Universidade Estadual do Piauí - UESPI
Curso.: Ciência da Computação
Disciplina.: Projeto e Análise de Algoritmos
Professor.: Marcus Vinícius Carvalho
Bloco 4 - 2022.1

Leonardo Ferreira de Cerqueira
Lázaro Geiel Sousa Costa

Trabalho Pratico de Análise de Algoritmos

Teresina, 19 de novembro de 2023

Introdução:

Trabalho Prático sobre Análise de Algoritmos: realizamos uma avaliação comparativa entre algoritmos considerados não eficientes, tais como BubbleSort, SelectionSort e InsertionSort, e algoritmos eficientes como QuickSort, HeapSort e MergeSort. A implementação dos algoritmos foi conduzida utilizando a linguagem de programação Python, junto com a estrutura de dados conhecida como lista em Python. Essa implementação envolveu sequências de dados com elementos variando em tamanhos de 10, 100, 1000, 10000, 100000 e 1000000.

Implementação:

Na análise dos dados, optamos por utilizar a estrutura de lista em Python, fundamentando nossa escolha em sua natureza flexível e dinâmica, que possibilita o armazenamento eficiente de coleções ordenadas de elementos. O estudo incorporou a inclusão de três variáveis, destinadas a avaliar as comparações, as atribuições (considerando apenas aquelas provenientes da lista), e o tempo de execução do código, empregando, para tanto, a biblioteca ''time'' do Python.

Ex:

```
#n= 10 elementos

lista_ordenada = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
lista_inversamente_ordenada = [10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
lista_quase_ordenada = [1, 2, 4, 6, 5, 3, 7, 8, 9, 10]
lista_aleatoria = [76, 27, 27, 35, 8, 25, 77, 14, 49, 20]
```

BubbleSort:

O algoritmo de ordenação por bolha, também conhecido como “bubble sort” em inglês, é uma abordagem simples para organizar elementos. Ele compara repetidamente pares de elementos adjacentes e os troca se estiverem fora de ordem. O processo continua até que não seja mais necessária nenhuma troca, indicando que a lista está ordenada.

A implementação foi realizada como um método em Python, onde recebe uma lista e utiliza dois loops for para percorrer todos os elementos, efetuando comparações e trocas entre pares de elementos. O alcance desses loops vai de 0 até n-1, onde a troca ocorre após verificar se a comparação entre dois elementos da lista é verdadeira. Caso positivo, o algoritmo realiza a troca, contribuindo para a ordenação da lista.

```
def bubble_sort(lista):
    n = len(lista)
    for j in range(n-1):
        for i in range(n-1):
            if lista[i] > lista[i+1]:
                # troca de elementos nas posições i e i+1
                lista[i], lista[i+1] = lista[i+1], lista[i]


    return lista
```

InsertionSort:

O algoritmo de ordenação por inserção, conhecido como “insertion sort” em inglês, é uma abordagem simples que constrói uma sequência ordenada de elementos, um de cada vez. A ideia central é percorrer a lista e, para cada elemento, inseri-lo no lugar adequado entre os elementos já ordenados.

A implementação é realizada como um método em Python, onde recebe uma lista e utiliza dois loops for. No primeiro loop, é selecionada uma chave contendo o primeiro elemento da lista. No segundo loop, o algoritmo realiza comparações, deslocando os elementos maiores para a direita até encontrar a posição correta para inserir a chave, fazendo a construção da lista ordenada, elemento por elemento.

```python
def insertion_sort(lista):
    n = len(lista)
    for i in range(1, n):
        #pega a chave, primeiro elemento da lista
        chave = lista[i]
        j = i - 1
        #faz a busca pra saber qual é elemento é maior que a chave
        #e deslocando eles para a direita da chave
        while j >= 0 and lista[j] > chave:
            lista[j+1] = lista[j]
            j = j - 1
        #no fim coloca a chave na sua posição certa
        lista[j+1] = chave

    return lista
```

SelectionSort:
O algoritmo de ordenação por seleção, ou “selection sort” em inglês, é uma abordagem simples de ordenação que opera selecionando repetidamente o menor (ou maior, dependendo da ordem desejada) elemento da lista não ordenada e trocando-o com o primeiro elemento não ordenado. Esse processo é repetido para o restante da lista não ordenada.

A implementação é realizada como um método em Python, no qual a função recebe uma lista e utiliza dois loops for. No primeiro loop, o algoritmo inicia selecionando o primeiro elemento da lista, considerando seu índice como o menor. No segundo loop, cada elemento é comparado, e o índice é trocado caso seja encontrado um elemento menor do que o inicial. Ao final, o menor elemento é inserido na sua posição apropriada, contribuindo para a ordenação progressiva da lista.

```python
def selection_sort(lista):
    n = len(lista)

    for j in range(n-1):
        #seleciona o primeiro elemento da lista como minimo
        min_index = j
        for i in range(j, n):
            #realiza a comparação pra saber qual é o menor elemento da lista
            if lista[i] < lista[min_index]:
                min_index = i

        #realiza a troca se o elemento da posição j for maior que o elemento da busca anterior
        #que o indice ficou armazenado na variavel min_index
        if lista[j] > lista[min_index]:
            aux = lista[j]
            lista[j] = lista[min_index]
            lista[min_index] = aux

    return lista
```

QuickSort:
O Quicksort destaca-se como um algoritmo de ordenação eficiente e amplamente adotado. Classificado na categoria de algoritmos de ordenação por comparação, ele fundamenta-se na estratégia de divisão e conquista.

Implementado como um método em Python, o algoritmo recebe uma lista para realizar a ordenação, começando do índice 0 até n-1. Para listas com mais de um elemento, a função incorpora a 'partition', chamada para identificar o pivô, geralmente o último elemento da lista. A 'partition' percorre os elementos, organizando aqueles menores que o pivô à esquerda e os maiores à direita, assegurando que o pivô fique em sua posição já ordenada. Em seguida, o Quicksort emprega recursão, chamando-se recursivamente nas sublistas à esquerda e à direita do pivô, efetuando assim a ordenação de maneira eficaz.

```python
def quicksort(lista, inicio=0, fim=None):
    if fim is None:
        fim = len(lista)-1
    if inicio < fim:
        p = partition(lista, inicio, fim)
        # faz a chamada recursivamente na sublista à esquerda (menores)
        quicksort(lista, inicio, p-1)
        # recursivamente na sublista à direita (maiores)
        quicksort(lista, p+1, fim)

def partition(lista, inicio, fim):
    pivot = lista[fim]
    i = inicio
    for j in range(inicio, fim):
        # j sempre avança, pois representa o elementa em análise
        # e delimita os elementos maiores que o pivô
        if lista[j] <= pivot:
            lista[j], lista[i] = lista[i], lista[j]
            # incrementa-se o limite dos elementos menores que o pivô
            i = i + 1
    lista[i], lista[fim] = lista[fim], lista[i]
    return i
```

HeapSort:

O Heapsort é um algoritmo de ordenação que se fundamenta em uma estrutura de dados chamada heap. Uma heap é uma árvore binária especial onde a chave de cada nó é maior (ou menor, dependendo do tipo de heap) do que as chaves de seus filhos. O Heapsort tira proveito das propriedades da heap para realizar a ordenação com um número eficiente de comparações e trocas.

Implementado como método em Python, a função heapsort recebe uma lista e realiza a ordenação padrão de toda a lista. No primeiro loop, a função heapify é chamada, que transforma a lista em um heap máximo, utilizando parâmetros como a lista em questão, o índice do primeiro nó que não é folha, a posição atual i que começa do último índice da lista, em seguida, realiza a operação de heapificação subindo, colocando o maior elemento no topo. Posteriormente, o segundo loop efetua as trocas necessárias, movendo o elemento máximo para a última posição da lista.

Em seguida, a função heapify é novamente chamada para transformar a raiz em heap novamente. Esse processo é repetido até que a lista seja ordenada.

```python
def heapify(arr, n, i):
    # Encontra o maior entre raiz e filhos direito e esquerdo
    largest = i
    l = 2 * i + 1
    r = 2 * i + 2
    # Encontra o maior entre raiz e filhos esquerdo
    if l < n and arr[i] < arr[l]:
        largest = l
    # Encontra o maior entre raiz e filhos direito
    if r < n and arr[largest] < arr[r]:
        largest = r

    # caso o nó não for a maior elemento, é realizado a
    # troca pela maior e continue a heapificar
    if largest != i:
        arr[i], arr[largest] = arr[largest], arr[i]
        heapify(arr, n, largest)


def heapSort(arr):
    n = len(arr)

    # Criar heap máximo
    for i in range(n//2, -1, -1):
        heapify(arr, n, i)

    for i in range(n-1, 0, -1):
        # Troca
        arr[i], arr[0] = arr[0], arr[i]

        # Heapificar o elemento raiz
        heapify(arr, i, 0)
```

MergeSort:
O Mergesort se destaca como um algoritmo de ordenação eficiente que segue a abordagem de “divisão e conquista”. Seu funcionamento consiste em dividir a lista não ordenada em sublistas menores, ordená-las individualmente e, em seguida, mesclá-las para formar uma lista ordenada mais ampla.

Implementado como método em Python, a função mergesort recebe uma lista e realiza a ordenação padrão de toda a lista. Antes de prosseguir, verifica-se se a lista possui mais de um elemento, pois se contiver apenas um ou nenhum, ela já está ordenada. Em seguida, a função é chamada recursivamente, dividindo a lista ao meio em duas sublistas: a primeira vai do início até o meio, e a segunda do meio até o fim. Após essa divisão, a função merge é acionada para combinar as duas sublistas já ordenadas.

Na função merge, são criadas as listas esquerda (do início até o meio) e direita (do meio até o fim), juntamente com os índices correspondentes. A comparação entre os elementos dessas duas listas é realizada, permitindo a combinação das duas na lista principal. Os índices das duas sublistas são

atualizados progressivamente até que a combinação completa na lista principal seja alcançada. Esse processo assegura a ordenação eficiente e precisa da lista original.

```python
def mergesort(lista, inicio=0, fim=None):
    #seta o fim da lista com o tamanho, caso não seja passado o parametro
    if fim is None:
        fim = len(lista)
    #começa a dividir a lista se for que 1
    if(fim - inicio > 1):
        meio = (fim + inicio)//2
        #faz a chamada recursivamente para o inicio até o meio
        mergesort(lista, inicio, meio)
        #faz a chamada recursivamente para o meio até o fim
        mergesort(lista, meio, fim)
        #combina as duas listas anteriores
        merge(lista, inicio, meio, fim)
        return lista

def merge(lista, inicio, meio, fim):
    #cria duas sublistas
    left = lista[inicio:meio]
    right = lista[meio:fim]
    top_left, top_right = 0, 0
    #realiza a combinação das duas sublistas
    for k in range(inicio, fim):
        #caso todos os elementos da esquerda tenham sido utilizados
        if top_left >= len(left):
            lista[k] = right[top_right]
            top_right = top_right + 1
        #caso todos os elementos da direita tenham sido utilizados
        elif top_right >= len(right):
            lista[k] = left[top_left]
            top_left = top_left + 1
        #caso o topo da esquerda seha menor que da direita, é o elemento da esquerda que
        #entra na lista
        elif left[top_left] < right[top_right]:
            lista[k] = left[top_left]
            top_left = top_left + 1
        #se não, é o elemento da direita que entra na lista
        else:
            lista[k] = right[top_right]
            top_right = top_right + 1
```

Saída e entrada de dados: utilizado uma lista de elementos do tipo inteiro em python, foi implementado 4 métodos para serem gerados listas dos seguintes tipos: ordenada, inversamente ordenada, quase ordenada e aleatória. E criado 4 exclusivas para cada algoritmo de ordenação, e utilizado a biblioteca time do python para calcular o tempo de execução dos algoritmos

```python
import random

def generate_ordered_sequence(size):
    return list(range(1, size + 1))

def generate_reverse_ordered_sequence(size):
    return list(range(size, 0, -1))

def generate_almost_ordered_sequence(size, swap_prob):
    sequence = generate_ordered_sequence(size)
    for i in range(size):
        if random.random() < swap_prob:
            j = random.randint(0, size - 1)
            sequence[i], sequence[j] = sequence[j], sequence[i]
    return sequence

def generate_random_sequence(size, min_val=1, max_val=100):
    return [random.randint(min_val, max_val) for _ in range(size)]
```

Análise de Complexidade

A complexidade dos algoritmos não-eficientes como o bubble sort, insertion sort e selection tem complexidade $O(n^2)$, devido a eles terem um for dentro de outro for que no pior caso vai ter que ir 1 até n nos dois loops for

A complexidade dos algoritmos eficientes como merge sort, heap sort tem complexidade de $O(n*\log n)$. Já o quick sort em seu pior caso tem complexidade de $O(n^2)$

Listagem de testes executados

Tabela algoritmos não-eficientes

Tamanho lista	Tipo lista	Bubble sort	Insertion sort	Selection sort
10	Ordenada	0	0	0
10	Inversamente	0	0	0
10	Quase	0	0	0
10	Aleatória	0	0	0
100	Ordenada	0	0	0
100	Inversamente	0	0	0
100	Quase	0	0	0
100	Aleatória	0	0	0
1000	Ordenada	0,04	0	0,02
1000	Inversamente	0,06	0,02	0,02
1000	Quase	0,05	0,01	0,01
1000	Aleatória	0,05	0,01	0,02
10000	Ordenada	3,9	0	1,43
10000	Inversamente	6,15	2,49	1,44
10000	Quase	5,02	1,09	1,43
10000	Aleatória	5,08	1,21	1,44

Comparando os tempos de execuções dos algoritmos não eficientes, é possível observar que até 100 elementos todos os algoritmos tem tempo de execução bem próximos. Após analisar com mais de 1000 elementos, é observado que o bubble sort começa a ter o pior desempenho entre os 3 algoritmos, sendo bem mais perceptível com 10000 elementos. Foi tentado realizar o teste com uma lista contendo 100000 elementos, mas é inviável usando os algoritmos não eficiente, pois demora muito para serem ordenados

Tabela algoritmos eficientes

Tamanho lista	Tipo lista	Quick sort	Heap sort	Merge sort
10	Ordenada	0	0	0
10	Inversamente	0	0	0
10	Quase	0	0	0
10	Aleatória	0	0	0
100	Ordenada	0	0	0
100	Inversamente	0	0	0
100	Quase	0	0	0
100	Aleatória	0	0	0
1000	Ordenada	Limiterecursao	0	0
1000	Inversamente	Limiterecursao	0	0
1000	Quase	Limiterecursao	0	0
1000	Aleatória	Limiterecursao	0	0
10000	Ordenada	Limiterecursao	0,02	0,01
10000	Inversamente	Limiterecursao	0,02	0,01
10000	Quase	Limiterecursao	0,02	0,02
10000	Aleatória	Limiterecursao	0,02	0,02
100000	Ordenada	Limiterecursao	0,26	0,14
100000	Inversamente	Limiterecursao	0,24	0,16
100000	Quase	Limiterecursao	0,27	0,18
100000	Aleatória	Limiterecursao	0,25	0,17
1000000	Ordenada	Limiterecursao	3,2	1,74
1000000	Inversamente	Limiterecursao	3,03	1,95
1000000	Quase	Limiterecursao	3,64	2,33
1000000	Aleatória	Limiterecursao	3,06	2,08

Comparando os tempos de execuções dos algoritmos, é possível observar que até 10000 elementos o heapsort e mergesort tem tempo de execução bem próximos. Após analisar com mais de 100000 elementos, é observado que o mergesort é mais eficiente que o heapsort, sendo bem mais perceptível ao ir aumento a quantidade de elementos. Comparando com os algoritmos não-eficientes, vemos que os eficiente são bem mais rápidos que eles, o mais lento demora 3 segundos para ordenar uma lista com 1.000.000 de elementos

Conclusão

A implementação desses algoritmos de ordenação proporcionou um aprofundamento no entendimento destes métodos. Através desse trabalho, foi possível aprimorar nossa compreensão sobre o funcionamento de cada algoritmo, percebendo de forma mais clara a lógica de programação utilizada na criação de cada algoritmo. Além disso, adquirimos habilidades práticas na implementação desses algoritmos utilizando a linguagem de programação Python.

O empecilho foi na implementação do quick sort, com lista de tamanho maiores que n=1000 dá o seguinte erro: “RecursionError: maximum recursion depth exceeded in comparison”, pois o limite de recursão em python é 1000

Bibliografia

https://github.com/python-cafe/algorithms/blob/master/sorting/sorting.py

https://www.programiz.com/dsa/heap-sort

https://www.programiz.com/dsa/quick-sort