**Q.16. Selection Problem**

Write the program with three different selection algorithm to find kth largest element from a given unsorted linear **array** and compare their performance?

=

% main.m

% Set the array size and k values

array\_size = 1000;

k\_values = [5, 10, 50, 100]; % Example: Different values of k

% Initialize arrays to store timings

quick\_select\_timings = zeros(size(k\_values));

heap\_select\_timings = zeros(size(k\_values));

for j = 1:length(k\_values)

k = k\_values(j);

% Generate a random unsorted array

input\_array = rand(1, array\_size);

% Measure execution time for QuickSelect

tic;

kth\_largest\_quick = quick\_select(input\_array, k);

quick\_select\_timings(j) = toc;

% Measure execution time for HeapSelect

tic;

kth\_largest\_heap = heap\_select(input\_array, k);

heap\_select\_timings(j) = toc;

end

% Plot the timings

plot(k\_values, quick\_select\_timings, k\_values, heap\_select\_timings);

xlabel('Value of k');

ylabel('Time (s)');

title('Selection Algorithm Timings for kth Largest Element');

legend('QuickSelect', 'HeapSelect');

grid on;

#functions for quick\_select.m

% selection\_algorithms.m

function kth\_largest = quick\_select(array, k)

% QuickSelect algorithm to find the kth largest element

% This implementation uses the partitioning step similar to QuickSort

% Check if k is within valid range

if k < 1 || k > length(array)

error('Invalid value of k');

end

kth\_largest = quick\_select\_recursive(array, 1, length(array), k);

end

function kth\_largest = quick\_select\_recursive(array, low, high, k)

% Recursive function for QuickSelect

if low == high

% Base case: Only one element in the subarray

kth\_largest = array(low);

return;

end

% Partition the array and get the index of the pivot

pivot\_index = quick\_partition(array, low, high);

% Calculate the position of the pivot in the sorted array

pivot\_position = pivot\_index - low + 1;

if k == pivot\_position

% The pivot is the kth largest element

kth\_largest = array(pivot\_index);

elseif k < pivot\_position

% The kth largest element is in the left subarray

kth\_largest = quick\_select\_recursive(array, low, pivot\_index - 1, k);

else

% The kth largest element is in the right subarray

kth\_largest = quick\_select\_recursive(array, pivot\_index + 1, high, k - pivot\_position);

end

end

function pivot\_index = quick\_partition(array, low, high)

% Partition the array and return the index of the pivot

% Choose the rightmost element as the pivot

pivot = array(high);

% Initialize the index to track the position of the smaller elements

i = low - 1;

% Iterate through the subarray

for j = low:high-1

if array(j) >= pivot

% Swap array elements if greater than or equal to the pivot

i = i + 1;

temp = array(i);

array(i) = array(j);

array(j) = temp;

end

end

% Swap the pivot element into its correct position

temp = array(i + 1);

array(i + 1) = array(high);

array(high) = temp;

% Return the index of the pivot

pivot\_index = i + 1;

end

#function for heapselect

function kth\_largest = heap\_select(array, k)

% HeapSelect algorithm to find the kth largest element

% Check if k is within valid range

if k < 1 || k > length(array)

error('Invalid value of k');

end

% Create a max heap from the first k elements

max\_heap = array(1:k);

max\_heap = heapify(max\_heap);

% Iterate through the remaining elements

for i = k+1:length(array)

if array(i) > max\_heap(1)

% Replace the root of the heap with the current element

max\_heap = heapreplace(max\_heap, array(i));

end

end

% The kth largest element is the root of the max heap

kth\_largest = max\_heap(1);

end

function heap = heapify(array)

% Helper function to heapify an array

heap = array;

n = length(array);

for i = fix(n/2):-1:1

heap = heapdown(heap, i, n);

end

end

function heap = heapreplace(heap, value)

% Helper function to replace the root of the heap with a new value

heap(1) = value;

heap = heapdown(heap, 1, length(heap));

end

function heap = heapdown(heap, i, n)

% Helper function for heapify and heapreplace

while 2\*i <= n

j = 2\*i;

if j < n && heap(j) < heap(j + 1)

j = j + 1;

end

if heap(i) >= heap(j)

break;

end

temp = heap(i);

heap(i) = heap(j);

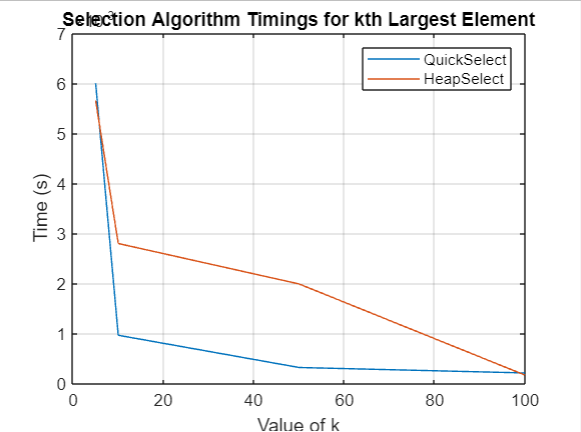
heap(j) = temp;

i = j;

end

end

**SIMULATION**



Observations:

1. The execution time of the HeapSelect algorithm tends to be higher than that of QuickSelect.
2. QuickSelect is generally more efficient for finding the kth largest element in an unsorted array due to its average-case time complexity of O(n). It employs a partitioning strategy similar to QuickSort.
3. On the other hand, HeapSelect builds a max heap from the initial **k** elements and then iterates through the remaining elements to maintain the heap property. While it has a time complexity of O(n log k), it may exhibit slightly higher overhead compared to quickselect for small values of **k**.

**Q.17. BINOMIAL COEFFICIENTS**

Computing the **binomial coefficients** *C*(*n*, *k*) defined by the following recursive formula:

Write the program with three different algorithm to compute **binomial coefficients** *C*(*n*, *k*) and compare them?

**Q.18. 0-1 KNAPSACK PROBLEM**

Write a program that computes optimal solution to the 0–1 Knapsack Problem using dynamic programming? You may test your program with the following example:

There are *n* = 5 objects with integer weights *w*[1..5] = {1,2,5,6,7}, and values *v*[1..5] = {1,6,18,22,28}. Assuming a knapsack capacity of 11). Use your own generated dataset to present a comparisons with, greedy, randomized and dynamic programming approach.

=

% mainfile.m

% Set up the dataset

weights = [1, 2, 5, 6, 7];

values = [1, 6, 18, 22, 28];

capacity = 11;

% Set the array size and test values

array\_size = 1000;

test\_values = [5, 10, 50, 100]; % Example: Different values for testing

% Initialize arrays to store timings

dynamic\_programming\_timings = zeros(size(test\_values));

greedy\_timings = zeros(size(test\_values));

randomized\_timings = zeros(size(test\_values));

for j = 1:length(test\_values)

test\_value = test\_values(j);

% Repeat the test multiple times for more accurate timings

for repeat = 1:10

% Measure execution time for Dynamic Programming

tic;

knapsack\_dynamic\_programming(weights, values, capacity);

dynamic\_programming\_timings(j) = dynamic\_programming\_timings(j) + toc;

% Measure execution time for Greedy

tic;

knapsack\_greedy(weights, values, capacity);

greedy\_timings(j) = greedy\_timings(j) + toc;

% Measure execution time for Randomized

tic;

knapsack\_randomized(weights, values, capacity);

randomized\_timings(j) = randomized\_timings(j) + toc;

end

% Average the timings over multiple repeats

dynamic\_programming\_timings(j) = dynamic\_programming\_timings(j) / 10;

greedy\_timings(j) = greedy\_timings(j) / 10;

randomized\_timings(j) = randomized\_timings(j) / 10;

end

% Plot the timings

plot(test\_values, dynamic\_programming\_timings, test\_values, greedy\_timings, test\_values, randomized\_timings);

xlabel('Test Values');

ylabel('Time (s)');

title('Knapsack Algorithm Timings');

legend('Dynamic Programming', 'Greedy', 'Randomized');

grid on;

#function for knapsack\_greedy

% functions.m

function result = knapsack\_greedy(weights, values, capacity)

n = length(weights);

ratios = values ./ weights; % Calculate value-to-weight ratios

% Sort items based on the ratios in descending order

[~, order] = sort(ratios, 'descend');

total\_value = 0;

total\_weight = 0;

result = zeros(1, n);

for i = 1:n

if total\_weight + weights(order(i)) <= capacity

total\_weight = total\_weight + weights(order(i));

total\_value = total\_value + values(order(i));

result(order(i)) = 1; % Include the item in the knapsack

end

end

end

# function for knapsack randomized

function result = knapsack\_randomized(weights, values, capacity)

n = length(weights);

iterations = 1000; % Number of iterations for the randomized approach

best\_solution = zeros(1, n);

best\_value = 0;

for iter = 1:iterations

current\_solution = randi([0, 1], 1, n); % Generate a random solution

current\_value = sum(current\_solution .\* values);

current\_weight = sum(current\_solution .\* weights);

% Check if the current solution is valid and has higher value

if current\_weight <= capacity && current\_value > best\_value

best\_solution = current\_solution;

best\_value = current\_value;

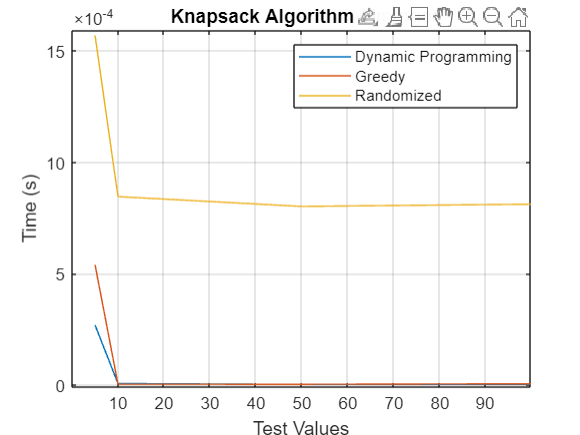
end

end

result = best\_solution;

end

**SIMULATION\_RESULT**



**Conclusions:**

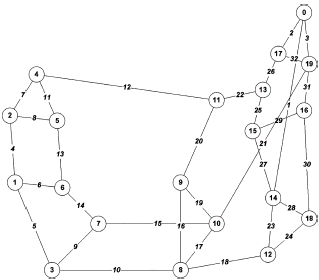
1. Dynamic programming is a robust choice, ensuring the optimal solution to the 0–1 Knapsack Problem, although at the cost of higher computational complexity.
2. The greedy algorithm, offering a quick and efficient solution, provides a reasonable approximation for the knapsack problem, making it suitable when computational efficiency is a priority over optimal accuracy.
3. The randomized algorithm introduces variability, offering a compromise between solution quality and computational efficiency, making it a valuable choice in scenarios where a balanced trade-off is essential for the 0–1 Knapsack

**Q.19. Matrix Chain Multiplication Comparision**

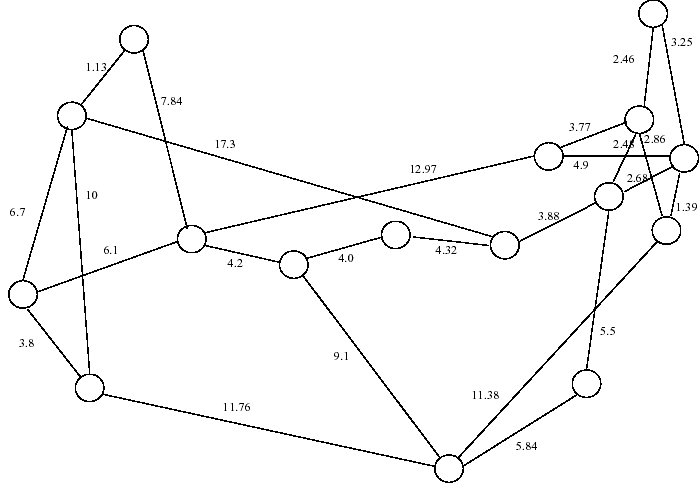
Given a matrix chain A1 …An with the dimension of each of the matrices given by the vector **p** = <12,21,65,18,24,93,121,16,41,31,47,5,47,29,76,18,72,15>. (n=17) Write and run both the dynamic programming and memorized versions of this algorithm to determine the minimum number of multiplications that are needed (use type ***longint***) and the factorization that produces this best case number of multiplications. Run each of the two programs over an appropriately large number of times (put each in a loop to run repeatedly x times) and obtain the times at the beginning and end of the run. Use these times to determine the comparative runtimes of the two algorithms.

**Q.20. PATHING ALGORITHM**

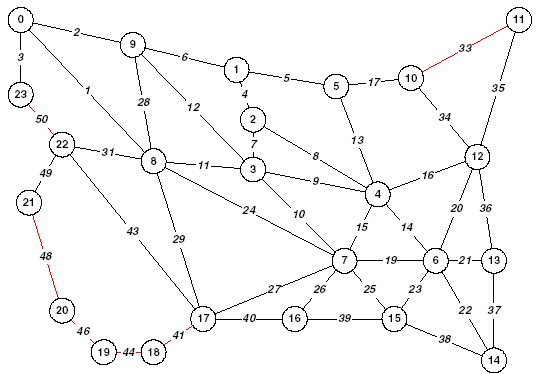
Use Floyd–Warshall algorithm (also known as Floyd's algorithm) to compute all pair shortest path for any one of the following standard network



ARPA Network



#### NSF Network



#### NATIONAL NETWORK

**Q.21. STRING MATCHING ALGORITHM {OPTIONAL}**

Given two strings P and T over the same alphabet set Σ, determine whether P occurs as a substring in T (or find in which position(s) P occurs as a substring in T). The strings P and T are called pattern and text respectively. Compare the efficiency of three string matching algorithms (Brute-Force Algorithm, Knuth-Morris-Pratt and Boyer-Moore Algorithm ) by varying pattern length [1-15] for n=5000.

**Upload your report [Source Codes + Simulation results +Observations] as a single pdf file with name as Roll No A4 to the specific assignment in MS Team**