MAYANK SRIVASTAVA

9922103143

WEEK 2

1-)

#include <iostream>

#include <vector>

using namespace std;

int mad(conststd::vector<int>&Ar, int x) {

int l = 0;

int h = Ar.size() - 1;

while (l <= h) {

int m = (l+ h) / 2;

// Find the first non-zero element in the array

while (l <= m &&Ar[m] == 0) {

m--;

}

// If the mid element is zero, update the search range accordingly

if (Ar[m] == 0) {

l = m + 1;

} else {

if (Ar[m] == x) {

return m; // Element found

} else if (Ar[m] < x) {

l = m + 1;

} else {

h = m - 1;

}

}

}

return -1; // Element not found

}

int main() {

vector<int>Ar = {11, 12, 13, 14, 15, 0, 0, 0, 0, 0}; // Example sorted array with zero-filled cells

int x = 3;

int position = mad(Ar, x);

if (position != -1) {

cout<< "Element " << x << " found at position " << position << "." <<endl;

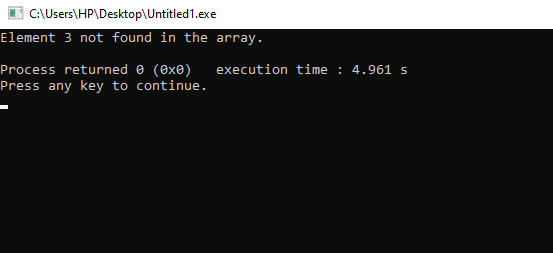
} else {

cout<< "Element " << x << " not found in the array." <<endl;

}

return 0;

}



2-)

#include <iostream>

#include <vector>

#include <unordered\_map>

using namespace std;

// Define the colours

enum C { ORANGE, PINK, WHITE };

// Define the pair structure

structNumberColourPair {

int number;

C colour;

};

voidsortPairsByColour(std::vector<NumberColourPair>& pairs) {

// Count the occurrences of each colour

unordered\_map<C, int>colourCount;

for (const auto& pair : pairs) {

colourCount[pair.colour]++;

}

// Calculate the starting position for each colour

intorangeStart = 0;

intpinkStart = orangeStart + colourCount[ORANGE];

intwhiteStart = pinkStart + colourCount[PINK];

// Create a temporary array to store the sorted pairs

vector<NumberColourPair>sortedPairs(pairs.size());

// Iterate through the original pairs and place them in the correct position in the sorted array

for (const auto& pair : pairs) {

int& position = (pair.colour == ORANGE) ? orangeStart : (pair.colour == PINK) ? pinkStart :whiteStart;

sortedPairs[position++] = pair;

}

// Copy the sorted pairs back to the original array

pairs = move(sortedPairs);

}

int main() {

// Example input

vector<NumberColourPair> pairs = {

{1, PINK},

{3, ORANGE},

{4, PINK},

{6, WHITE},

{9, ORANGE}

};

// Call the sorting function

sortPairsByColour(pairs);

// Print the sorted pairs

for (const auto& pair : pairs) {

cout<< "(" <<pair.number<< ", ";

switch (pair.colour) {

case ORANGE:

cout<< "orange";

break;

case PINK:

cout<< "pink";

break;

case WHITE:

cout<< "white";

break;

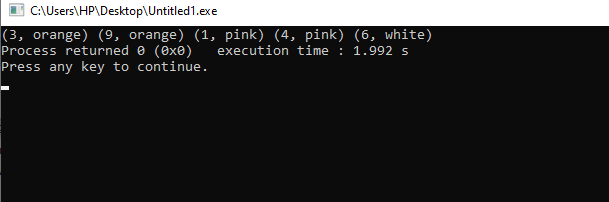
}

cout<< ") ";

}

return 0;

}



3-).

1. O(n)
2. O(n)
3. O(n)
4. O(n)
5. O(n^2)

f i) O(n^2)

ii) O(n\*m)

Q4-)

#include <iostream>

using namespace std;

voidtOH(int a, char p, char q, char r) {

if (a == 1) {

cout<< "Move disk 1 from " << p << " to " << r <<endl;

return;

}

tOH(a- 1, p, r, q);

cout<< "Move disk " << a << " from " << p << " to " << r<<endl;

tOH(a - 1, q, p, r);

}

int main() {

intnD;

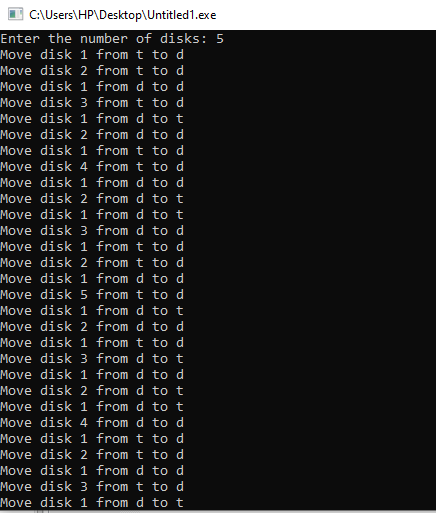
cout<< "Enter the number of disks: ";

cin>>nD;

tOH(nD, 'first', 'second', 'third');

return 0;

}



#### Time Complexity:

The time complexity of the Tower of Hanoi algorithm is O(2^n), where 'n' is the number of disks. This is because, for each additional disk, the number of moves doubles. The recurrence relation for time complexity is T(n) = 2\*T(n-1) + 1.

#### Space Complexity:

The space complexity is O(n) due to the recursive calls. In each recursive call, a constant amount of space is used for parameters and local variables. The maximum depth of the recursive call stack is 'n', so the space complexity is O(n).

#include <iostream>

using namespace std;

intfibonacci(int n) {

if (n <= 1) {

return n;

}

returnfibonacci(n - 1) + fibonacci(n - 2);

}

int main() {

int n;

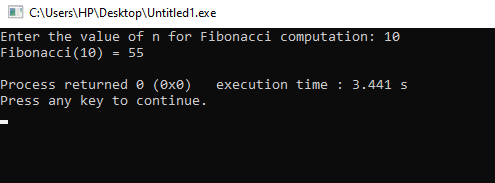
cout<< "Enter the value of n for Fibonacci computation: ";

cin>> n;

cout<< "Fibonacci(" << n << ") = " <<fibonacci(n) <<endl;

return 0;

}



#### Time Complexity:

The naive recursive Fibonacci algorithm has an exponential time complexity of O(2^n), where 'n' is the input value. This is because each Fibonacci number is computed by recursively summing the two previous Fibonacci numbers.

#### Space Complexity:

The space complexity is O(n) due to the maximum depth of the recursive call stack. The algorithm has O(n) recursion calls in the worst case, leading to O(n) space complexity.

It's important to note that the naive recursive implementations of both algorithms have exponential time complexity, making them inefficient for large inputs. For the Tower of Hanoi problem, there are more efficient algorithms with a time complexity of O(n), and for Fibonacci, dynamic programming or memoization techniques can significantly improve the time complexity to O(n) by avoiding redundant calculations.

Q5-)

Algo\_1(A [0 ... n-1])

{

if n = 2 and A[0] > A[1]

swap A[0] ↔ A[1]

else if n > 2

m = ⌈2n/3⌉

Algo\_1 (A [0 .. m − 1])

Algo\_1 (A [n – m .. n − 1])

Algo\_1 (A [0 .. m − 1])

}

Algo\_1, is a recursive algorithm for sorting an array A of size n. It is using a divide-and-conquer approach to perform the sorting. It divides the array into 3 parts and swaps the first two elements if the second element is greater than one.

T(n) = 3 \* T(⌈2n/3⌉) + O(1)

Space complexity : O(log n)