DAA PROGRAMS

1.**Write a program non-recursive and recursive program to calculate Fibonacci numbers andanalyze their time and space complexity.**

a.recursive

```
#include <iostream>

using namespace std;
int fibo(int n)
{
 if(n==0)
 {
 return 0;
 }
 if(n==1)
 {
 return 1;
 }
 return fibo(n-1)+fibo(n-2);
}
int main()
{
 int n;
```

```
 cin>>n;
 int f=fibo(n);
 cout<<f;


 return 0;
}
```

b.non recursive

```
#include <iostream>

using namespace
std; int main()
{
 int n;
 cin>>n;
 int fibo1=0;
 int fibo2=1;
 int fibo3;
 for(int i=2;i<=n;i++)
{
 fibo3=fibo1+fibo2;
fibo1=fibo2;
 fibo2=fibo3;
```

```
 }
 cout<<fibo3;
 return 0;
}
```

2.Write a program to implement Huffman Encoding using a greedy strategy.

```
#include <iostream>
#include <queue>
#include <map>
#include <string>
#include <sstream>

struct HuffmanNode {
    char data;
    int frequency;
    HuffmanNode* left;
    HuffmanNode* right;

    HuffmanNode(char data, int frequency) {
        this->data = data;
        this->frequency = frequency;
        this->left = nullptr;
        this->right = nullptr;
```

```
    }
};

struct CompareNodes {
    bool operator()(HuffmanNode* a, HuffmanNode* b) {
        return a->frequency > b->frequency;
    }
};

HuffmanNode* buildHuffmanTree(std::map<char, int>& frequencies) {
    std::priority_queue<HuffmanNode*, std::vector<HuffmanNode*>,
CompareNodes> minHeap;

    for (const auto& pair : frequencies) {
        minHeap.push(new HuffmanNode(pair.first, pair.second));
    }

    while (minHeap.size() > 1) {
        HuffmanNode* left = minHeap.top();
        minHeap.pop();
        HuffmanNode* right = minHeap.top();
        minHeap.pop();
```

```
        HuffmanNode* newNode = new HuffmanNode('\0', left->frequency +
right->frequency);
        newNode->left = left;
        newNode->right = right;
        minHeap.push(newNode);
    }

    return minHeap.top();
}

void generateHuffmanCodes(HuffmanNode* root, std::string code,
std::map<char, std::string>& huffmanCodes) {
    if (!root)
        return;

    if (root->data != '\0') {
        huffmanCodes[root->data] = code;
    }

    generateHuffmanCodes(root->left, code + "0", huffmanCodes);
    generateHuffmanCodes(root->right, code + "1", huffmanCodes);
}

int main() {
```

```
    std::map<char, int> frequencies;
    std::string input;

    std::cout << "Enter a string: ";
    getline(std::cin, input);

    for (char c : input) {
        frequencies[c]++;
    }

    HuffmanNode* root = buildHuffmanTree(frequencies);

    std::map<char, std::string> huffmanCodes;
    generateHuffmanCodes(root, "", huffmanCodes);

    std::cout << "Huffman Codes:\n";
    for (const auto& pair : huffmanCodes) {
        std::cout << pair.first << ": " << pair.second << std::endl;
    }

    return 0;
}
```

O/P:

Enter a string: Saurav

Huffman Codes:

S: 100

a: 11

r: 101

u: 01

v: 00

3.Write a program to solve a 0-1 Knapsack problem using dynamic programming or branch and bound strategy.

```
#include <bits/stdc++.h>
using namespace std;


// Function to solve the 0/1 Knapsack problem using memoization

int knapsackUtil(vector<int>& wt, vector<int>& val, int ind, int W,
vector<vector<int>>& dp) {

 // Base case: If there are no items left or the knapsack has no capacity,
return 0

 if (ind == 0 || W == 0) {

 return 0;

 }
```

```
 // If the result for this state is already calculated, return it
if (dp[ind][W] != -1) {
 return dp[ind][W];
 }

 // Calculate the maximum value by either excluding the current item or including it
 int notTaken = knapsackUtil(wt, val, ind - 1, W, dp);
 int taken = 0;

 // Check if the current item can be included without exceeding the knapsack's capacity
 if (wt[ind] <= W) {
 taken = val[ind] + knapsackUtil(wt, val, ind - 1, W - wt[ind], dp);  }
 // Store the result in the DP table and return
 return dp[ind][W] = max(notTaken, taken);
}

// Function to solve the 0/1 Knapsack problem
int knapsack(vector<int>& wt, vector<int>& val, int n, int W)
{  vector<vector<int>> dp(n, vector<int>(W + 1, -1));  return
knapsackUtil(wt, val, n - 1, W, dp);
}
```

```
int main() {
 vector<int> wt = {1, 2, 4, 5};
 vector<int> val = {5, 4, 8, 6};
 int W = 5;
 int n = wt.size();

 cout << "The Maximum value of items the thief can steal is " <<
knapsack(wt, val, n, W);

 return 0;
}
```

O/p:

The Maximum value of items the thief can steal is 8

4.Design n-Queens matrix having first Queen placed. Use backtracking to place remainingQueens to generate the final n-queen

```
#include <iostream>
#include <vector>

using namespace std;

bool isSafe(vector<vector<int>>& board, int row, int col, int n) {  //
Check if no other Queen can attack this cell in the same column  for
```

```
(int i = 0; i < row; ++i) {
 if (board[i][col] == 1) {
 return false;
 }
 }

 // Check upper-left diagonal
 for (int i = row, j = col; i >= 0 && j >= 0; --i, --j) {
 if (board[i][j] == 1) {
 return false;
 }
 }

 // Check upper-right diagonal
 for (int i = row, j = col; i >= 0 && j < n; --i, ++j) {
 if (board[i][j] == 1) {
 return false;
 }
 }

 return true;
}
```

```
bool solveNQueens(vector<vector<int>>& board, int row, int n)
{  if (row == n) {
// All Queens have been successfully placed
return true;
}

for (int col = 0; col < n; ++col) {
if (isSafe(board, row, col, n)) {
board[row][col] = 1; // Place the Queen

// Recursively try to place the next Queen
if (solveNQueens(board, row + 1, n)) {
return true;
}

// If placing the Queen doesn't lead to a solution, backtrack
board[row][col] = 0;
}
}

// No valid placement in this row
return false;
```

```
}

int main() {
 int n;
 cin>>n;
 cout<<"Enter the row no and column no for placing the queen";
int r,c;
 cin>>r>>c;
 vector<vector<int>> board(n, vector<int>(n, 0));
board[r][c]=1;

 // int n = 4; // Change n to the desired board size

 // Place the first Queen at (0, 0)
 // board[0][0] = 1;

 if (solveNQueens(board, 1, n)) {
 // Print the final n-Queens configuration
 for (int i = 0; i < n; ++i) {
 for (int j = 0; j < n; ++j) {
 cout << board[i][j] << " ";
 }
 cout << endl;
```

```
 }
 } else {
 cout << "No solution exists." << endl;
 }

 return 0;
}
```

O/P:

4

Enter the row no and column no for placing the queen0

1 0 1 0 0

0 0 0 1

1 0 0 0

0 0 1 0

5.Write a program for analysis of quick sort by using deterministic and randomized variant.

```
#include <iostream>
#include <vector>
#include <ctime>
#include <cstdlib>
#include <chrono>
```

```
using namespace std;
using namespace chrono;

// Function to partition the array for Quick Sort
int partition(vector<int>& arr, int low, int high)
{  int pivot = arr[low];
 int i = low + 1;
 int j = high;

 while (true) {
 while (i <= j && arr[i] <= pivot)
i++;
 while (i <= j && arr[j] > pivot)
 j--;
 if (i <= j)
 swap(arr[i], arr[j]);
 else
 break;
 }
 swap(arr[low], arr[j]);
 return j;
}
```

```
// Deterministic Quick Sort
void deterministicQuickSort(vector<int>& arr, int low, int high)
{  if (low < high) {
 int pivot = partition(arr, low, high);
 deterministicQuickSort(arr, low, pivot - 1);
 deterministicQuickSort(arr, pivot + 1, high);
 }
}

// Randomized Quick Sort
int randomPartition(vector<int>& arr, int low, int high)
{  srand(time(0));
 int random = low + rand() % (high - low);
 swap(arr[random], arr[low]);
 return partition(arr, low, high);
}

void randomizedQuickSort(vector<int>& arr, int low, int high)
{  if (low < high) {
 int pivot = randomPartition(arr, low, high);
 randomizedQuickSort(arr, low, pivot - 1);
randomizedQuickSort(arr, pivot + 1, high);  }
```

```
}

int main() {
 int n;
 cout << "Enter the number of elements: ";
cin >> n;

 vector<int> arr(n);
 vector<int> arrCopy(n);

 cout << "Enter " << n << " integers:" << endl;
for (int i = 0; i < n; ++i) {
 cin >> arr[i];
 arrCopy[i] = arr[i];
 }

 high_resolution_clock::time_point start_time;
high_resolution_clock::time_point end_time;
duration<double> time_span;

 // Deterministic Quick Sort
 start_time = high_resolution_clock::now();
 deterministicQuickSort(arr, 0, n - 1);
 end_time = high_resolution_clock::now();
```

```
 time_span = duration_cast<duration<double>>(end_time - start_time);


 cout << "Deterministic Quick Sort took " << time_span.count() << "
seconds." << endl;


 // Randomized Quick Sort
 start_time = high_resolution_clock::now();
 randomizedQuickSort(arrCopy, 0, n - 1);
 end_time = high_resolution_clock::now();
 time_span = duration_cast<duration<double>>(end_time - start_time);


 cout << "Randomized Quick Sort took " << time_span.count() << "
seconds." << endl;


 return 0;
}
```

O/P:

```
Enter the number of elements: 4
Enter 4 integers:
2 3 5 7
Deterministic Quicksort took 4.75e-07 seconds.
Randomized Quick Sort took 3.663e-06 seconds.
```

— Mini Project

Implement merge sort and multithreaded merge sort. Compare time requiredby both the algorithms. Also analyze the performance of each algorithm for the best case and the worst case.

Code:

```
#include <iostream>

#include <vector>

#include <thread>


void merge(std::vector<int>& arr, int left, int mid, int right) {

 int n1 = mid - left + 1;

 int n2 = right - mid;


 std::vector<int> left_arr(n1);

 std::vector<int> right_arr(n2);


 for (int i = 0; i < n1; i++) {

 left_arr[i] = arr[left + i];

 }


 for (int i = 0; i < n2; i++) {

 right_arr[i] = arr[mid + 1 + i];

 }


 int i = 0, j = 0, k = left;

 while (i < n1 && j < n2) {
 if (left_arr[i] <= right_arr[j]) {
```

```
arr[k] = left_arr[i];
i++;
} else {
arr[k] = right_arr[j];
j++;
}
k++;
}

while (i < n1) {
arr[k] = left_arr[i];
i++;
k++;
}

while (j < n2) {
arr[k] = right_arr[j];
j++;
k++;
}
}

void merge_sort(std::vector<int>& arr, int left, int right) {
if (left < right) {
int mid = left + (right - left) / 2;

merge_sort(arr, left, mid);
```

```
 merge_sort(arr, mid + 1, right);

 merge(arr, left, mid, right);
 }
}

void multithreaded_merge_sort(std::vector<int>& arr, int left, int right, int depth)
{  if (left < right) {
 if (depth == 0) {
 merge_sort(arr, left, right);
 } else {
 int mid = left + (right - left) / 2;

 std::thread left_thread(multithreaded_merge_sort, std::ref(arr), left, mid, depth - 1);  std::thread
right_thread(multithreaded_merge_sort, std::ref(arr), mid + 1, right, depth - 1);

 left_thread.join();
 right_thread.join();

 merge(arr, left, mid, right);
 }
 }
}

int main() {
 std::vector<int> arr = {12, 11, 13, 5, 6, 7};
```

```
// Merge Sort
merge_sort(arr, 0, arr.size() - 1);


std::cout << "Sorted array using Merge Sort: ";

for (int num : arr) {

std::cout << num << " ";

}

std::cout << std::endl;


// Multithreaded Merge Sort

std::vector<int> arr2 = {12, 11, 13, 5, 6, 7};

int num_threads = 2; // Adjust the number of threads as needed


multithreaded_merge_sort(arr2, 0, arr2.size() - 1, num_threads);


std::cout << "Sorted array using Multithreaded Merge Sort: ";

for (int num : arr2) {

std::cout << num << " ";

}

std::cout << std::endl;


return 0;

}
```

Regular Merge Sort:

Best Case: The best-case time complexity of regular merge sort is O(n log n), where "n" is the number of elements in the list. This is because the list is consistently divided into

two halves until it reaches the base case (single-element sublists), and merging is performed efficiently.

Worst Case: The worst-case time complexity of regular merge sort is also O(n log n). This occurs when the list is divided into two halves in a balanced manner at each level of recursion, resulting in log n levels of recursion, and merging is still performed efficiently.

Performance Analysis: Regular merge sort is highly consistent in its performance, and its time complexity is optimal for general sorting tasks. It doesn't benefit significantly from parallelization, as the recursive nature of the algorithm makes it difficult to parallelize efficiently without incurring thread management overhead.

Multithreaded Merge Sort:

Best Case: The best-case time complexity of multithreaded merge sort remains O(n log n) for the same reasons as the regular merge sort. In the best case, multithreading may provide some speedup, but it may not be substantial for smaller inputs or when the number of available CPU cores is limited.

Worst Case: The worst-case time complexity of multithreaded merge sort is also O(n log n), but the constant factors may be higher than in the regular merge sort due to the overhead of thread creation and synchronization. In some cases, the performance might even degrade compared to the regular merge sort, especially for small input sizes or when the number of threads used is excessive.

Performance Analysis: The performance of multithreaded merge sort depends heavily on factors such as the size of the input, the number of CPU cores available, and the efficiency of thread management. In general, it is most beneficial for large input sizes and when there are many available CPU cores. However, the benefits may not be noticeable for small lists or when the overhead of managing threads is significant.

In summary, both regular merge sort and multithreaded merge sort have the same theoretical time complexity of O(n log n) for both best and worst cases. Regular merge sort is more predictable and doesn't rely on parallelism, making it a reliable choice for general sorting tasks. Multithreaded merge sort can provide performance improvements but introduces complexities related to thread management, and its benefits are more pronounced for large input sizes and multicore processors. Careful benchmarking and analysis are needed to determine the actual performance gains for a specific use case.
