FUNDAÇÃO EDSON QUEIROZ
UNIVERSIDADE DE FORTALEZA
ENSINANDO E APRENDENDO

Trab.	-	-	X

Curso: Construção e Análise de Algoritmos	Código/Turma:
Professor: Ronaldo Gonçalves Junior	Data:
Aluno(a): Aureliano Claudio de Queiroz Sousa	Matrícula: 2214647
Aluno(a): Henrique Façanha Dutra	Matrícula: 2217674
Aluno(a): Pedro Andrade	Matrícula: 2214627
Aluno(a): Thiago de Oliveira	Matrícula: 2210387
Aluno(a): Carlos Eduardo	Matrícula: 2217672

INSTRUÇÕES PARA RESOLUÇÃO DO LAB 3

- Preencher o campo Aluno(a) com o nome completo e o campo Matrícula para cada integrante da equipe
- Preencher os passos abaixo ou anexar foto com respostas à mão - utilizar caneta ou lápis de fácil visibilidade - apenas soluções legíveis serão corrigidas.
- Enviar o documento em formato pdf pelo AVA na área “Trabalhos”

LAB 3

Exercício 1 – Ordenação

Passo 1: Para um vetor sobre o domínio dos números inteiros positivos, desejamos resolver o problema de ordenação dos elementos deste vetor em ordem **decrescente**. Construa três algoritmos para resolver este problema, utilizando pelo menos uma vez cada um dos seguintes paradigmas algorítmicos: força bruta e divisão e conquista.

Passo 2: Implemente os algoritmos de ordenação construídos no passo anterior utilizando a linguagem Java ou outra linguagem de programação.

Passo 3: Para cada programa do passo anterior, modifique o código para mostrar cada atualização feita no vetor de entrada até que uma saída válida seja obtida. Inclua um *print* com os resultados do console.

pseudo-código:

```
1    QuickSort(A, p, r){
2        se p < r
3            então q <- Particao(A, p, r)
4                QuickSort(A, p, q-1)
5                QuickSort(A, q + 1, r)
6    }
7
8    Particao(A, p, r){
9        x <- A[r]
10       i <- p - 1
11       aux <- 0
12       para j <- p até r - 1
13           faça se A[j] <= x
14               então i <- i + 1
15                   aux <- A[j]
16                   A[j] <- A[i]
17                   A[i] <- aux
18       aux <- A[r]
19       A[r] <- A[i+1]
20       A[i] <- aux
21       retorne i+1
22
23   }
```

```
1    MergeSort(A, p, r){
2        se p >  r
3            então q <- |_(p+r)/2_|
4                  MergeSort(A, p, q)
5                  MergeSort(A, q+1, r)
6                  Merge(A, p, q, r)
7    }
8
9
10   Merge(A, p, q, r){
11       n1 <- q - p +1
12       n2 <- r - q
13
14       arranjo L[n1+1]
15       arranjo R[n2+1]
16
17       para i <- 1 até n1
18           faça L[i] <- A[p+i-1]
19       para j <- 1 até n2
20           faça R[j] <- A[q+j]
21
22       L[n1 + 1] <- infinito positivo
23       R[n2 + 1] <- infinito positivo
24
25       i <- 1
26       j <- 1
27
28       para k <- p até r
29           faça se L[i] <= R[j]
30                   então A[k] <- L[i]
31                        i <- i + 1
32                   se não A[k] <- R[j]
33                        j <- j + 1
34   }
```

```
1    BruteForceSorting(A, n){
2        aux := 0
3        para i:= 1 até n
4            para j:= 1 até n
5                se A[i] > A[j]:
6                    aux := A[i]
7                    A[i] := A[j]
8                    A[j] := aux
9    }
```

Implementação

```python
from vetorAleatorio import vetorAleatorio

def QuickSort(A, p, r):
    if p < r:
        q = Partition(A, p ,r)
        QuickSort(A, p, q-1)
        QuickSort(A, q+1, r)
    pass


def Partition(A, p, r):
    x = A[r]
    i = p-1
    aux = 0
    for j in range(p, r):
        if A[j] >= x:
            i += 1
            aux = A[j]
            A[j] = A[i]
            A[i] = aux
        print(A)
    aux = A[r]
    A[r] = A[i+1]
    A[i+1] = aux
    print(A)
    return i+1

A = vetorAleatorio()
p=0
r = len(A) -1

print(A)

QuickSort(A, p, r)
```

```python
import math
from vetorAleatorio import vetorAleatorio
def MergeSort(A, p, r):
    if p < r:
        q = int((p + r) / 2)
        MergeSort(A, p, q)
        MergeSort(A, q + 1, r)
        Merge(A, p, q, r)

def Merge(A, p, q, r):
    n1 = q - p + 1
    n2 = r - q

    L = []
    R = []
    for i in range(int(n1)):
        L.append(A[p + i])
    for i in range(int(n2)):
        R.append(A[q + 1 + i])

    i = 0
    j = 0
    for k in range(p, r + 1):
        if i < n1 and (j >= n2 or L[i] >= R[j]):
            A[k] = L[i]
            i += 1
        else:
            A[k] = R[j]
            j += 1
        print(A)


A = vetorAleatorio()
p = 0
r = len(A) - 1
print(A)

MergeSort(A, p, r)
```

```python
#Gerar vetor
from vetorAleatorio import vetorAleatorio

vetor = vetorAleatorio()

print(vetor)
aux = 0
for i in range(len(vetor)):
    for j in range(len(vetor)):
        if vetor[i] > vetor[j]:
            aux = vetor[i]
            vetor[i] = vetor[j]
            vetor[j] = aux
            print(vetor)
```

Análise:

EXERCÍCIO 1:

==========

FORÇA BRUTA

Código em Python:

		TEMPO:
from vetorAleatorio import vetorAleatorio		O(1)
vetor = vetorAleatorio()	O(1)	
print(vetor)		O(1)
aux = 0		O(1)
for i in range(len(vetor)):		O(n)
for j in range(len(vetor)):		O(n^2)
if vetor[i] > vetor[j]:		O(n^2)
aux = vetor[i]	O(n^2)	
vetor[i] = vetor[j]		O(n^2)
vetor[j] = aux	O(n^2)	
print(vetor)	O(n^2)	

TOTAL: TEMPO: O(n^2) ESPAÇO: O(1)

Pseudocódigo:----------

c ← primeiro(S)
enquanto c ≠ vazio faça
  se validar(S,c) então fim(S, c)
  c ← próximo(S,c)
fim enquanto

==========

MERGE SORT

Código Python:

	TEMPO:
import math	
from vetorAleatorio import vetorAleatorio	

```
def MergeSort(A, p, r):
    if p < r:                                   O(1)
        q = int((p + r) / 2)                    O(1)
        MergeSort(A, p, q)                      O(logn)
        MergeSort(A, q + 1, r)                  O(logn)
        Merge(A, p, q, r)

def Merge(A, p, q, r):
    n1 = q - p + 1                              O(1)
    n2 = r - q                                  O(1)

    L = []
    R = []
    for i in range(int(n1)):                    O(n1)
        L.append(A[p + i])
    for i in range(int(n2)):                    O(n2)
        R.append(A[q + 1 + i])

    i = 0
    j = 0
    for k in range(p, r + 1):                   O(1)
        if i < n1 and (j >= n2 or L[i] >= R[j]):O(1)
            A[k] = L[i]                         O(1)
            i += 1
        else:
            A[k] = R[j]                         O(1)
            j += 1
        print(A)
A = vetorAleatorio()
p = 0
r = len(A) - 1
print(A)

MergeSort(A, p, r)
```

TOTAL: TEMPO: O(nlogn) ESPAÇO: O(n)

Pseudocódigo:----------

```
função mergesort (vetor a)
    se (n == 1) retornar a

    vetor l1 = a[0] ... a[n/2]
    vetor l2 = a[n/2 + 1] ... a[n]

    l1 = mergesort(l1)
    l2 = mergesort(l2)

    retornar mesclar(l1, l2)
fim da função mergesort

função mesclar (vetor a, vetor b)
    vetor c

    enquanto (a e b têm elementos)
        if (a[0] > b[0])
            adicionar b[0] ao final de c
            remover b[0] de b
        senão
            adicionar a[0] ao final de c
            remover a[0] de a
    enquanto (a tem elementos)
        adicionar a[0] ao final de c
        remover a[0] de a
    enquanto (b tem elementos)
        adicionar b[0] ao final de c
        remover b[0] de b
    retornar c
fim da função mesclar
```

==========

QUICKSORT
Código Python:

```
from vetorAleatorio import vetorAleatorio        TEMPO:

def QuickSort(A, p, r):
    if p < r:                                    O(1)
        q = Partition(A, p, r)                   O(n)
        QuickSort(A, p, q-1)                     T(n/2)
        QuickSort(A, q+1, r)                     T(n/2)

def Partition(A, p, r):
    x = A[r]
    i = p - 1
    aux = 0
    for j in range(p, r):                O(n)
        if A[j] >= x:                            O(n)
            i += 1                       O(n)
            aux = A[j]                           O(n)
            A[j] = A[i]                          O(n)
            A[i] = aux                           O(n)
        print(A)                         O(n)
    aux = A[r]
    A[r] = A[i+1]
    A[i+1] = aux
    print(A)
    return i + 1
                                                 T(n)=2T(n/2)+O(n)
                        TOTAL: TEMPO: O(nlogn) ESPAÇO: O(log2n)
```

Pseudocódigo:----------

```
procedimento QuickSort(X[], IniVet, FimVet)
var
   i, j, pivo, aux
início
   i <- IniVet
   j <- FimVet
   pivo <- X[(IniVet + FimVet) div 2]
   enquanto(i <= j)
         enquanto (X[i] < pivo) faça
             i <- i + 1
         fimEnquanto
         enquanto (X[j] > pivo) faça
             j <- j - 1
         fimEnquanto
         se (i <= j) então
             aux  <- X[i]
             X[i] <- X[j]
             X[j] <- aux
             i <- i + 1
             j <- j - 1
         fimSe
   fimEnquanto
   se (IniVet < j) então
      QuickSort(X, IniVet, j)
   fimSe
   se (i < FimVet) então
      QuickSort(X, i, FimVet)
   fimSe
fimProcedimento
```

==========

Exercício 2 – Fibonacci

"Na matemática, a sucessão de Fibonacci (ou sequência de Fibonacci), é uma sequência de números inteiros, começando normalmente por 0 e 1, na qual cada termo subsequente corresponde à soma dos dois anteriores. A sequência recebeu o nome do matemático italiano Leonardo de Pisa ou Leonardo Fibonacci, mais conhecido por apenas Fibonacci, que descreveu, no ano de 1202, o crescimento de uma população de coelhos, a partir desta. Esta sequência já era, no entanto, conhecida na antiguidade." (Wikipédia, 2023).

Passo 1: Construa um algoritmo de divisão-e-conquista para retornar o valor na sequência de Fibonacci de acordo com uma entrada *n,* um inteiro não-negativo, que indica a posição do elemento na sequência. O algoritmo deve possuir uma complexidade de tempo $\Theta(2^n)$.

Passo 2: Para o mesmo problema do passo anterior, construa um algoritmo de programação dinâmica na abordagem *top-down*. O algoritmo deve possuir uma complexidade de tempo $\Theta(n)$.

Passo 3: Para o mesmo problema do passo 1, construa um algoritmo de programação dinâmica na abordagem *bottom-up*. O algoritmo deve possuir uma complexidade de tempo $\Theta(n)$.

Passo 4: Implemente os algoritmos construídos no exercício 2 utilizando a linguagem Java ou outra linguagem de programação.

Passo 5: Para cada programa do passo anterior, modifique o código para mostrar o resultado de cada cálculo de Fibonacci até que uma saída válida seja obtida. Inclua um *print* com os resultados do console.

Implementação:

```
function fibonacci_dc(n) {
    if (n <= 1) {
        return n;
    } else {
        return fibonacci_dc(n - 1) + fibonacci_dc(n - 2);
    }
}

function fibonacci_top_down(n, memo = {}) {
    if (n <= 1) {
        return n;
    } else if (!(n in memo)) {
        memo[n] = fibonacci_top_down(n - 1, memo) + fibonacci_top_down(n - 2, memo);
    }
    return memo[n];
}

function fibonacci_bottom_up(n) {
    if (n <= 1) {
        return n;
    }
    const fib = new Array(n + 1).fill(0);
    fib[1] = 1;
    for (let i = 2; i <= n; i++) {
        fib[i] = fib[i - 1] + fib[i - 2];
    }
    return fib[n];
}

function main() {
    const n = parseInt(prompt("Digite um número inteiro não-negativo:"));

    const result_dc = fibonacci_dc(n);
    console.log(`Algoritmo de Divisão e Conquista: O ${n}-ésimo termo na sequência de Fibonacci é: ${result_dc}`);

    const result_top_down = fibonacci_top_down(n);
    console.log(`Algoritmo de Programação Dinâmica (Top-Down): O ${n}-ésimo termo na sequência de Fibonacci é: ${result_top_down}`);

    const result_bottom_up = fibonacci_bottom_up(n);
    console.log(`Algoritmo de Programação Dinâmica (Bottom-Up): O ${n}-ésimo termo na sequência de Fibonacci é: ${result_bottom_up}`);
}

main();
```

Analise:

```
ANALISE FIBONACCI função fibonacci_dc(n):
    se n <= 1:
        retornar n
    senão:
        retornar fibonacci_dc(n - 1) + fibonacci_dc(n - 2)


Analise de tempo:
1
fibonacci(n) = fibonacci(n-1) + fibonacci(n-2) 2

fibonacci(n-1) = fibonacci(n-2) + fibonacci(n-3)/fibonacci(n-2) = fibonacci(n-3) + fibonacci(n-4)  4

fibonacci(n-2) = fibonacci(n-3) + fibonacci(n-4)/fibonacci(n-3) = fibonacci(n-4) + fibonacci(n-5)/fibonacci(n-3) = fibonacci(n-4) + fibonacci(n-5)/fibonacci(n-4) = fibonacci(n-5) + fibonacci(n-6)  8

.
.
.
...  fibonacci(1) + fibonacci(0)...


altura igual a n e o númeor de chamadas duplica a cada nível, logo a complexidade é 2^n
```

```
Analise de espaço:


função fibonacci_dc(n):                                 1
    se n <= 1:
        retornar n                                      0
    senão:
        retornar fibonacci_dc(n - 1) + fibonacci_dc(n - 2)      0
função fibonacci_bottom_up(n):
    se n <= 1:
        retornar n
    fib = array preenchido com 0 de tamanho n + 1
    fib[1] = 1
    para i de 2 até n:
        fib[i] = fib[i - 1] + fib[i - 2]
    retornar fib[n]


complexidade de tempo:


função fibonacci_bottom_up(n):
    se n <= 1:                                  1
        retornar n                              1
    fib = array preenchido com 0 de tamanho n + 1       1
    fib[1] = 1                                  1
    para i de 2 até n:                          n-1
        fib[i] = fib[i - 1] + fib[i - 2]        n-2
    retornar fib[n]                             1


complexidade é igual a n
```

```
complexidade de espaço:


função fibonacci_bottom_up(n):                          1
    se n <= 1:
        retornar n
    fib = array preenchido com 0 de tamanho n + 1       n+1
    fib[1] = 1
    para i de 2 até n:
        fib[i] = fib[i - 1] + fib[i - 2]
    retornar fib[n]

complexidade de espaço = n
função fibonacci_top_down(n, memo = {}):
    se n <= 1:
        retornar n
    senão, se n não estiver em memo:
        memo[n] = fibonacci_top_down(n - 1, memo) + fibonacci_top_down(n - 2, memo)
    retornar memo[n]




complexidade de tempo


ftd(n)

ftd(n-1)

ftd(n-2)

ftd(n-3)

ftd(n-4)

ftd(n-5)

...

ftd(0)


altura igual a n, apenas uma chamada por nível, logo complexidade é n
```

```
complexidade de espaço


função fibonacci_top_down(n, memo = {}):                                1+n
    se n <= 1:
        retornar n                                                      0
    senão, se n não estiver em memo:
        memo[n] = fibonacci_top_down(n - 1, memo) + fibonacci_top_down(n - 2, memo)     0
    retornar memo[n]                                                    0


complexidade de espaço: n
```

Exercício 3 – Oito damas

“O problema das oito damas é o problema matemático de dispor oito damas em um tabuleiro de xadrez de dimensão 8x8, de forma que nenhuma delas seja atacada por outra. Para tanto, é necessário que duas damas quaisquer não estejam numa mesma linha, coluna, ou diagonal. Este é um caso específico do Problema das n damas, no qual temos n damas e um tabuleiro com n×n casas(para qualquer n ≥ 4). … O problema foi inicialmente proposto na revista Schachzeitung pelo enxadrista Max Bezzel em 1848, e ao longo dos anos foi avaliado por diversos matemáticos incluindo Gauss e Georg Cantor. A primeira solução foi proposta em 1850 por Franz Nauck, que também o generalizou para o Problema das n damas.” (Wikipédia, 2023)

Passo 1: Veja a explicação do problema das oito damas[1] e construa um algoritmo de tentativa-e-erro (backtracking) para imprimir o resultado em um tabuleiro de xadrez.

Passo 2: Implemente o algoritmo construído no passo anterior utilizando a linguagem Java ou outra linguagem de programação.
Passo 3: Para o programa do passo anterior, modifique o código para mostrar o resultado de cada tentativa no tabuleiro de xadrez.

Implementação:

```
n = 8
board = [[0 for i in range(n)] for i in range(n)]

solutions = []

def check_column(board, row, column):
    for i in range(row, -1, -1):
        if board[i][column] == 1:
            return False
    return True

def check_diagonal(board, row, column):
    for i, j in zip(range(row, -1, -1), range(column, -1, -1)):
        if board[i][j] == 1:
            return False
    for i, j in zip(range(row, -1, -1), range(column, n, 1)):
        if board[i][j] == 1:
            return False
    return True

def nqn(board, row):
    if row == n:

        solutions.append([row[:] for row in board])
        return

    for i in range(n):
        if check_column(board, row, i) and check_diagonal(board, row, i):
            board[row][i] = 1
            nqn(board, row + 1)
            board[row][i] = 0

nqn(board, 0)


for solution in solutions:
    for row in solution:
        print(row)
    print("\n")
```

Algoritmo:

[1] https://pt.wikipedia.org/wiki/Problema_das_oito_damas

```
Função check_column(board, row, column):
    Para i de row até 0, passo -1 faça
        Se board[i][column] == 1 então
            Retorne Falso
    Retorne Verdadeiro

Função check_diagonal(board, row, column):
    Para i, j em zip(range(row, -1, -1), range(column, -1, -1)) faça
        Se board[i][j] == 1 então
            Retorne Falso
    Para i, j em zip(range(row, -1, -1), range(column, n, 1)) faça
        Se board[i][j] == 1 então
            Retorne Falso
    Retorne Verdadeiro

Procedimento nqn(board, row):
    Se row == n então
        Adicione uma cópia da configuração atual do tabuleiro em solutions
        Retorne
    Para i de 0 até n-1 faça
        Se check_column(board, row, i) e check_diagonal(board, row, i) então
            board[row][i] = 1
            Chame nqn(board, row + 1)
            board[row][i] = 0

Procedimento imprimir_solucoes(solutions):
    Para cada solução em solutions faça
        Para cada linha em solução faça
            Imprima linha
        Imprima uma linha em branco


n = LeiaInteiro()

board = Matriz de tamanho n x n, preenchida com zeros
solutions = Lista vazia

nqn(board, 0)

imprimir_solucoes(solutions)
```

Analise:

```
Analise da Função check_column

board = [[0 for i in range(n)] for i in range(n)]   # Custo: n²

solutions = []                                     # Custo: 1

def check_column(board, row, column):
    for i in range(row, -1, -1):                   # Custo: row + 1
        if board[i][column] == 1:                  # Custo: 1
            return False                           # Custo: 1
    return True                                    # Custo: 1

Analise de tempo:
n² + 1 + (row + 1) * (1 + 1) = n² + 2(row + 1) = n² + 2n => O(n²)



board = [[0 for i in range(n)] for i in range(n)]   # Espaço: n²

solutions = []                                     # Espaço: 1

def check_column(board, row, column):
    for i in range(row, -1, -1):                   # Espaço: 1
        if board[i][column] == 1:                  # Espaço: 1
            return False
    return True                                    # Espaço: 1

Analise de Espaço:
n² + 1 + 1 + 1 + 1 = n² + 4 => O(n²)
```

```
Analise da Função chech_diagonal

def check_diagonal(board, row, column):
    for i, j in zip(range(row, -1, -1), range(column, -1, -1)):  # Custo: n
        if board[i][j] == 1:                                     # Custo: 1
            return False                                         # Custo: 1
    for i, j in zip(range(row, -1, -1), range(column, n, 1)):    # Custo: n
        if board[i][j] == 1:                                     # Custo: 1
            return False                                         # Custo: 1
    return True                                                  # Custo: 1

Analise de tempo:
n + 1 + 1 + n + 1 + 1 + 1 = 2n + 5 => O(n)



def check_diagonal(board, row, column):
    for i, j in zip(range(row, -1, -1), range(column, -1, -1)):  # Espaço: 1
        if board[i][j] == 1:                                     # Espaço: 1
            return False
    for i, j in zip(range(row, -1, -1), range(column, n, 1)):    # Espaço: 1
        if board[i][j] == 1:                                     # Espaço: 1
            return False
    return True                                                  # Espaço: 1

Analise de Espaço:
1+ 1 + 1 + 1 + 1 = 5 => O(1)
```

```
Analise da Função nqn

def nqn(board, row):
    if row == n:                                                 # Custo: n¹

        solutions.append([row[:] for row in board])
        return

    for i in range(n):                                           # Custo: n
        if check_column(board, row, i) and check_diagonal(board, row, i): #Custo: n
            board[row][i] = 1                                    # Custo: 1
            nqn(board, row + 1)                                  # Custo: n^3
            board[row][i] = 0                                    # Custo: 1

nqn(board, 0)

Analise de tempo:
n² + n + n + 1 + n^3 + 1 = n^3 + n² + 2n + 2 => O(n^3)

def nqn(board, row):
    if row == n:                                                 # Espaço: n²

        solutions.append([row[:] for row in board])
        return

    for i in range(n):
        if check_column(board, row, i) and check_diagonal(board, row, i):
            board[row][i] = 1                                    # Espaço: 1
            nqn(board, row + 1)
            board[row][i] = 0                                    # Espaço: 1

nqn(board, 0)

Analise de Espaço: n² + 1 + 1 = n² + 2 => O(n²)
```

Exercício 4 – Agendamento

Sua empresa de desenvolvimento de *software* recebeu muitas ofertas de projeto para o ano de 2024 e seus analistas projetaram quando os projetos seriam iniciados e concluídos, além de calcular um valor de lucro para cada projeto. Em outras palavras, cada oferta possui uma data de início, uma data de conclusão e um valor de lucro. Como os projetos são muitos e existem conflitos de data, será necessário fazer uma escolha entre quais projetos serão desenvolvidos.

Passo 1: De acordo com o cenário acima, construa um algoritmo guloso para escolher quais projetos serão desenvolvidos. O algoritmo deve receber como entrada a lista de projetos, com data inicial, data final e um número real de lucro.

Passo 2: Implemente o algoritmo construído no passo anterior utilizando a linguagem Java ou outra linguagem de programação.

Passo 3: Para o programa do passo anterior, modifique o código para mostrar cada seleção assim que é realizada.

Implementação:

```python
def greedy_project_selection_with_print(projects):
    projects.sort(key=lambda x: x[2] / (x[1] - x[0]), reverse=True)

    selected_projects = []
    current_end = float('-inf')

    for project in projects:
        start, end, profit = project
        if start >= current_end:
            selected_projects.append(project)
            current_end = end
            print(f"Projeto selecionado - Data de Início: {start}, Data de

    return selected_projects


projects = [(1, 3, 5), (2, 5, 8), (4, 6, 3), (6, 8, 2)]
result = greedy_project_selection_with_print(projects)
```

```python
def greedy_project_selection_with_print(projects):

    projects.sort(key=lambda x: x[2] / (x[1] - x[0]), reverse=True)


    selected_projects = []
    current_end = float('-inf')


    for project in projects:
        start, end, profit = project

        if start >= current_end:

            selected_projects.append(project)
            current_end = end
            print(f"Projeto selecionado - Data de Início: {start}, Data de Conclusão: {end}, Lucro: {profit}")


    return selected_projects Tempo: O(1)


projects = [(1, 3, 5), (2, 5, 8), (4, 6, 3), (6, 8, 2)]
result = greedy_project_selection_with_print(projects)


ANALISE TEMPO: N LOG N + N + N + 1 + 1 = O(N LOG N)



Analise de Espaço:

def greedy_project_selection_with_print(projects):


    projects.sort(key=lambda x: x[2] / (x[1] - x[0]), reverse=True)


    selected_projects = []
    current_end = float('-inf')


    for project in projects:
        start, end, profit = project

        if start >= current_end:

            selected_projects.append(project)
```

Custo:
Tempo: O(n log n)
Tempo: O(n)
Tempo: O(n)
Tempo: O(1)
Tempo: O(1)

Tempo:
Espaço: O(1)
Espaço: O(n)
Espaço: O(1)
Espaço: O(1)
Espaço: O(1)

```python
            current_end = end
            print(f"Projeto selecionado - Data de Início: {start}, Data de Conclusão: {end}, Lucro: {profit}")


     return selected_projects


projects = [(1, 3, 5), (2, 5, 8), (4, 6, 3), (6, 8, 2)]
result = greedy_project_selection_with_print(projects)


ANALISE ESPAÇO: 1 + N + 1 + 1 + 1 + N + O(N)
```

Espaço: O(n)

Analise

Exercício 5 – Caixeiro-viajante

“O problema do caixeiro-viajante (PCV) é um problema que tenta determinar a menor rota para percorrer uma série de cidades (visitando uma única vez cada uma delas), retornando à cidade de origem. Ele é um problema de otimização NP-difícil inspirado na necessidade dos vendedores em realizar entregas em diversos locais (as cidades) percorrendo o menor caminho possível, reduzindo o tempo necessário para a viagem e os possíveis custos com transporte e combustível.” (Wikipédia, 2023)

Passo 1: Veja a explicação do problema do caixeiro-viajante e construa um algoritmo força bruta para resolver o problema exaustivamente dado um pequeno número de cidades (8, 9 e 10).

```dart
import 'dart:math';

Run | Debug
void main() {
  List<List<int>> grafo = gerarGrafo(8);
  //List<List<int>> grafo = gerarGrafo(9);
  //List<List<int>> grafo = gerarGrafo(10);

  CaixeiroViajante caixeiro = CaixeiroViajante();
  List<int> melhorRota = caixeiro.encontrarMenorRota(grafo);

  print("Melhor rota final: $melhorRota");
  print(
      "Distância mínima final: ${caixeiro.calcularDistancia(grafo, melhorRota)}");
}

List<List<int>> gerarGrafo(int n) {
  Random random = Random();
  List<List<int>> grafo = List.generate(n, (index) => List<int>.filled(n, 0));

  for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
      if (i != j) {
        grafo[i][j] = random.nextInt(20) + 1;
      }
    }
  }

  return grafo;
}
```

No código ao lado a função main possui as chamadas para o grafo usar a função gerar grafos com 8x8,9x9 e 10x10 cidades, dessa forma facilita na criação dos grafos. A função gerarGrafo ela gera uma arrays com valores aleatórios dependendo do valor da entrada.

```dart
class CaixeiroViajante {
  List<int> encontrarMenorRota(List<List<int>> grafo) {
    int n = grafo.length;
    List<int> cidades = List.generate(n, (index) => index);
    List<int> melhorRota = List.from(cidades);
    int menorDistancia = calcularDistancia(grafo, melhorRota);

    int interacao = 1;
    do {
      int distanciaAtual = calcularDistancia(grafo, cidades);
      print("Interação $interacao: Rota = $cidades, Distância = $distanciaAtual");

      if (distanciaAtual < menorDistancia) {
        menorDistancia = distanciaAtual;
        melhorRota.setAll(0, cidades);
      }
      interacao++;
    } while (proxPermut(cidades));

    return melhorRota;
  }
```

A função encontrarMenorRota ,ele gera permutações das cidades, calcula a distância para cada permutação e armazena a rota com a menor distância encontrada. O processo é repetido até testar todas as permutações possíveis, retornando a rota mais curta.

```dart
int calcularDistancia(List<List<int>> grafo, List<int> rota) {
  int distancia = 0;
  for (int i = 0; i < rota.length - 1; i++) {
    distancia += grafo[rota[i]][rota[i + 1]];
  }
  distancia += grafo[rota.last][rota.first];
  return distancia;
}
```

A função calcularDistancia calcula a distância total percorrida ao longo de uma rota em um grafo ponderado. Utilizando uma matriz de adjacência (grafo) e uma lista que representa a ordem de visita das cidades (rota), a função soma os custos das arestas correspondentes na ordem da rota, incluindo a última aresta para fechar o ciclo. O resultado é a distância total percorrida na rota.

```dart
bool proxPermut(List<int> array) {
  int i = array.length - 1;
  while (i > 0 && array[i - 1] >= array[i]) {
    i--;
  }

  if (i <= 0) {
    return false;
  }

  int j = array.length - 1;
  while (array[j] <= array[i - 1]) {
    j--;
  }

  array[i - 1] ^= array[j];
  array[j] ^= array[i - 1];
  array[i - 1] ^= array[j];

  j = array.length - 1;
  while (i < j) {
    array[i] ^= array[j];
    array[j] ^= array[i];
    array[i] ^= array[j];
    i++;
    j--;
  }

  return true;
}
```

A função proxPermut implementa um algoritmo de força bruta para gerar permutações de um array, utilizado no contexto do Problema do Caixeiro Viajante. Ele troca elementos no array e inverte a parte restante para encontrar a próxima permutação. O termo "força bruta" refere-se à abordagem exaustiva de testar todas as possíveis permutações para encontrar a solução desejada, como realizada neste código para explorar todas as ordens de visita às cidades.

Passo 2: Implemente o algoritmo construído no passo anterior utilizando a linguagem Java ou outra linguagem de programação.

https://dontpad.com/CAnA_Aureliano_2023_FB_dart

Código completo do Caixeiro Viajante implementando força bruta feito em Dart*

*Necessário instalação da linguagem, mas pode usar compilador online.

Passo 3: Para o programa do passo anterior, modifique o código para mostrar cada atualização no resultado de escolhas de cidade.

```
List<int> encontrarMenorRota(List<List<int>> grafo) {
  int n = grafo.length;
  List<int> cidades = List.generate(n, (index) => index);
  List<int> melhorRota = List.from(cidades);
  int menorDistancia = calcularDistancia(grafo, melhorRota);

  int interacao = 1;
  do {
    int distanciaAtual = calcularDistancia(grafo, cidades);
    print("Interação $interacao: Rota = $cidades, Distância = $distanciaAtual");

    if (distanciaAtual < menorDistancia) {
      menorDistancia = distanciaAtual;
      melhorRota.setAll(0, cidades);
    }
    interacao++;
  } while (proxPermut(cidades));

  return melhorRota;
}
```

Nessa linha marcada é onde em cada interação é printada no console a atualização.

Análise de soluções

Realize uma análise detalhada de complexidade de tempo e espaço para cada algoritmo construído nos exercícios anteriores. O trabalho pode ser feito de forma colaborativa, mas cada integrante deve ser responsável pela análise de, pelo menos, um exercício. Um integrante não pode analisar seu próprio exercício.

Para cada integrante da equipe, informe o nome do aluno(a) e indique quais exercícios foram solucionados e analisados por este integrante:

Integrante	Exercício(s) - Solução	Exercício(s) - Análise
Aureliano Claudio	5	1
Henrique Façanha	1	2
Carlos Eduardo	4	3
Thiago de Oliveira	2	4
Pedro Andrade	3	5