ALESSANDRO SOARES DA SILVA

MATRICULA: 20231023705

4° LISTA DE ALGORITMO

1 - Implemente o algoritmo de mínimo e máximo simultâneos da seção 9.1 do livro do Cormen, 4a Ed na sua linguagem favorita e mostre através de medição de tempo que é mais rápido que a abordagem não-simultânea para um vetor de entrada suficientemente grande.

```python
import random
import time
import matplotlib.pyplot as plt

def Min_Max_simultaneo(A):
    n = len(A)
    if (n % 2 == 1):                        # verifica se o array possui numero de elementos impares
        min_global = max_global = A[0]      # se sim define o primeiro elemento do array como minimo e maximo
        i = 1                               # inicia o indice do array em 1
    else:
        i = 2                               # inicia o indice do array em 2
        if (A[0] < A[1]):                   # verifica quem é o menor elemento
            min_global = A[0]               # armazena o menor elemento em min_global
            max_global = A[1]               # armazena o maior elemento em max_global
        else:
            min_global = A[1]               # senão inverte as posições
            max_global = A[0]               # senão inverte as posições
    while (i+1 < n):                        # verifica se o indice nao deu estouro no vetor
        if (A[i] > A[i+1]):                 # verificação de qual é maior do par de números
            min_atual = A[i+1]
            max_atual = A[i]
        else:
            min_atual = A[i]
            max_atual = A[i+1]

        if (min_atual < min_global):        # verifica quem é o minimo global
            min_global = min_atual

        if (max_atual > max_global):        # verifica quem é o maximo global
            max_global = max_atual
        i = i + 2

    return min_global, max_global


def Minimo_Maximo(B):                      #
    n = len(B)                             #
    mini = maxi = B[0]                     #
                                           #
    for i in range(1, n):                  #
        if (B[i] < mini):                  # Essa função varre todo o array já faz a comparação
            mini = B[i]                    # com as variaveis mini e maxi, retornando no final
                                           # o maior e o menor elemento
        if (B[i] > maxi):                  #
            maxi = B[i]                    #
                                           #
    return mini, maxi                      #


def mede_tempo(funcao, arr):
    tempo_inicio = time.time()
    minimo, maximo = funcao(arr)
    tempo_fim = time.time()
    tempo_total = (tempo_fim - tempo_inicio)
    return minimo, maximo, tempo_total
```

```python
if __name__ == "__main__":
    sizes = []
    t_min_max = []
    t_simul = []
    i = 1
    max_size = 1000000

    while i < max_size:
        list_size = i
        sizes.append(i)
        i = i * 10
        arr = [random.randint(1, 1000) for _ in range(list_size)]
        minimo1, maximo1, tempo_m = mede_tempo(Minimo_Maximo, arr)
        minimo2, maximo2, tempo_s = mede_tempo(Min_Max_simultaneo, arr)
        t_min_max.append(tempo_m)
        t_simul.append(tempo_s)

    fig, ax = plt.subplots(figsize=(5, 4), layout='constrained')
    plt.plot(sizes, t_min_max, label="modo simples")
    plt.plot(sizes, t_simul, label="modo simultâneo")
    plt.title("Comparação entre modo simples e modo simultâneo")
    plt.xlabel("tamano do array")
    plt.ylabel("Tempo em (s)")
    plt.legend()
    plt.grid(True)
    plt.show()
```

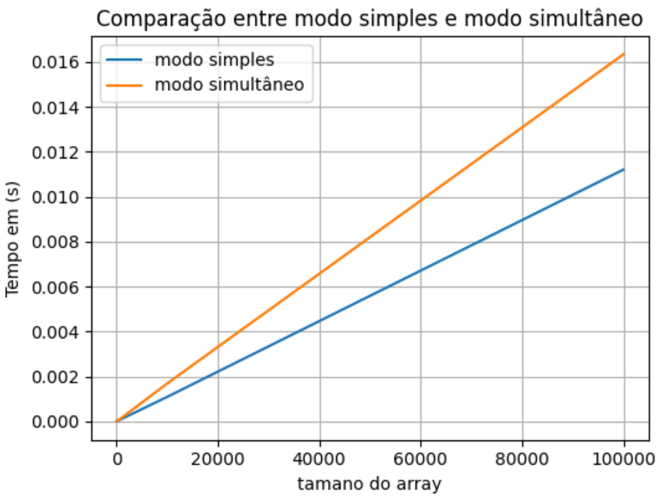

Gráfico 1

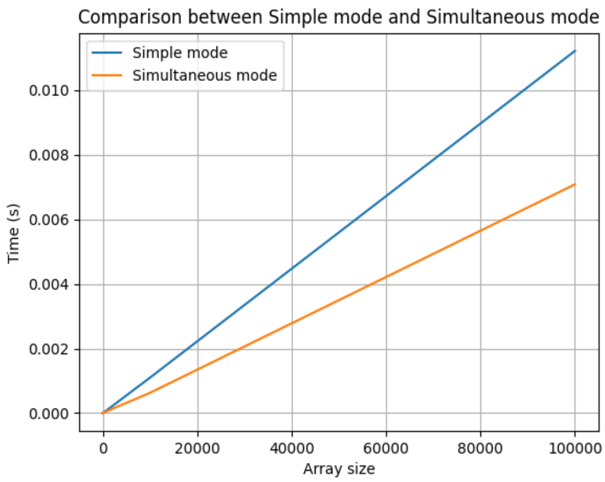

Gráfico 2

Observações: Teoricamente, o algoritmo de min_max simultâneo é mais rápido do que a abordagem não simultânea, no entanto não foi o resultado que obtive nas análises, como se pode verificar no gráfico 1, e como foi discutido em sala, a influência dos if’s (saltos) no algoritmo tem papel fundamental nesse resultado, pois a arquitetura de computadores tem um fluxo de execução que ocorre em fila, de modo que, quando o programa carrega uma operação do tipo, existe uma quebra de linha de execução do pipeline, e os dados já carregados são perdidos sendo necessário carregar novamente as informações, e isso custa tempo. Nós temos no pior caso 5 testes de condições no modo simultâneo, contra 2 no min_max não simultâneo.

Realizamos mais testes, e o nosso colega Dionísio nos mostrou uma implementação otimizada, que demonstrou que o min_max simultâneo de fato é mais rápido, como mostra o gráfico 2. No entanto resolvi manter os resultados do meu experimento, pois apesar de entender a forma como o colega o fez, não consegui replicar sua implementação.

2. Implemente os algoritmos de seleção aleatória e seleção das seções 9.2 e 9.3 do livro do Cormen, 4a Ed., e realize experimentos numéricos para demonstrar em quais casos um tem vantagens com relação ao outro.

O algoritmo Randomized Select é uma variação do algoritmo QuickSort e é usado para encontrar o k-ésimo menor elemento em um array não ordenado. Ele funciona selecionando aleatoriamente um pivô, particionando o array em torno desse pivô e, em seguida, recursivamente selecionando a partição correta onde o k-ésimo elemento está localizado.

```python
import numpy
import time
import random
import matplotlib.pyplot as plt
import random

def randomized_select(arr, menor, maior, k):
    if menor == maior:                                      # verifica se o array tem apenas um elemento
        return arr[menor]                                   # se for o caso retorna o valor unico do array

    pivot_index = random_partition(arr, menor, maior)       # chamada da função random_partition
    posicao = (pivot_index - menor) + 1                     # subitrai do menor valor do array e soma com um
                                                            # (questão do vetor começar em 0)
    if k == posicao:                                        # se posicao for igual ao pivot
        return arr[pivot_index]                             # retorna o pivo como resposta
    elif k < posicao:                                       # k está a esquerda
        return randomized_select(arr, menor, pivot_index - 1, k) # trabalha com o array a esquerda do pivo
    else:
        return randomized_select(arr, pivot_index + 1, maior, k - posicao)  # trabalha com o array a direita do pivo

def random_partition(arr, menor, maior):
    pivot_index = random.randint(menor, maior)              # seleciona um valor randômico e atribui a pivot
    arr[pivot_index], arr[maior] = arr[maior], arr[pivot_index]  # troca o o valor do pivot pelo ultimo numero do array
    return partition(arr, menor, maior)                     # chama a função partition()

def partition(arr, menor, maior):
    pivot = arr[maior]                                      # carrega o valor da ultima posição em pivot
    i = menor - 1                                           # seta o valor de i em -1

    for j in range(menor, maior):                           # percorrer o array
        if arr[j] <= pivot:                                 # avalia se o valor e menor que o pivo
            i += 1                                          # incrementa o i
            arr[i], arr[j] = arr[j], arr[i]                 # faz aa troca no sentido de ordenar o array
                                                            # depois de avaliar todos os valores que são menores que
    arr[i + 1], arr[maior] = arr[maior], arr[i + 1]         # o pivo, coloca o pivo um uma posição a frente do indice
    return i + 1                                            # i, dividindo o array em valore menore e maiores que o piv
                                                            # retornando o indice da nova posição do pivot_index

def select(arr, menor, maior, k):
    if menor == maior:
        return arr[menor]
    pivot_index = deterministic_partition(arr, menor, maior)
    posicao = pivot_index - menor + 1
    if k == posicao:
        return arr[pivot_index]
    elif k < posicao:
        return select(arr, menor, pivot_index - 1, k)
    else:
        return select(arr, pivot_index + 1, maior, k - posicao)

def deterministic_partition(arr, menor, maior):

    groups = [arr[i:i+5] for i in range(menor, maior + 1, 5)]      # Divide o array em grupos de 5 elementos
    medians = [sorted(group)[len(group) // 2] for group in groups] # Calcula a mediana de cada grupo
    median_of_medians = select(medians, 0, len(medians) - 1, len(medians) // 2)    # Encontra a mediana das medianas
    pivot_index = arr.index(median_of_medians)            # Encontra o índice da mediana das medianas no array original
    arr[pivot_index], arr[maior] = arr[maior], arr[pivot_index] # Move o pivô para o final do array

    return partition(arr, menor, maior)

def mede_tempo(funcao, arr, menor, maior, k):
    tempo_inicio = time.time()
    minimo = funcao(arr, menor, maior, k)
    tempo_fim = time.time()
    tempo_total = (tempo_fim - tempo_inicio)
    return minimo, tempo_total
```

Já o Select é um algoritmo determinístico para encontrar o k-ésimo menor elemento em um array não ordenado.

```
if __name__ == "__main__":

    # o algoritmo não funciona com valores 0
    # e pra valores repetidos conta como um elemento
    # Ex: se for pedido o segundo elemento e o array
    # for [1,1,2,3,5] ele devolve 1, que é o segundo elemento
    sizes = []
    t_random_select = []
    t_select = []
    i = 1
    k = 5
    menor = 0
    max_size = 10000

    while i < max_size:
        list_size = i
        sizes.append(i)
        i = i * 10
        arr = [random.randint(1, 100) for _ in range(list_size)]
        maior = len(arr) - 1
        minimo1, tempo_random_select = mede_tempo(randomized_select, arr, menor, maior, k)
        minimo2, tempo_select = mede_tempo(select, arr, menor, maior, k-1)
        t_random_select.append(tempo_random_select)
        t_select.append(tempo_select)

    fig, ax = plt.subplots(figsize=(5, 4), layout='constrained')
    plt.plot(sizes, t_random_select, label="random_select")
    plt.plot(sizes, t_select, label="select")
    plt.title("Comparação select e random_select")
    plt.xlabel("tamano do array")
    plt.ylabel("Tempo em (s)")
    plt.legend()
    plt.grid(True)
    plt.show()
```

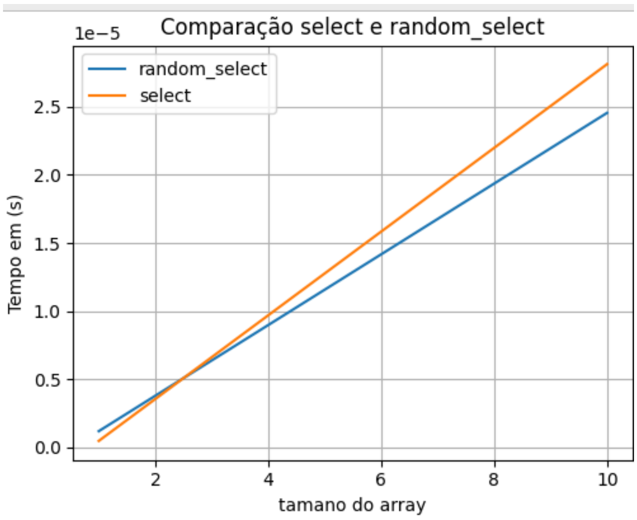

Gráfico 1

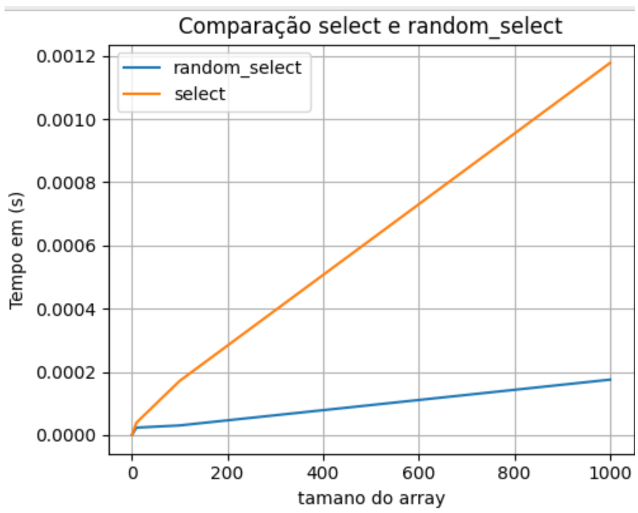

Gráfico 2

Resultados e conclusões: A escolha aleatória do pivô ajuda a evitar o pior casode seleção, tornando o Select Randomized um algoritimo eficiente em média. No entanto, tempo de execução ainda pode ser afetado por situações extremas, mas a probabilidade tende a baixar a medida que aumentamos o tamanho do array como é visualizado no gráfico 2. Em resumo, o Select é uma boa técnica embora mais complexa de implementar, é efieciente para arrays pequenos, conforme início do gráfico 1, quando se deseja o caso de tempo médio a melhor escolha é select randômico.

3. Implemente o algoritmo da mediana ponderada e use-o para resolver o item *e* do Problema 9-3 do Cormen, 4a Ed.

```python
import numpy as np
import time
import random
import matplotlib.pyplot as plt
import random

def mediana_ponderada(pontos, pesos):
    # combina os dados e as poderações em pares
    dado_pesos = list(zip(pontos, pesos))

    # Ordena os pares com base nas ponderações
    dado_pesos.sort(key=lambda x: x[1])
    pesos_totais = sum(pesos)

    if len(pesos) % 2 == 1:
        # Caso impar : Encontra o valor no meio
        ponto_medio = (pesos_totais / 2)mediana ponderada
        peso_atual = 0
        for pontos, peso in dado_pesos:
            peso_atual += peso
            if (peso_atual >= ponto_medio):
                return pontos
    else:
        # Caso par calcula a média dos dois valores do meio
        ponto_medio = (pesos_totais / 2)
        peso_atual = 0

        for pontos, peso in dado_pesos:
            peso_atual += peso
            if (peso_atual >= ponto_medio):
                return pontos

def calcular_soma_ponderada(pontos, p, pesos):
    x = np.sum(np.linalg.norm(pontos - p, axis=1) * pesos)   # calcula a soma (p1(x,y) - p(x,y)) * pesos
    return x

def encontrar_valor_p(pontos, pesos):
    # Começar com um valor incial para p (ponto média)
    p = np.mean(pontos, axis=0)
    # Definir um critério de parada, quando a mudança em p for pequena o sufuciente
    tol = 1e-3

    while True:
        # Calcula a soma ponderada atual
        soma_atual = calcular_soma_ponderada(pontos, p, pesos)
        # Encontra a mediana ponderada
        p = mediana_ponderada(pontos, pesos)
        # Calcula a nova soma ponderada
        nova_soma = calcular_soma_ponderada(pontos, p, pesos)
        # Verifica se a mudança em p é pequena o suficiente
        if (np.linalg.norm(nova_soma - soma_atual) < tol):
            break
    return p
if __name__ == "__main__":
    pontos = np.array([[3,2],[8,1],[2,4],[5,6],[4,3],[1,5],[6,7]])  # Lista de pontos (coordenadas x,y)
    pesos = np.array([0.12,0.35, 0.025, 0.08, 0.15, 0.075, 0.2])  # Lista de pesos associados aos pontos
    valor_p_otimo = encontrar_valor_p(pontos, pesos)
    print("Valor de p que minimiza a soma ponderada:", valor_p_otimo)
```

Resultados e concluções: Este é um problema de otimização, e a escolha do método de otimização específico depende da complexidade dos seus dados e das questões de desempenho. A mediana ponderada é útil para ajudar a encontrar um valor **p** que minimize a soma ponderada das diferenças entre os elementos do vetor $\mathbf{p}_x$ $\mathbf{p}_y$ e **p**. A ideia neste algoritmo é que a mediada ponderada é uma tendencia central que leva em consideralção tanto os valores quanto os pesos. Usando a mediana ponderada como valor de **p**, o algoritmo está escolhendo um valor que está no “meio” dos valores ponderados, o que pode ajudar a minimizar a soma ponderada das diferenças. Aqui eu escolhi primeiramente o valor médio do array, mas na atualização realizado novamente o processo e utilizando a soma com a mediana ponderada, o valor **p_otimo** nos meus testes, tendeu a levar em consideração os valores da mediana ponderada em todos os casos em que eu executei.