**Mini Project 1**

**Problem Statement:**

Mini Project - Write a program to implement matrix multiplication. Also implement multithreaded matrix multiplication with either one thread per row or one thread per cell. Analyze and compare their performance.

import time

def matrix\_multiply(A, B):

n = len(A)

m = len(A[0])

p = len(B[0])

# Initialize result matrix

C = [[0 for \_ in range(p)] for \_ in range(n)]

# Perform matrix multiplication

for i in range(n):

for j in range(p):

for k in range(m):

C[i][j] += A[i][k] \* B[k][j]

return C

def get\_matrix(rows, cols):

print(f"Enter the matrix (rows: {rows}, cols: {cols}):")

matrix = []

for i in range(rows):

row = list(map(int, input(f"Row {i + 1}: ").split()))

if len(row) != cols:

print(f"Error: Row {i + 1} must contain exactly {cols} elements.")

exit(1)

matrix.append(row)

return matrix

# Main execution

if \_\_name\_\_ == "\_\_main\_\_":

# User input for dimensions of matrices

rows\_A = int(input("Enter the number of rows for matrix A: "))

cols\_A = int(input("Enter the number of columns for matrix A: "))

rows\_B = int(input("Enter the number of rows for matrix B (must be equal to columns of A): "))

cols\_B = int(input("Enter the number of columns for matrix B: "))

if cols\_A != rows\_B:

print("Error: Number of columns in A must equal number of rows in B.")

exit(1)

# Get matrices A and B from user

A = get\_matrix(rows\_A, cols\_A)

B = get\_matrix(rows\_B, cols\_B)

# Multiply matrices A and B

start = time.time()

C = matrix\_multiply(A, B)

end = time.time()

# Print resulting matrix C

print("Resulting Matrix C (A \* B):")

for row in C:

print(row)

print(f"Time taken: {end - start} seconds")

**OUTPUT:**

Enter the number of rows for matrix A: 3

Enter the number of columns for matrix A: 3

Enter the number of rows for matrix B (must be equal to columns of A): 3

Enter the number of columns for matrix B: 3

Enter the matrix (rows: 3, cols: 3):

Row 1: 1 2 3

Row 2: 4 5 6

Row 3: 7 8 9

Enter the matrix (rows: 3, cols: 3):

Row 1: 9 8 7

Row 2: 6 5 4

Row 3: 3 2 1

Resulting Matrix C (A \* B):

[30, 24, 18]

[84, 69, 54]

[138, 114, 90]

Time taken: 0.0 seconds

import time

import threading

# Thread function to compute one row

def multiply\_row(A, B, C, row):

n = len(A)

m = len(A[0])

p = len(B[0])

for j in range(p):

C[row][j] = sum(A[row][k] \* B[k][j] for k in range(m))

def matrix\_multiply\_multithreaded(A, B):

n = len(A)

p = len(B[0])

# Initialize result matrix

C = [[0 for \_ in range(p)] for \_ in range(n)]

# Create threads, one per row

threads = []

for i in range(n):

thread = threading.Thread(target=multiply\_row, args=(A, B, C, i))

threads.append(thread)

thread.start()

# Wait for all threads to complete

for thread in threads:

thread.join()

return C

def get\_matrix(rows, cols):

print(f"Enter the matrix (rows: {rows}, cols: {cols}):")

matrix = []

for i in range(rows):

row = list(map(int, input(f"Row {i + 1}: ").split()))

if len(row) != cols:

print(f"Error: Row {i + 1} must contain exactly {cols} elements.")

exit(1)

matrix.append(row)

return matrix

# Main execution

if \_\_name\_\_ == "\_\_main\_\_":

# User input for dimensions of matrices

rows\_A = int(input("Enter the number of rows for matrix A: "))

cols\_A = int(input("Enter the number of columns for matrix A: "))

rows\_B = int(input("Enter the number of rows for matrix B (must be equal to columns of A): "))

cols\_B = int(input("Enter the number of columns for matrix B: "))

if cols\_A != rows\_B:

print("Error: Number of columns in A must equal number of rows in B.")

exit(1)

# Get matrices A and B from user

A = get\_matrix(rows\_A, cols\_A)

B = get\_matrix(rows\_B, cols\_B)

# Multiply matrices A and B using multithreading

start = time.time()

C = matrix\_multiply\_multithreaded(A, B)

end = time.time()

# Print resulting matrix C

print("Resulting Matrix C (A \* B):")

for row in C:

print(row)

print(f"Time taken: {end - start} seconds")

**OUTPUT:**

Enter the number of rows for matrix A: 3

Enter the number of columns for matrix A: 3

Enter the number of rows for matrix B (must be equal to columns of A): 3

Enter the number of columns for matrix B: 3

Enter the matrix (rows: 3, cols: 3):

Row 1: 1 2 3

Row 2: 4 5 6

Row 3: 7 8 9

Enter the matrix (rows: 3, cols: 3):

Row 1: 9 8 7

Row 2: 6 5 4

Row 3: 3 2 1

Resulting Matrix C (A \* B):

[30, 24, 18]

[84, 69, 54]

[138, 114, 90]

Time taken: 0.0009872913360595703 seconds

**Mini Project 2**

**Problem Statement:** Implement merge sort and multithreaded merge sort. Compare time required by both the algorithms. Also analyze the performance of each algorithm for the best case and the worst case.

import time

# Merge function to merge two sorted halves

def merge(arr, left, mid, right):

n1 = mid - left + 1

n2 = right - mid

# Create temporary arrays

L = arr[left:left+n1]

R = arr[mid+1:mid+1+n2]

i = j = 0

k = left

# Merge the arrays

while i < n1 and j < n2:

if L[i] <= R[j]:

arr[k] = L[i]

i += 1

else:

arr[k] = R[j]

j += 1

k += 1

# Copy remaining elements of L[]

while i < n1:

arr[k] = L[i]

i += 1

k += 1

# Copy remaining elements of R[]

while j < n2:

arr[k] = R[j]

j += 1

k += 1

# Merge sort function

def merge\_sort(arr, left, right):

if left < right:

mid = (left + right) // 2

# Sort first and second halves

merge\_sort(arr, left, mid)

merge\_sort(arr, mid + 1, right)

# Merge the sorted halves

merge(arr, left, mid, right)

def get\_array():

while True:

try:

# Get user input and convert it into a list of integers

arr = list(map(int, input("Enter numbers to sort (space-separated): ").split()))

return arr

except ValueError:

print("Invalid input. Please enter integers only.")

# Main execution

if \_\_name\_\_ == "\_\_main\_\_":

# Get array from user

arr = get\_array()

start = time.time()

merge\_sort(arr, 0, len(arr) - 1)

end = time.time()

print("Sorted array:", arr)

print(f"Time taken (Single-threaded): {end - start:.6f} seconds")

**OUTPUT:**

Enter numbers to sort (space-separated): 38 27 43 3 9 82 10

Sorted array: [3, 9, 10, 27, 38, 43, 82]

Time taken (Single-threaded): 0.000000 seconds

import time

import threading

# Merge function to merge two sorted halves

def merge(arr, left, mid, right):

n1 = mid - left + 1

n2 = right - mid

# Create temporary arrays

L = arr[left:left+n1]

R = arr[mid+1:mid+1+n2]

i = j = 0

k = left

# Merge the arrays

while i < n1 and j < n2:

if L[i] <= R[j]:

arr[k] = L[i]

i += 1

else:

arr[k] = R[j]

j += 1

k += 1

# Copy remaining elements of L[]

while i < n1:

arr[k] = L[i]

i += 1

k += 1

# Copy remaining elements of R[]

while j < n2:

arr[k] = R[j]

j += 1

k += 1

# Threaded merge sort function

def threaded\_merge\_sort(arr, left, right):

if left < right:

mid = (left + right) // 2

# Create threads for sorting the two halves

left\_thread = threading.Thread(target=threaded\_merge\_sort, args=(arr, left, mid))

right\_thread = threading.Thread(target=threaded\_merge\_sort, args=(arr, mid + 1, right))

# Start threads

left\_thread.start()

right\_thread.start()

# Wait for both threads to complete

left\_thread.join()

right\_thread.join()

# Merge the sorted halves

merge(arr, left, mid, right)

def get\_array():

while True:

try:

# Get user input and convert it into a list of integers

arr = list(map(int, input("Enter numbers to sort (space-separated): ").split()))

return arr

except ValueError:

print("Invalid input. Please enter integers only.")

# Main execution

if \_\_name\_\_ == "\_\_main\_\_":

# Get array from user

arr = get\_array()

start = time.time()

threaded\_merge\_sort(arr, 0, len(arr) - 1)

end = time.time()

print("Sorted array (Multithreaded):", arr)

print(f"Time taken (Multithreaded): {end - start:.6f} seconds")

**OUTPUT:**

Enter numbers to sort (space-separated): 38 27 43 3 9 82 10

Sorted array (Multithreaded): [3, 9, 10, 27, 38, 43, 82]

Time taken (Multithreaded): 0.003768 seconds

**Mini Project 3**

**Problem Statement :**

Mini Project - Implement the Naive string-matching algorithm and Rabin-Karp algorithm for string matching. Observe difference in working of both the algorithms for the same input.

**# Code for Naive String Matching Algorithm**

def naive\_string\_matcher(text, pattern):

n = len(text)

m = len(pattern)

result = []

# Slide the pattern over text one by one

for i in range(n - m + 1):

# Check the substring text[i:i+m]

match = True

for j in range(m):

if text[i + j] != pattern[j]:

match = False

break

if match:

result.append(i)

return result

def get\_input():

text = input("Enter the text: ")

pattern = input("Enter the pattern to search for: ")

return text, pattern

# Main execution

if \_\_name\_\_ == "\_\_main\_\_":

# Get user input for text and pattern

text, pattern = get\_input()

result = naive\_string\_matcher(text, pattern)

print(f"Pattern '{pattern}' found at positions (Naive): {result}")

**OUTPUT:**

Enter the text: ABAAABCD

Enter the pattern to search for: ABC

Pattern 'ABC' found at positions (Naive): [4]

**# Code for Rabin-Karp Algorithm**

def rabin\_karp(text, pattern, q=101): # q is a prime number

d = 256 # Number of characters in the input alphabet

n = len(text)

m = len(pattern)

result = []

h = 1 # Hash factor

p = 0 # Hash value for pattern

t = 0 # Hash value for text

# Precompute h = (d^(m-1)) % q

for i in range(m - 1):

h = (h \* d) % q

# Compute the hash value of the pattern and first window of text

for i in range(m):

p = (d \* p + ord(pattern[i])) % q

t = (d \* t + ord(text[i])) % q

# Slide the pattern over text one by one

for i in range(n - m + 1):

# If the hash values match, then only check for characters one by one

if p == t:

if text[i:i + m] == pattern:

result.append(i)

# Calculate hash value for next window of text

if i < n - m:

t = (d \* (t - ord(text[i]) \* h) + ord(text[i + m])) % q

if t < 0:

t += q

return result

# Main execution

if \_\_name\_\_ == "\_\_main\_\_":

# Get user input for text and pattern

text, pattern = get\_input()

result = rabin\_karp(text, pattern)

print(f"Pattern '{pattern}' found at positions (Rabin-Karp): {result}")

**OUTPUT:**

Enter the text: ABAAABCD

Enter the pattern to search for: ABC

Pattern 'ABC' found at positions (Rabin-Karp): [4]

**Mini Project 4**

**Problem Statement :**

Mini Project - Different exact and approximation algorithms for Travelling-Sales-Person Problem

import itertools

import sys

import time

def calculate\_total\_distance(route, distance\_matrix):

total = 0

for i in range(len(route)):

total += distance\_matrix[route[i]][route[(i + 1) % len(route)]]

return total

def brute\_force\_tsp(distance\_matrix):

n = len(distance\_matrix)

cities = list(range(n))

min\_distance = sys.maxsize

best\_route = []

# Generate all possible permutations of cities

for perm in itertools.permutations(cities):

current\_distance = calculate\_total\_distance(perm, distance\_matrix)

if current\_distance < min\_distance:

min\_distance = current\_distance

best\_route = perm

return best\_route, min\_distance

def get\_distance\_matrix(num\_cities):

distance\_matrix = []

print("Enter the distance matrix row by row:")

for i in range(num\_cities):

row = list(map(int, input(f"Row {i + 1}: ").split()))

if len(row) != num\_cities:

print(f"Error: Row {i + 1} must contain exactly {num\_cities} distances.")

sys.exit(1)

distance\_matrix.append(row)

return distance\_matrix

# Main execution

if \_\_name\_\_ == "\_\_main\_\_":

# User input for number of cities

num\_cities = int(input("Enter the number of cities: "))

distance\_matrix = get\_distance\_matrix(num\_cities)

start\_time = time.time()

route, distance = brute\_force\_tsp(distance\_matrix)

end\_time = time.time()

print("Optimal Route (Brute-Force):", route)

print("Minimum Distance:", distance)

print(f"Time taken: {end\_time - start\_time:.4f} seconds")

**OUTPUT:**

Enter the number of cities: 4

Enter the distance matrix row by row:

Row 1: 0 10 15 20

Row 2: 10 0 35 25

Row 3: 15 35 0 30

Row 4: 20 25 30 0

Optimal Route (Brute-Force): (0, 1, 3, 2)

Minimum Distance: 80

Time taken: 0.0000 seconds

import time

def nearest\_neighbor\_tsp(distance\_matrix, start=0):

n = len(distance\_matrix)

unvisited = set(range(n))

unvisited.remove(start)

route = [start]

total\_distance = 0

current = start

while unvisited:

next\_city = min(unvisited, key=lambda city: distance\_matrix[current][city])

total\_distance += distance\_matrix[current][next\_city]

route.append(next\_city)

current = next\_city

unvisited.remove(next\_city)

# Return to start

total\_distance += distance\_matrix[current][start]

route.append(start)

return route, total\_distance

def get\_distance\_matrix(num\_cities):

distance\_matrix = []

print("Enter the distance matrix row by row:")

for i in range(num\_cities):

row = list(map(int, input(f"Row {i + 1}: ").split()))

if len(row) != num\_cities:

print(f"Error: Row {i + 1} must contain exactly {num\_cities} distances.")

sys.exit(1)

distance\_matrix.append(row)

return distance\_matrix

# Main execution

if \_\_name\_\_ == "\_\_main\_\_":

# User input for number of cities

num\_cities = int(input("Enter the number of cities: "))

distance\_matrix = get\_distance\_matrix(num\_cities)

start\_time = time.time()

route, distance = nearest\_neighbor\_tsp(distance\_matrix)

end\_time = time.time()

print("Route (Nearest Neighbor):", route)

print("Total Distance:", distance)

print(f"Time taken: {end\_time - start\_time:.6f} seconds")

**OUTPUT:**

Enter the number of cities: 4

Enter the distance matrix row by row:

Row 1: 0 10 15 20

Row 2: 10 0 35 25

Row 3: 15 35 0 30

Row 4: 20 25 30 0

Route (Nearest Neighbor): [0, 1, 3, 2, 0]

Total Distance: 80

Time taken: 0.000000 seconds