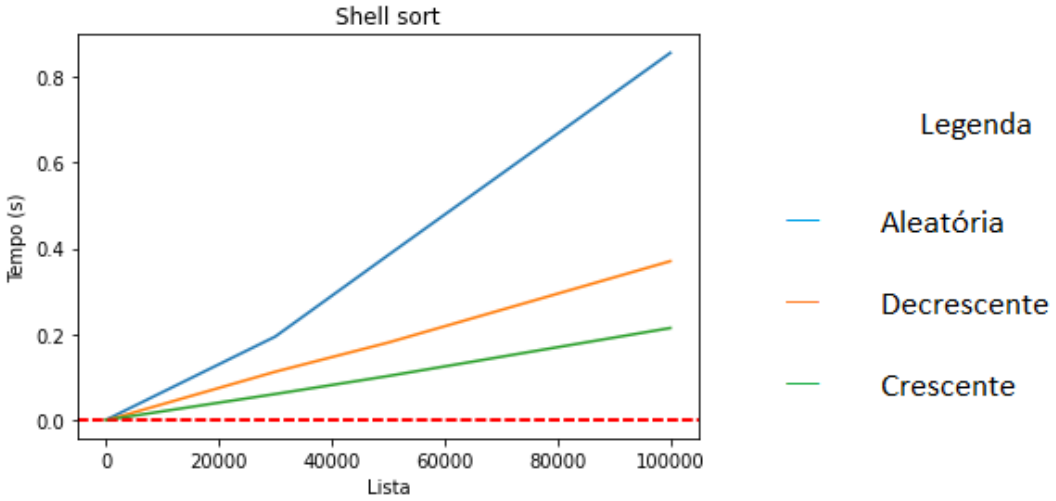
No algoritmo de ordenação Shell Sort é possível observar de acordo com o gráfico gerado contendo os resultados de sua análise exploratória que sua execução

CURSO DE CIÊNCIA DE DADOS

demanda mais tempo para valores aleatórios, cujo ultrapassam 0.8 segundos de execução para entradas maiores que 100000, ou seja, esse é seu pior caso.

É possível analisar também que o menor tempo de execução que o algoritmo teve para ordenar uma lista foi para realizar essa tarefa em listas crescentes demandando um tempo de execução em torno de 0.2 segundos para entradas de 100000 elementos, sendo este seu melhor caso.

De acordo com sua complexidade de pior e melhor caso, o algoritmo possui um comportamento assintótico dado por $O(n \log_2 n)$.

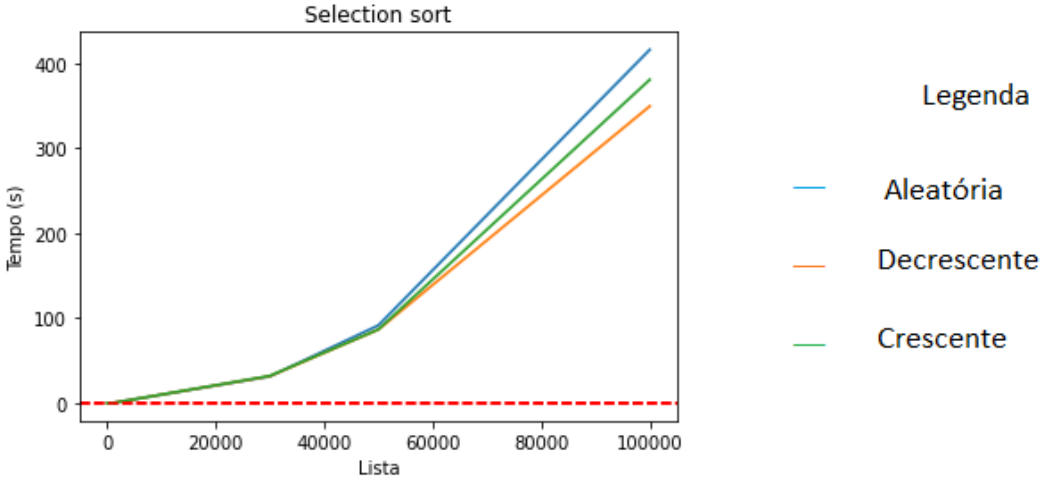


No algoritmo de ordenação Selection Sort é possível observar de acordo com o gráfico gerado contendo os resultados de sua análise exploratória que sua execução demanda mais tempo para valores aleatórios, cujo ultrapassam 400 segundos de execução para entradas maiores que 100000, ou seja, esse é seu pior caso.

CURSO DE CIÊNCIA DE DADOS

É possível analisar também que o menor tempo de execução que o algoritmo teve para ordenar uma lista foi para realizar essa tarefa em listas decrescentes demandando um tempo de execução em torno de 349 segundos para entradas de 100000 elementos, sendo este seu melhor caso.

De acordo com sua complexidade de pior e melhor caso, o algoritmo possui um comportamento assintótico dado por $O(n^2)$.



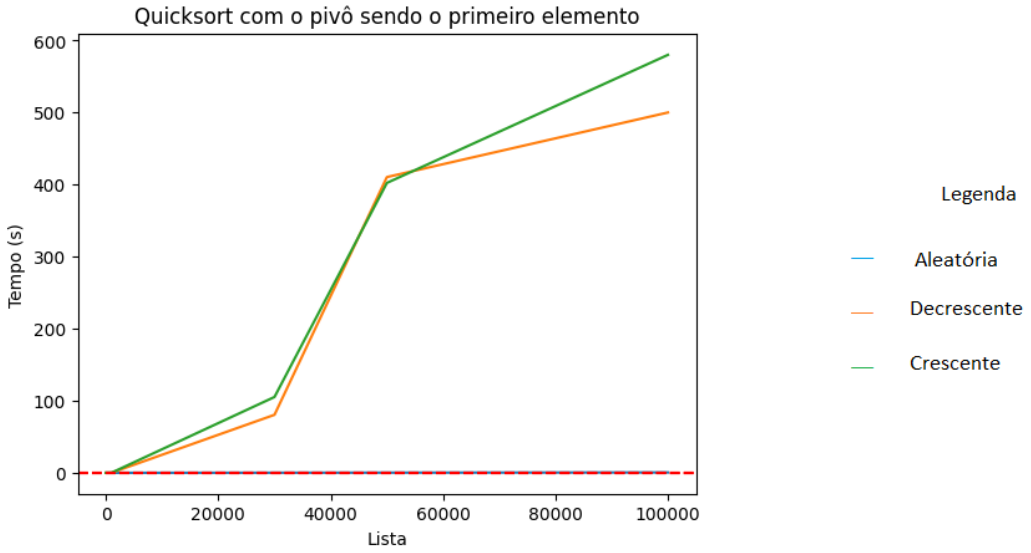
No algoritmo de ordenação Quicksort com pivô sendo o primeiro elemento da lista é possível observar de acordo com o gráfico gerado contendo os resultados de sua análise exploratória que sua execução demanda mais tempo para valores crescentes, cujo ultrapassam 500 segundos de execução para entradas maiores que 100000, ou seja esse é seu pior caso.

É possível analisar também que o menor tempo de execução que o algoritmo teve para ordenar uma lista foi para realizar essa tarefa em listas aleatórias demandando

CURSO DE CIÊNCIA DE DADOS

um tempo de execução em torno de 0,5 segundos para entradas de 100000 elementos, sendo este seu melhor caso.

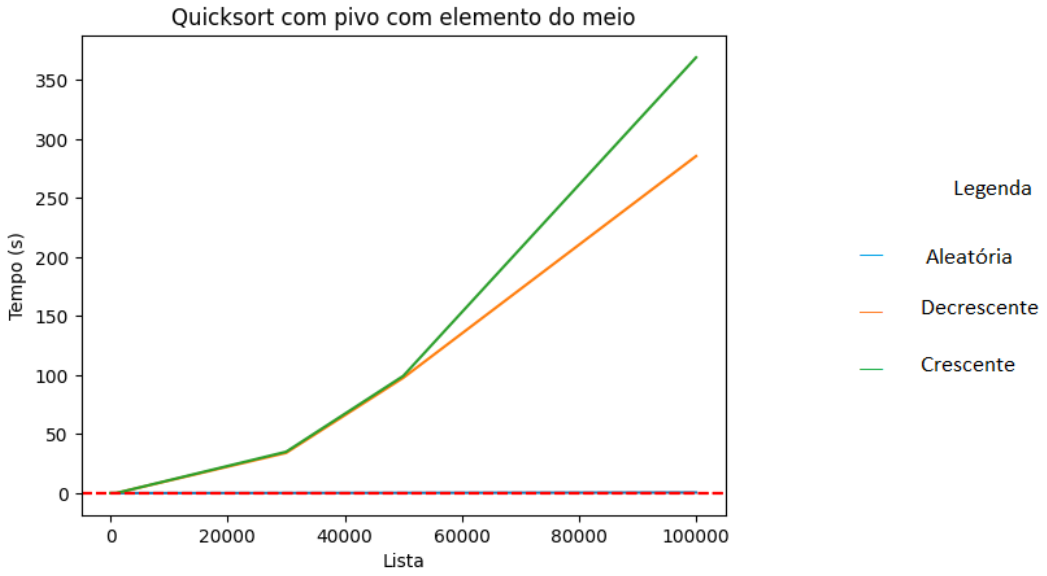
De acordo com sua complexidade o algoritmo possui um comportamento assintótico dado por $O(n^2)$ para seu pior caso, e, um comportamento dado por O(n log n) para seu melhor caso.



Na ordenação Quicksort (com pivô sendo o primeiro central da lista - partição) observa-se que a lista entrada que demandou maior tempo é a Crescente, tendo seu maior tempo com a maior entrada de 100000, e de distanciando consideravelmente de outra lista a partir de 50000 entradas.

É possível analisar também que a lista de entrada com melhor tempo foi a Aleatória, que se manteve constante no tempo de zero em todas as entradas. Sua complexidade no melhor caso e pior caso O(n log n).

CURSO DE CIÊNCIA DE DADOS

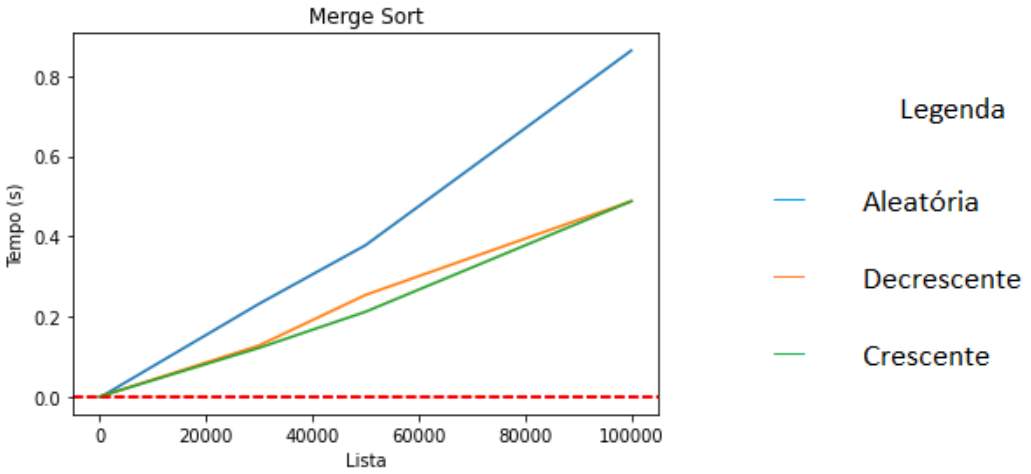


Na ordenação Merge Sort, observa-se que a lista de entrada que demandou o maior tempo é a aleatória, que teve uma elevação acentuada em relação às demais listas de entradas a partir de 50000, e piorando com o aumento de entradas, tendo seu pior caso em 100000 entradas.

É possível analisar também que a lista de entradas com melhor tempo de execução foi a crescente, onde ela exibe um tempo ligeiramente melhor que a decrescente, sendo seu melhor caso com 50000 entradas.

Esse algoritmo possui a complexidade de tempo no pior e também no melhor caso dado por O(n log n).

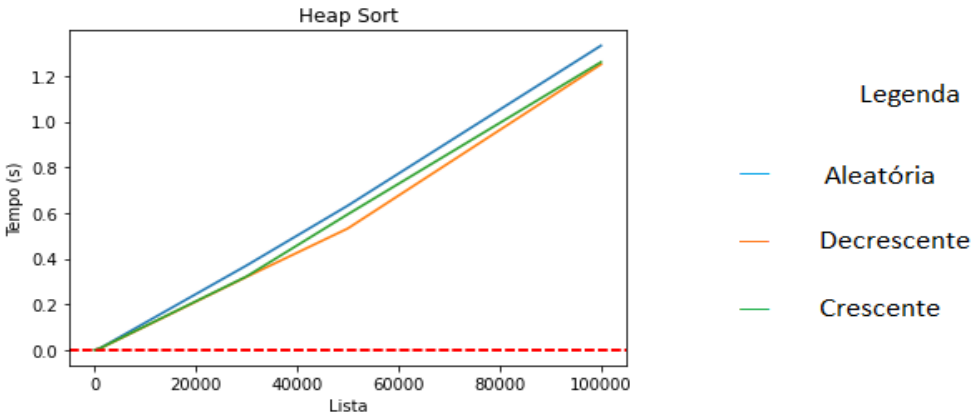
CURSO DE CIÊNCIA DE DADOS



Na ordenação Heap Sort observa-se o desempenho das listas muito próximas, entretanto a decrescente ainda possui leve vantagem, principalmente a 50000 mil entradas que demanda um tempo menor.

E com ligeira desvantagem a lista aleatória, que a partir de 20000 entradas passa a levar um tempo gradativamente maior das demais.

Com complexidade de tempo de O(n log n) no pior e no melhor caso, é considerado um algoritmo eficiente, embora seja limitado em relação a algoritmos mais simples, devido a sua necessidade de implementar uma estrutura de dados adicional.



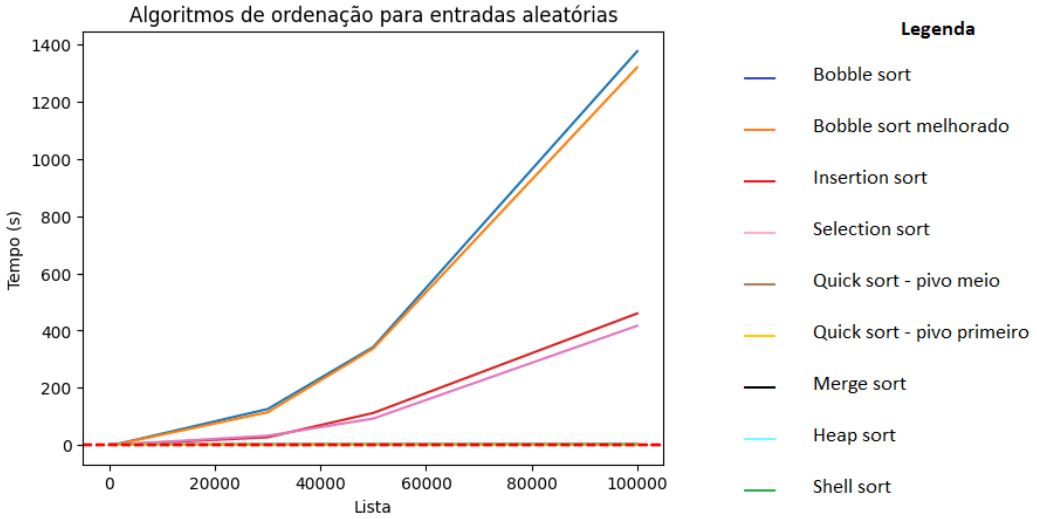
4 RESULTADOS E CONCLUSÕES

Ao analisar os gráficos com o tempo de execução dos nove algoritmos de ordenação, sendo respectivamente Bubble Sort (versão original sem melhorias), Bubble Sort (versão que verifica se o vetor já está ordenado), Insertion Sort, Shell Sort, Selection Sort, Quicksort (com pivô sendo o primeiro elemento da lista - partição), Quicksort (com pivô sendo o primeiro central da lista - partição), Merge Sort e Heapsort para entradas de cinco listas crescentes, decrescentes e aleatórias, com valores de 10, 1000, 30000, 50000 e 100000 podemos observar que os melhores algoritmos de ordenação para listas aleatórias, que executam as mesmas em um tempo estimado de aproximadamente 0,5 segundos são, de modo respectivo o Quicksort com pivô sendo o primeiro central da lista, Quicksort com pivô sendo o primeiro elemento da lista, Shell sort, Merge sort e o Heap sort.

Consequentemente, o pior algoritmo de ordenação para as mesmas entradas são os

CURSO DE CIÊNCIA DE DADOS

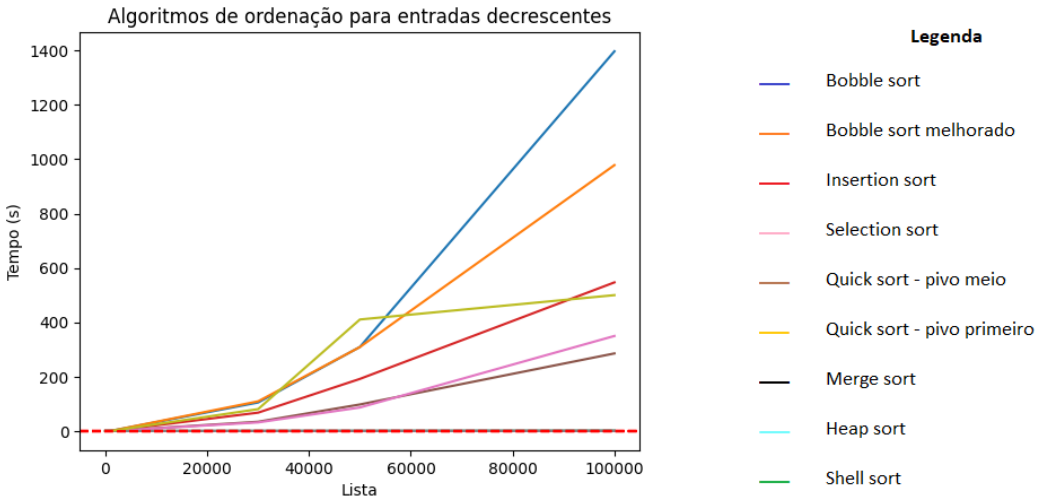
algoritmos Bubble sort e Bubble sort melhorado, levando em torno de 1400 segundos.



Já para entradas das mesmas listas em ordem decrescente, os algoritmos que se destacam levando menos tempo para ordená-las em aproximadamente 0,37 até 1 segundo são, de modo respectivo o Shell sort, Merge sort e o Heap sort.

Para a mesma lista, os piores algoritmos de ordenação são Bubble sort, Bubble sort melhorado, Insertion sort e Selection sort, levando em torno de 400 a 1400 segundos.

CURSO DE CIÊNCIA DE DADOS



Em entradas das mesmas listas em ordem crescente, os algoritmos que levaram menos tempo para ordená-las de aproximadamente 0,2 a 0,4 segundos são, de modo respectivo o Shell sort, Insertion sort e Merge sort.

Já os piores algoritmos, levando em torno de 600 a 800 segundos são o Bubble sort e o Quicksort com pivô sendo o primeiro central da lista.

CURSO DE CIÊNCIA DE DADOS

Contudo, podemos concluir com base nos resultados obtidos através desses experimentos de análise exploratória que existem vários algoritmos de ordenação com diferentes complexidades e desempenhos para diferentes tipos de entradas. A escolha do melhor algoritmo depende do tamanho da entrada, do tipo de dados a serem ordenados e do contexto em que o algoritmo será utilizado, além disso o melhor algoritmo de ordenação depende do contexto específico em que o mesmo será utilizado, como restrições de memória ou de tempo de execução, dessa maneira, é importante considerar todos os fatores relevantes ao escolher um algoritmo de ordenação para um problema específico.

CURSO DE CIÊNCIA DE DADOS

REFERÊNCIAS

SHELL sort. Wikipédia. Disponível em: https://pt.wikipedia.org/wiki/Shell_sort. Acesso em: 20 mar. 2023.

DANIEL, V. Treinaweb. Disponível em: https://www.treinaweb.com.br/blog/conheca-os-principais-algoritmos-de-ordenacao#:~:text=Criado%20por%20Donald%20Shell%20em,determinar%20o%20ponto%20de%20inser%C3%A7%C3%A3o. Acesso em: 20 mar. 2023.

Caderno geek. Disponível em: https://cadernogeek.wordpress.com/tag/shell-sort/. Acesso em: 20 mar.2023.

GUSTAVO, P. Pantuza. Disponível em: https://blog.pantuza.com/artigos/o-algoritmo-de-ordenacao-quicksort. Acesso em: 25 mar. 2023.

QUICKSORT. Wikipédia. Disponível em: https://pt.wikipedia.org/wiki/Quicksort. Acesso em: 25 mar. 2023.

BRUNO. Devmedia. Disponível em: https://www.devmedia.com.br/algoritmos-de-ordenacao-analise-e-comparacao/28261. Acesso em: 23 mar.2023.

BUBBLE sort. Wikipédia. Disponível em: https://pt.wikipedia.org/wiki/Bubble_sort. Acesso em: 20 mar. 2023.

JOÃO, A.B. Github. Disponível em: https://joaoarthurbm.github.io/eda/posts/insertion-sort/. Acesso em: 23 mar.2023.

JOÃO, A.B. Github. Disponível em https://joaoarthurbm.github.io/eda/posts/merge-sort/da/posts/merge-sort/. Acesso em: 23 mar.2023.