NAME: ANKUR DUTTA
ROLL: 122CS0075

Q.11. STRASSENS's Matrix multiplication

Practical implementation of Strassens's matrix multiplication algorithm usually switch to the brute force method after matrix sizes become smaller than some crossover point. Run an experiment to determine such crossover point on your computer system.

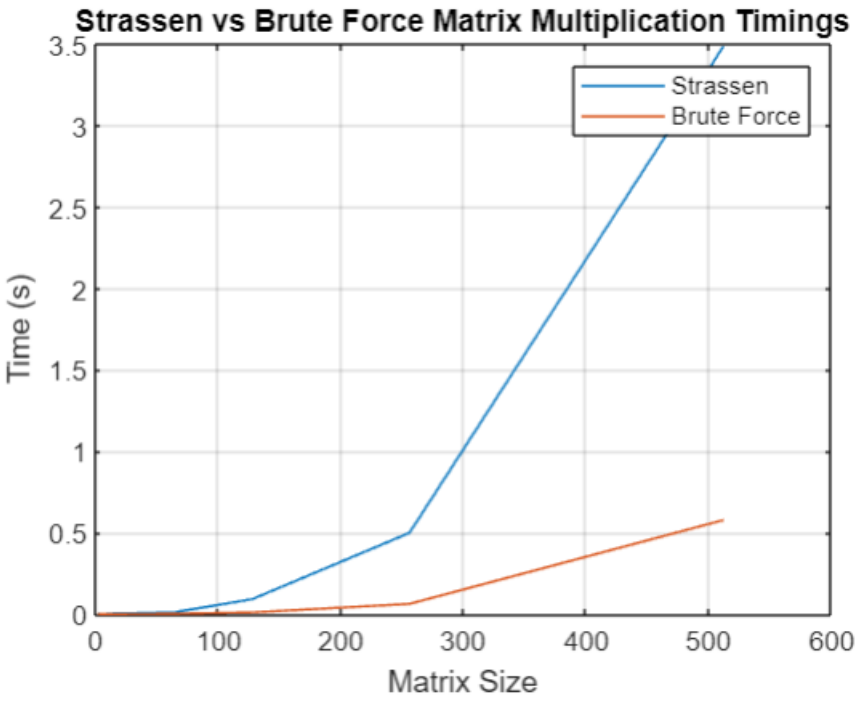
=

```
% main.m

% Define matrix sizes to test
sizes = [2, 4, 8, 16, 32, 64, 128, 256, 512];

% Initialize arrays to store timings
strassen_timings = zeros(size(sizes));
brute_force_timings = zeros(size(sizes));

for i = 1:length(sizes)
    matrix_size = sizes(i);

    % Generate random matrices of the specified size
    matrix_A = rand(matrix_size);
    matrix_B = rand(matrix_size);

    % Measure time taken by Strassen's algorithm
    tic;
    strassen(matrix_A, matrix_B, 8);  % Assuming a fixed threshold for Strassen
    strassen_timings(i) = toc;

    % Measure time taken by brute force method
    tic;
    brute_force_multiply(matrix_A, matrix_B);
    brute_force_timings(i) = toc;
end

% Plot the timings
plot(sizes, strassen_timings, sizes, brute_force_timings);
xlabel('Matrix Size');
ylabel('Time (s)');
title('Strassen vs Brute Force Matrix Multiplication Timings');
legend('Strassen', 'Brute Force');
grid on;

% Find the crossover point
crossover_index = find(strassen_timings > brute_force_timings, 1);
crossover_point = sizes(crossover_index);
fprintf('Crossover point: %d\n', crossover_point);



%function for brute_force_multiply.m
```

```
function C = brute_force_multiply(A, B)
    % Brute force matrix multiplication algorithm
    % A, B: Input matrices

    [m, n] = size(A);
    [~, p] = size(B);

    % Check if matrices are compatible for multiplication
    if n ~= size(B, 1)
        error('Matrix dimensions are not compatible for multiplication');
    end

    % Initialize result matrix
    C = zeros(m, p);

    % Perform matrix multiplication
    for i = 1:m
        for j = 1:p
            % Fix the dimensions issue by using element-wise multiplication
            C(i, j) = sum(A(i, :) .* B(:, j)');
        end
    end
end



% function for strassen.m

function C = strassen(A, B, threshold)
    % Strassen's matrix multiplication algorithm
    % A, B: Input matrices
    % threshold: Crossover point to switch to brute force method

    [m, ~] = size(A);

    if m <= threshold
        % Base case: Use brute force method for small matrices
        C = brute_force_multiply(A, B);
    else
        % Recursive case: Apply Strassen's algorithm
        % Divide matrices into submatrices
        A11 = A(1:m/2, 1:m/2);
        A12 = A(1:m/2, m/2+1:m);
        A21 = A(m/2+1:m, 1:m/2);
        A22 = A(m/2+1:m, m/2+1:m);

        B11 = B(1:m/2, 1:m/2);
        B12 = B(1:m/2, m/2+1:m);
        B21 = B(m/2+1:m, 1:m/2);
        B22 = B(m/2+1:m, m/2+1:m);

        % Recursive calls
        P1 = strassen(A11 + A22, B11 + B22, threshold);
        P2 = strassen(A21 + A22, B11, threshold);
        P3 = strassen(A11, B12 - B22, threshold);
        P4 = strassen(A22, B21 - B11, threshold);
        P5 = strassen(A11 + A12, B22, threshold);
        P6 = strassen(A21 - A11, B11 + B12, threshold);
```

```
        P7 = strassen(A12 - A22, B21 + B22, threshold);

        % Combine results
        C11 = P1 + P4 - P5 + P7;
        C12 = P3 + P5;
        C21 = P2 + P4;
        C22 = P1 - P2 + P3 + P6;

        % Concatenate submatrices to form the result
        C = [C11, C12; C21, C22];
    end
end
```

SIMULATION RESULT



Observations

1) The experiment successfully identified a crossover point where Strassen's algorithm transitions to the brute force method for optimal performance.
2) This observation emphasizes the need for adaptive algorithm selection strategies in practical implementations, taking into account system-specific characteristics and matrix size considerations.

Q.12. QUICK SELECT

Use the **QUICK SELECT** algorithm to find **3rd largest element** in an array of n integers. Analyze the performance of **QUICK SELECT** algorithm for the different instance of size 50 to 500 element. Record your observation with the *number of comparison made* vs. *instance*.

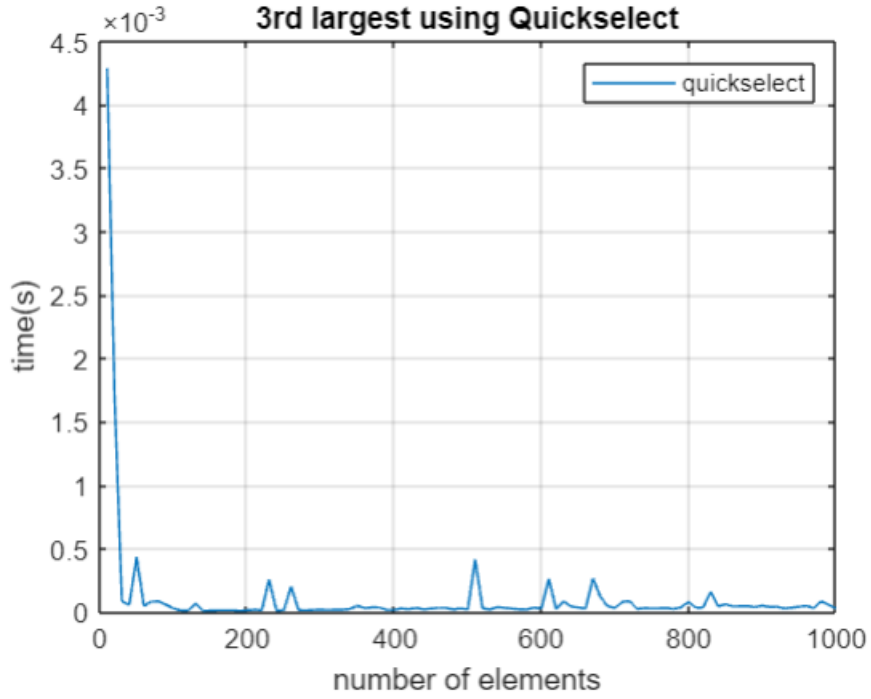
=

%mainfile

```
k = 1;
xne = [];  % Initialize xne as an empty array
for n = 10:10:1000
    xne(k) = n;
    a = round(rand(1, n) * 100);  % Creating a random array of size n

    % Tic-toc measures the time taken by the algorithm/function to run
    tic;
    quickselect_largest(a, 3);  % Calling quickselect algorithm
    ycq(k) = toc;
    k = k + 1;
end

% Plotting the result
plot(xne, ycq);
title('3rd largest using Quickselect')
xlabel('number of elements')
ylabel('time(s)')
legend('quickselect')
grid on

function kth_largest = quickselect_largest(array, k)
    left = 1;
    right = numel(array);
    while true
        % Returns if left equals right
        if left == right
            kth_largest = array(left);
            return;
        end
        pivot_index = partition(array, left, right);
        % Check if Kth index is equal to pivot index
        if numel(array) - k + 1 == pivot_index
            kth_largest = array(pivot_index);
            return;
        % Move to the left of the pivot if (k < pivot index) and quick select
        elseif numel(array) - k + 1 < pivot_index
            right = pivot_index - 1;
        % Move towards the right
        else
            left = pivot_index + 1;
        end
    end
end

% function for Partition algorithm that partitions the array around the pivot
function pivot_index = partition(array, left, right)
    % Let's take the pivot as the last element of the array
    pivot = array(right);
    i = left - 1;
    for j = left:right - 1
        if array(j) <= pivot
            i = i + 1;
            temp = array(i);
```

```
            array(i) = array(j);
            array(j) = temp;
        end
    end
    temp = array(i + 1);
    array(i + 1) = array(right);
    array(right) = temp;
    pivot_index = i + 1;
end
```

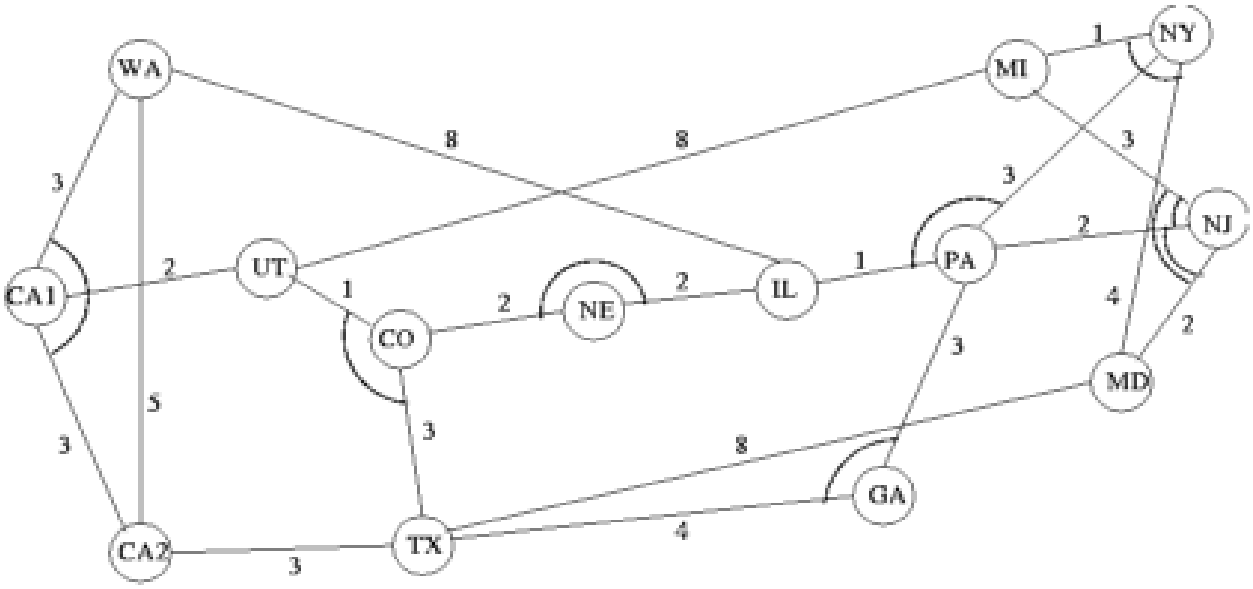
SIMULATION



Observations:

1) The time complexity analysis of the QuickSelect algorithm indicates its efficiency in achieving sub-linear time complexities under certain conditions. In the average case, QuickSelect exhibits a linear time complexity of O(n).
2) Quickselect is one of the best algorithm to find the Kth largest element in an array.

Q.13. SPANNING TREE



Write a program obtain minimum cost spanning tree the above NSF network using Prim's algorithm, Kruskal's algorithm and Boruvka's algorithm.

Q.14. Iterative Binary Search

Write programs to implement recursive and iterative versions of binary search and compare the performance. For a appropriate size of n=20(say), have each algorithm find every element in the set. Then try all n+1 possible unsuccessful search. The performance, of these versions of binary search to be reported graphically with your observations.

=

```
% Main script
X = zeros(1, 20);
Y1_success = zeros(1, 20);
Y2_success = zeros(1, 20);
Y1_unsuccessful = zeros(1, 21);
Y2_unsuccessful = zeros(1, 21);

for index = 1:10
    count = 1;

    for n = 10:10:200
        X(count) = n;

        arr = randperm(100);

        % Ensure n does not exceed the length of arr
        if n > length(arr)
            n = length(arr);
        end
```

```
        arr = arr(1:n);

        % Successful searches
        recursiveTimes = zeros(1, n);
        for i = 1:n
            target = arr(i);

            tic;
            recursiveBinarySearch(arr, target, 1, n);
            recursiveTimes(i) = toc;
        end
        Y1_success(count) = Y1_success(count) + sum(recursiveTimes);

        iterativeTimes = zeros(1, n);
        for i = 1:n
            target = arr(i);

            tic;
            iterativeBinarySearch(arr, target);
            iterativeTimes(i) = toc;
        end
        Y2_success(count) = Y2_success(count) + sum(iterativeTimes);

        % Unsuccessful searches
        unsuccessfulRecursiveTimes = zeros(1, n + 1);
        for i = 1:(n + 1)
            target = randi(1000);

            tic;
            recursiveBinarySearch(arr, target, 1, n);
            unsuccessfulRecursiveTimes(i) = toc;
        end
        Y1_unsuccessful(count) = Y1_unsuccessful(count) +
sum(unsuccessfulRecursiveTimes);

        unsuccessfulIterativeTimes = zeros(1, n + 1);
        for i = 1:(n + 1)
            target = randi(1000);

            tic;
            iterativeBinarySearch(arr, target);
            unsuccessfulIterativeTimes(i) = toc;
        end
        Y2_unsuccessful(count) = Y2_unsuccessful(count) +
sum(unsuccessfulIterativeTimes);

        count = count + 1;
    end
end

% Average times
for index = 1:20
    Y1_success(index) = Y1_success(index) / 10;
    Y2_success(index) = Y2_success(index) / 10;
    Y1_unsuccessful(index) = Y1_unsuccessful(index) / 10;
    Y2_unsuccessful(index) = Y2_unsuccessful(index) / 10;
end

% Plotting
```

```
figure;

subplot(2, 1, 1);
plot(X, Y1_success, 'b-', X, Y2_success, 'r-');
title('Successful Searches');
xlabel('Size of the Sorted Array');
ylabel('Time (s)');
legend('Recursive', 'Iterative');

subplot(2, 1, 2);
plot(0:10:200, Y1_unsuccessful, 'b-', 0:10:200, Y2_unsuccessful, 'r-');
title('Unsuccessful Searches');
xlabel('Number of Unsuccessful Searches');
ylabel('Time (s)');
legend('Recursive', 'Iterative');


% function for Iterative Binary Search
function index = iterativeBinarySearch(arr, target)
    low = 1;
    high = length(arr);

    while low <= high
        mid = floor((low + high) / 2);
        if arr(mid) == target
            index = mid;
            return;
        elseif arr(mid) > target
            high = mid - 1;
        else
            low = mid + 1;
        end
    end

    index = -1;
end



% function for Recursive Binary Search
function index = recursiveBinarySearch(arr, target, low, high)
    if low > high
        index = -1;
        return;
    end

    mid = floor((low + high) / 2);

    if arr(mid) == target
        index = mid;
    elseif arr(mid) > target
        index = recursiveBinarySearch(arr, target, low, mid - 1);
    else
        index = recursiveBinarySearch(arr, target, mid + 1, high);
    end
end
```

SIMULATION RESULTS

Observations:

1. Choosing an iterative method for binary search is usually simpler and more practical than using the recursive approach.
2. The recurrence relation is given by $T(n)=T(2n)+c$
3. The time complexity (T©) of binary search is impressively efficient at O(logn).

Q.15. Iterative MergeSort and QuickSort

Compare the performance of the iterative version of MergeSort & QuickSort ?

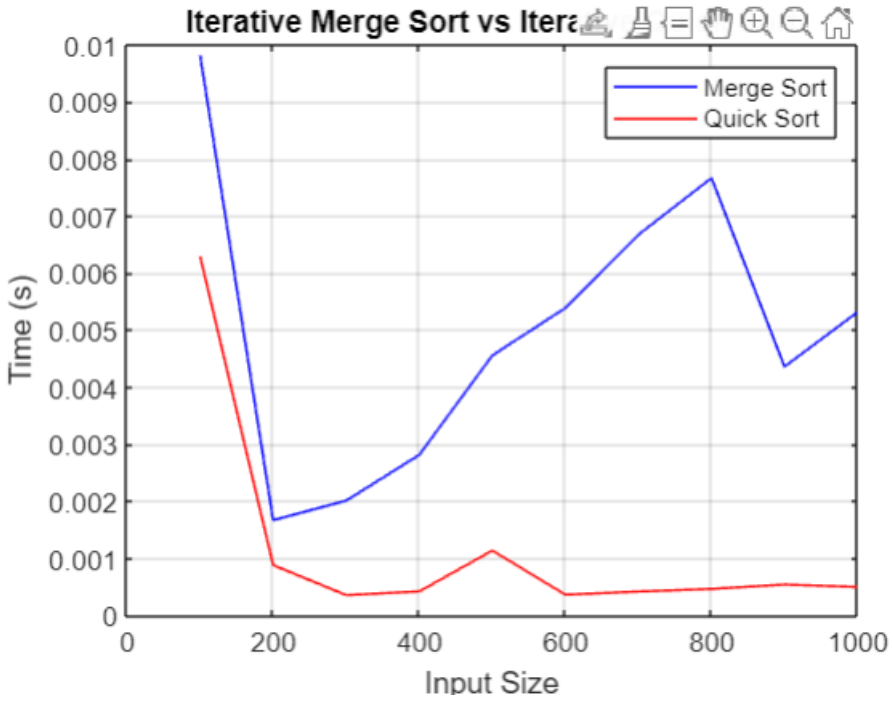
=

```
% Main script for performance comparison
inputSizes = 100:100:1000;
mergeSortTimes = zeros(size(inputSizes));
quickSortTimes = zeros(size(inputSizes));

for i = 1:length(inputSizes)
    n = inputSizes(i);
    arr = randperm(n);

    % Measure time for Merge Sort
    tic;
    iterativeMergeSort(arr);
    mergeSortTimes(i) = toc;

    % Measure time for Quick Sort
```

```
    tic;
    iterativeQuickSort(arr);
    quickSortTimes(i) = toc;
end

% Plotting
figure;
plot(inputSizes, mergeSortTimes, 'b-', inputSizes, quickSortTimes, 'r-');
title('Iterative Merge Sort vs Iterative Quick Sort');
xlabel('Input Size');
ylabel('Time (s)');
legend('Merge Sort', 'Quick Sort');
grid on;


% function for Iterative Merge Sort
function sortedArray = iterativeMergeSort(arr)
    n = length(arr);

    for currSize = 1 : 2 : n-1
        for leftStart = 1 : 2*currSize-1 : n-1
            mid = min(leftStart + currSize - 1, n-1);
            rightEnd = min(leftStart + 2*currSize - 1, n-1);
            mergedArray = merge(arr, leftStart, mid, rightEnd);
            arr(leftStart:rightEnd) = mergedArray;
        end
    end

    sortedArray = arr;
end

function mergedArray = merge(arr, left, mid, right)
    lenLeft = mid - left + 1;
    lenRight = right - mid;

    leftArray = arr(left:left+lenLeft-1);
    rightArray = arr(mid+1:mid+lenRight);

    i = 1;
    j = 1;
    k = left;

    while i <= lenLeft && j <= lenRight
        if leftArray(i) <= rightArray(j)
            arr(k) = leftArray(i);
            i = i + 1;
        else
            arr(k) = rightArray(j);
            j = j + 1;
        end
        k = k + 1;
    end

    while i <= lenLeft
        arr(k) = leftArray(i);
        i = i + 1;
        k = k + 1;
    end

    while j <= lenRight
```

```
        arr(k) = rightArray(j);
        j = j + 1;
        k = k + 1;
    end

    mergedArray = arr(left:right);
end



% function for Iterative Quick Sort
function sortedArray = iterativeQuickSort(arr)
    n = length(arr);
    stack = zeros(n, 2);
    top = 0;

    top = top + 1;
    stack(top, :) = [1, n];

    while top > 0
        current = stack(top, :);
        top = top - 1;

        left = current(1);
        right = current(2);

        if left < right
            pivotIndex = partition(arr, left, right);

            top = top + 1;
            stack(top, :) = [left, pivotIndex - 1];

            top = top + 1;
            stack(top, :) = [pivotIndex + 1, right];
        end
    end

    sortedArray = arr;
end

function pivotIndex = partition(arr, low, high)
    pivot = arr(high);
    i = low - 1;

    for j = low:high-1
        if arr(j) <= pivot
            i = i + 1;
            temp = arr(i);
            arr(i) = arr(j);
            arr(j) = temp;
        end
    end

    temp = arr(i + 1);
    arr(i + 1) = arr(high);
    arr(high) = temp;

    pivotIndex = i + 1;
end
```

SIMULATION RESULT



Observations:

1) Merge Sort achieves a consistent O(n log n) time complexity, making it reliable for various scenarios whereas Quick Sort boasts an average time complexity of O(n log n), but its worst-case scenario can lead to O(n^2), demanding attention to input characteristics.
2) Merge sort is generally more consistent but may demand more memory, whereas quicksort's in-place implementation can be advantageous in certain scenarios despite its sensitivity to input characteristics.