Atividades de Complexidade de Algoritmos da Aula 14

Link Github códigos da questão 1: https://github.com/Luh022/Ativ-Complex-14

1 - Utilizando o método de divisão e conquista, crie algoritmos recursivos para:

→ Busca Linear:

Exemplo de uma implementação feita em python:

```
def busca_linear_recursiva(arr, elemento, inicio, fim):

if inicio > fim:

retornar -1

if arr[inicio] == elemento:

retorno inicio

if arr[fim] == elemento:

fim de retorno

return busca_linear_recursiva(arr, elemento, inicio + 1, fim - 1)

arr = [1, 2, 3, 4, 5, 6, 7, 8, 9]

elemento = 5

índice = busca_linear_recursiva(arr, elemento, 0, len(arr) - 1)

se índice! = -1:

print(f'O elemento {elemento} foi encontrado no índice {indice}.')

outro:

print(f'O elemento {elemento} não foi encontrado na lista.')
```

Neste algoritmo, a função `busca_linear_recursiva` recebe uma lista `arr`, o elemento que está sendo procurado `elemento`, o índice de início `inicio` e o índice de fim `fim`.

A cada chamada recursiva, o algoritmo divide a lista pela metade e verifica se o elemento está na metade esquerda ou direita. Se o elemento for encontrado, retorna o índice em que ele está. Caso contrário, a função é chamada recursivamente na metade completa da lista até que o elemento seja encontrado ou a lista seja completamente percorrida.

→ Busca Binária:

Exemplo de uma implementação em python:

```
def busca_binaria(arr, alvo, inicio, fim):

if inicio > fim:

retornar -1

meio = (inicio + fim) // 2

if arr[meio] == alvo:

retornar meio

elif arr[meio] < alvo:

return busca_binaria(arr, alvo, meio + 1, fim)

outro:

return busca_binaria(arr, alvo, inicio, meio - 1)
```

Nesse exemplo, `arr` representa o conjunto ordenado em que estamos procurando o elemento `alvo`. `inicio` e `fim` são os índices que delimitam a parte do conjunto em que estamos realizando a busca.

Primeiro, verificamos se `inicio` é maior que `fim`. Se for, isso significa que o elemento não foi encontrado e retornamos -1.

Em seguida, calculamos o índice do meio do conjunto, arredondando para baixo. Comparamos o elemento do meio com o elemento alvo. Se forem iguais, retornamos o índice do elemento encontrado.

Se o elemento do meio for menor que o alvo, chamamos a função `busca_binaria` recursivamente, passando `meio + 1` como novo início e mantendo o mesmo fim.

Se o elemento do meio para maior que o alvo, chamamos a função `busca_binaria` recursivamente, passando o mesmo início e `meio - 1` como novo fim.

→ Encontrar o menor elemento:

1. Comece dividindo a lista em duas metades.

2. Compare os menores elementos de cada metade.

3. Se o menor elemento da primeira metade for menor que o menor elemento da segunda metade, aplique recursivamente o algoritmo à primeira metade.

4. Se o menor elemento da segunda metade for menor que o menor elemento da primeira metade, aplique recursivamente o algoritmo à segunda metade.
5. Repita as etapas 1 a 4 até que a lista seja reduzida a um único elemento.
6. Retorne o elemento único como o menor elemento da lista.

Exemplo de uma implementação em python:

```
def find_smallest_element(arr):

# Caso base: se a lista contiver apenas um elemento, retorne-o

se len(arr) == 1:

retornar arr[0]

# Divida a lista em duas metades

meio = len(arr) // 2

primeira_metade = arr[:mid]

segundo_metade = arr[meio:]

# Encontre recursivamente o menor elemento em cada metade

menor_primeiro = encontrar_menor_elemento(primeira_metade)

menor_segundo = encontrar_menor_elemento(segundo_metade)

# Compare os menores elementos de cada metade

se menor_primeiro < menor_segundo:

retornar menor_primeiro

outro:

retornar menor_segundo
```

Para chamar uma função que passe uma lista como argumento, ela retorna o menor elemento da lista. Exemplo:

```
arr = [5, 2, 9, 1, 7]

menor = find_smallest_element(arr)

imprimir(menor) # Saída: 1
```

2 - Utilizando o método guloso, defina um algoritmo para obter a árvore geradora de um grafo G = (V, E)

Para obter a árvore geradora mínima de um grafo G = (V, E) utilizando o método guloso, podemos seguir o seguinte algoritmo:

1. Inicialize uma árvore vazia T, que será nossa árvore geradora mínima.

2. Escolha um vértice arbitrário v em V e acrescenta-o a T.

3. Enquanto T não contiver todos os vértices de V, faça:

a. Encontre uma aresta de menor peso que conecta um vértice em T a um vértice fora de T.

b. Adicione essa aresta a T.

c. Adicione o vértice fora de T à T.

4. Retorne a árvore T como uma árvore geradora mínima de G.

O algoritmo guloso funciona selecionando uma área de menor peso a cada iteração, garantindo que uma árvore geradora mínima seja construída passo a passo. É importante notar que esse método não garante a obtenção da árvore geradora mínima em todos os casos, mas é eficiente e geralmente produz bons resultados.

3 - Utilizando programação dinâmica, solucione o problema do máximo subarray

O problema do subarray máximo consiste em encontrar um subarray contígua com a maior soma em um array dado. Podemos resolver esse problema utilizando programação dinâmica com o seguinte algoritmo:

1. Inicialize duas variáveis: "max_ate_agora" e "max_total", ambas com valor igual ao primeiro elemento do array.

2. Percorra o array do segundo elemento até o último.

3. Para cada elemento, atualize a variável "max_ate_agora" para o máximo entre o elemento atual e a soma do elemento atual com "max_ate_agora".

4. Atualize a variável "max_total" para o máximo entre "max_total" e "max_ate_agora".

5. Retorne o valor de "max_total" como a soma máxima do subarray.

O algoritmo utiliza duas variáveis para rastrear a soma máxima até o momento e a soma máxima total encontrada até o momento. A cada iteração, atualizamos essas variáveis para garantir que sempre tenhamos o soma máxima.

A complexidade desse algoritmo é O(n), onde n é o tamanho do array, já que precisamos percorrer todos os elementos do array uma única vez.