Logo

Description automatically generated

**MAHARAJA AGRASEN INSTITUTE OF TECHNOLOGY**

**VISION**

To nurture young minds in a learning environment of high academic value and imbibe spiritual and ethical values with technological and management competence.

**MISSION**

The Institute shall endeavor to incorporate the following basic missions in the teaching methodology:

**Engineering Hardware – Software Symbiosis**

Practical exercises in all Engineering and Management disciplines shall be carried out by Hardware equipment as well as the related software enabling deeper understanding of basic concepts and encouraging inquisitive nature.

**Life – Long Learning**

The Institute strives to match technological advancements and encourage students to keep updating their knowledge for enhancing their skills and inculcating their habit of continuous learning.

**Liberalization and Globalization**

The Institute endeavors to enhance technical and management skills of students so that they are intellectually capable and competent professionals with Industrial Aptitude to face the challenges of globalization.

**Diversification**

The Engineering, Technology and Management disciplines have diverse fields of studies with different attributes. The aim is to create a synergy of the above attributes by encouraging analytical thinking.

**Digitization of Learning Processes**

The Institute provides seamless opportunities for innovative learning in all Engineering and Management disciplines through digitization of learning processes using analysis, synthesis, simulation, graphics, tutorials and related tools to create a platform for multi-disciplinary approach.

**Entrepreneurship**

The Institute strives to develop potential Engineers and Managers by enhancing their skills and research capabilities so that they become successful entrepreneurs and responsible citizens.

Logo

Description automatically generated

**MAHARAJA AGRASEN INSTITUTE OF TECHNOLOGY**

**COMPUTER SCIENCE & ENGINEERING DEPARTMENT**

**VISION**

“To be centre of excellence in education, research and technology transfer in the field of computer engineering and promote entrepreneurship and ethical values.”

**MISSION**

“To foster an open, multidisciplinary and highly collaborative research environment to produce world-class engineers capable of providing innovative solutions to real life problems and fulfil societal needs.”

**PRACTICAL RECORD**

**PAPER CODE : CIC-359**

Name of the student : Lakshay Sharma

University Roll No. : 02396402721

Branch : Computer Science Engineering (CSE)

Section/Group : 5C12

**PRACTICAL DETAILS**

Experiments according to COA lab syllabus prescribed by GGSIPU

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Exp. no** | **Experiment Name** | **Date of performance** | **Date of checking** | **R1 (3)** | **R2 (3)** | **R3 (3)** | **R4 (3)** | **R5 (3)** | **Total Marks (15)** | **Signature** |
|  |  |  |  |  |  |  |  |  |  |  |
| **Exp. no** | **Experiment Name** | **Date of performance** | **Date of checking** | **R1 (3)** | **R2 (3)** | **R3 (3)** | **R4 (3)** | **R5 (3)** | **Total Marks (15)** | **Signature** |
|  |  |  |  |  |  |  |  |  |  |  |

**EXPERIMENT – 1**

**AIM:** Implementation of different sorting algorithms and analysis of their time complexities.

**SOURCE CODE:**

#include <iostream>

#include <vector>

#include <chrono>

#include <functional>

using namespace std;

using namespace std::chrono;

void printArr(string name, vector<int> arr) {

cout << "\n------------------------------------\n";

cout << name << "\n";

for (auto& i : arr) cout << i << " ";

}

void printExecutionTime(function<vector<int>()> sortingFunction, string functionName) {

auto start\_time = high\_resolution\_clock::now();

vector<int> result = sortingFunction();

auto end\_time = high\_resolution\_clock::now();

auto duration = duration\_cast<microseconds>(end\_time - start\_time);

printArr(functionName, result);

cout << "\nExecution time: " << duration.count() << " microseconds";

}

vector<int> selectionSort(vector<int> arr) {

for (int i = 0; i < arr.size() - 1; i++) {

int min\_index = i;

for (int j = i + 1; j < arr.size(); j++)

if (arr[min\_index] > arr[j]) min\_index = j;

swap(arr[min\_index],arr[i]);

}

return arr;

}

vector<int> bubbleSort(vector<int> arr) {

for (int i = 0; i < arr.size() - 1; i++)

for (int j = 0; j < arr.size() - i - 1; j++)

if (arr[j] > arr[j + 1]) swap(arr[j], arr[j + 1]);

return arr;

}

vector<int> merge(vector<int> left, vector<int> right) {

vector<int> result;

for (int i = 0, j = 0; i < left.size() || j < right.size();) {

if (i < left.size() && (j >= right.size() || left[i] <= right[j]))

result.push\_back(left[i++]);

else

result.push\_back(right[j++]);

}

return result;

}

vector<int> mergeSort(vector<int> arr) {

if (arr.size() == 1) return arr;

int mid = arr.size() / 2;

vector<int> left(arr.begin(), arr.begin() + mid);

vector<int> right(arr.begin() + mid, arr.end());

left = mergeSort(left);

right = mergeSort(right);

return merge(left, right);

}

int main() {

int size = 20;

vector<int> arr(size);

for (int i = 0; i < size; i++) arr[i] = rand() % (size + 1);

printArr("Original Array",arr);

printExecutionTime([&]() { return bubbleSort(arr); }, "Bubble Sort");

printExecutionTime([&]() { return selectionSort(arr); }, "Selection Sort");

printExecutionTime([&]() { return mergeSort(arr); }, "Merge Sort");

return 0;

}

**OUTPUT:**

------------------------------------

Original Array

1 4 9 19 8 10 10 9 15 10 2 19 20 4 20 7 3 15 16 16

------------------------------------

Bubble Sort

1 2 3 4 4 7 8 9 9 10 10 10 15 15 16 16 19 19 20 20

Execution time: 6 microseconds

------------------------------------

Selection Sort

1 2 3 4 4 7 8 9 9 10 10 10 15 15 16 16 19 19 20 20

Execution time: 4 microseconds

------------------------------------

Merge Sort

1 2 3 4 4 7 8 9 9 10 10 10 15 15 16 16 19 19 20 20

Execution time: 68 microseconds

**VIVA QUESTIONS:**

Q1. **What is a sorting algorithm, and why is it essential in computer science?**

1. A sorting algorithm is a method for arranging elements in a specific order, such as ascending or descending. Sorting is essential in computer science because it enables efficient searching, data retrieval, and facilitates various operations on data by maintaining a structured order, ultimately improving the performance of many algorithms and applications.

Q2. **Compare the time complexities of Merge Sort and Quick Sort. Which one is generally faster in practice and why?**

1. Both Merge Sort and Quick Sort have an average time complexity of O(n log n). Quick Sort is often faster in practice due to its smaller constant factors, making it more cache-efficient. However, Merge Sort is a stable and predictable choice for sorting, whereas Quick Sort's performance may vary with different data distributions.

Q3. **How can parallel computing and distributed systems be leveraged to improve sorting efficiency?**

1. Parallel computing uses multiple processors to sort different parts of data simultaneously, reducing sorting time. In distributed systems, data is split across multiple machines, each sorting its portion, which can be advantageous for huge datasets. Parallelism and distribution enable efficient sorting by utilizing resources in a more scalable and faster manner.

Q4. **How are indexing and sorting related in database systems, and why is sorting important for query optimization?**

1. Indexing and sorting are related in database systems because indexing structures like B-trees maintain data in a sorted order, improving data retrieval efficiency. Sorting is important for query optimization as it allows the database engine to quickly find and access relevant data, reducing query execution time and enhancing overall system performance.

**EXPERIMENT – 2**

**AIM:** Implementation of different searching algorithms and analysis of their time complexities.

**SOURCE CODE:**

#include <iostream>

#include <vector>

#include <chrono>

using namespace std;

using namespace std::chrono;

int main() {

// non-decreasing array due to binary search

vector<int> arr = {1,4,5,6,6,7,7,7,7,8,10,12,16,18,19,20,32,36,100};

cout << "Given Array\n";

for (auto& i : arr) cout << i << " ";

int elem,index = 0, strt = 0, end = arr.size(),bin\_cache = arr.size();

cout << "\nElement to search:"; cin >> elem;

cout << "---------------------\n";

cout << "Linear Search\n";

auto start\_time = high\_resolution\_clock::now();

while (index < arr.size())

if (arr[index++] == elem) {

cout << "Element at index " << index-1;

break;

}

if (index == arr.size()) cout << "Element not found";

auto duration = duration\_cast<microseconds>(high\_resolution\_clock::now() - start\_time);

cout << "\nExecution time: " << duration.count() << " microseconds\n";

cout << "---------------------\n";

cout << "Binary Search\n";

bool found = false;

start\_time = high\_resolution\_clock::now();

while (strt <= end) {

index = strt + (end - strt)/2;

if (bin\_cache == index) break;

else bin\_cache = index;

if (arr[index] == elem) {found = true; break;}

else if (arr[index] < elem) strt = index+1;

else end = index-1;

}

if (found) cout << "Element at index " << index;

else cout << "Element not found";

duration = duration\_cast<microseconds>(high\_resolution\_clock::now() - start\_time);

cout << "\nExecution time: " << duration.count() << " microseconds\n";

return 0;

}

**OUTPUT:**

Given Array

1 4 5 6 6 7 7 7 7 8 10 12 16 18 19 20 32 36 100

Element to search:16

---------------------

Linear Search

Element at index 12

Execution time: 2 microseconds

---------------------

Binary Search

Element at index 12

Execution time: 0 microseconds

**VIVA QUESTIONS:**

Q1. **Why is binary search considered more efficient than linear search for large datasets?**

1. Binary search is more efficient for large datasets because it divides the dataset in half at each step, significantly reducing the number of comparisons needed. This logarithmic time complexity (O(log n) as opposed to O(n) for linear search) makes binary search faster and more suitable for handling vast amounts of data.

Q2. **Can linear search be used for non-numeric data, such as strings? If so, how?**

1. Yes, linear search can be used for non-numeric data like strings. To perform a linear search on strings, compare the target string with each element in the list sequentially until a match is found.

Q3. **Can binary search be used for non-numeric data, such as strings? If so, how?**

1. Yes, binary search can be used for non-numeric data like strings if the data is sorted in lexicographical order. Compare the target string with the middle element, and if it's smaller, search in the lower half; if larger, search in the upper half.

Q4. **Provide real-world examples or scenarios where linear search is commonly used?**

1. 1. Searching for a specific name in an unsorted contact list.

2. Scanning through an unsorted list of files to find a particular document.

3. Checking a shopping list for the presence of an item.

Q5. **Provide real-world examples or scenarios where binary search is commonly used?**

1. 1. Searching for a word in a sorted dictionary.

2. Locating a book in a library using the Dewey Decimal System.

3. Finding a name in a phonebook.

**EXPERIMENT – 3**

**AIM:** Implementation of Huffman Tree (variable size)

**SOURCE CODE:**

#include <bits/stdc++.h>

using namespace std;

using namespace std::chrono;

class node {

public:

char ch;

int data;

node \*left, \*right;

node(int data, char ch) {

this->ch = ch; this->data = data; left = right = NULL;

}

};

class cmp {

public:

bool operator()(node\* a, node\* b) {return a->data > b->data;}

};

node\* buildHuffmanTree(string S, vector<int>& freq, int n) {

priority\_queue<node\*, vector<node\*>, cmp> pq;

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

node\* temp = new node(freq[i], S[i]);

pq.push(temp);

}

while(pq.size() > 1) {

node\* left = pq.top();

pq.pop();

node\* right = pq.top();

pq.pop();

node\* newNode = new node(left->data + right->data, S[0]);

newNode->left = left;

newNode->right = right;

pq.push(newNode);

}

node\* root = pq.top();

return root;

}

void traverse(node\* root, string temp, vector<string>& ans, vector<char>& l) {

if(!root->left && !root->right) {

l.push\_back(root->ch);

ans.push\_back(temp);

return;

}

traverse(root->left, temp + "0", ans, l);

traverse(root->right, temp + "1", ans, l);

}

int main() {

string s;

cout<<"Enter the message with distinct letters:\n"; cin>>s;

cout<<"Enter Corresponding Frequencies:\n";

int n = s.size();

vector<int> freq(n);

for(int i = 0; i < n; i++) cin>>freq[i];

high\_resolution\_clock::time\_point start = high\_resolution\_clock::now();

node\* root = buildHuffmanTree(s, freq, n);

vector<string> encodings;

vector<char> leaves;

traverse(root, "", encodings, leaves);

cout<<"The Huffman Encodings are:\n";

for(int i = 0; i < n; i++) cout<<leaves[i]<<":"<<" "<<encodings[i]<<endl;

high\_resolution\_clock::time\_point stop = high\_resolution\_clock::now();

auto duration = duration\_cast<microseconds>(stop - start);

cout << "Time taken: " << duration.count() << " microseconds" << endl;

return 0;

}

**OUTPUT:**

Enter the message with distinct letters:

thisa

Enter Corresponding Frequencies:

1 1 1 1 1 2 2 1

The Huffman Encodings are:

a: 00

i: 01

s: 10

t: 110

h: 111

Time taken: 80 microseconds

**VIVA QUESTIONS:**

Q1.  **What is the difference between static and dynamic Huffman coding?**

1. Static Huffman coding uses a fixed code table, predetermined before compression. Dynamic Huffman coding adapts the code during compression based on changing symbol probabilities, offering flexibility for variable sources but requiring the transmission of the code table.

Q2. **Can you describe the process of building a Huffman tree for a given set of symbols and their frequencies?**

1. Frequency Count:

- Calculate the frequency of each symbol in the given set.

1. Create Nodes:

- Create a leaf node for each symbol, associating it with its frequency.

3. Build Priority Queue:

- Place all leaf nodes into a priority queue or min-heap based on their frequencies.

4. Build Huffman Tree:

- While there is more than one node in the priority queue:

- Extract two nodes with the lowest frequencies.

- Create a new internal node with a frequency equal to the sum of the two extracted nodes.

- Insert the new node back into the priority queue.

1. Repeat until a Single Node is Left:

- Continue this process until only one node remains in the priority queue. This node becomes the root of the Huffman tree.

6. Assign Codes:

- Traverse the Huffman tree to assign binary codes to each symbol. Assign '0' for left branches and '1' for right branches.

The resulting Huffman tree is a binary tree where each leaf represents a symbol and each internal node has a frequency equal to the sum of its children. The binary codes assigned to symbols are determined by the path from the root to each leaf in the tree. The codes for each symbol are unique, and the tree is used for both compression and decompression.

**EXPERIMENT – 4**

**AIM:** Solving the N-Queens problem using the backtracking approach

**SOURCE CODE:**

#include <iostream>

#include <vector>

using namespace std;

bool isSafe(int row,int col,vector<string>& board,int& n) {

// As we are filling from left to right.

// So there is no need to check at right.

// Same row

int x = row, y = col;

while (y >= 0) {

if (board[x][y] == 'Q') return false;

y--;

}

// As we are placing 1 queen per column So no need to check it.

x = row; y = col;

// Left Upper diagonal

while (x >= 0 && y >= 0) {

if (board[x][y] == 'Q') return false;

x--; y--;

}

x = row; y = col;

// Left Lower diagonal

while (x < n && y >= 0) {

if (board[x][y] == 'Q') return false;

x++; y--;

}

return true;

}

void solve(int col,vector<string>& board,vector<string>& ans,int& n) {

// Base case

if (col == n) {ans = board; return ;};

// Recursion case.

for (int row = 0; row < n; row++) {

if (isSafe(row,col,board,n)) {

// Place the queen

board[row][col] = 'Q';

// Move to next row

solve(col+1,board,ans,n);

// Remove the queen

board[row][col] = '.';

}

}

}

vector<string> solveNQueens(int n) {

vector<string> ans;

// Initializing the board

vector<string> board(n,string(n,'.'));

solve(0,board,ans,n);

return ans;

}

int main() {

int n = 0;

cout << "Number of Queens:";

cin >> n;

vector<string> res = solveNQueens(n);

for (auto& i: res) cout << i << "\n";

return 0;

}

**OUTPUT:**

Number of Queens:6

..Q...

.....Q

.Q....

....Q.

Q.....

...Q..

**VIVA QUESTIONS:**

Q1.  **Describe the concept of backtracking and its role in solving the N-Queens problem?**

**Backtracking:**

Backtracking is a systematic algorithmic technique for finding all (or some) solutions to computational problems, particularly constraint satisfaction problems. It incrementally builds candidates for solutions and abandons a candidate ("backtracks") as soon as it determines that the candidate cannot possibly be completed to a valid solution.

**Role in N-Queens Problem:**

In the N-Queens problem, the goal is to place N chess queens on an N×N chessboard in such a way that no two queens threaten each other. Backtracking is instrumental in solving this problem:

1. Placement Attempt:

- Start placing queens on the chessboard one by one in a row-wise manner.

1. Check Validity:

- After placing a queen, check if it violates any constraints (other queens in the same row, column, or diagonals).

3. Explore:

- If the current placement is valid, move on to the next row and repeat the process.

4. Backtrack:

- If there is no valid placement for a queen in the current row, backtrack to the previous row and explore other possibilities.

5. Repeat:

- Continue this process until all queens are placed without violating constraints or until all possibilities are explored.

6. Solution Found:

- If a placement is found that satisfies all constraints, it represents a solution to the N-Queens problem.

Backtracking efficiently explores the solution space, discarding paths that lead to invalid solutions. It's a depth-first search approach, and its recursive nature allows the algorithm to systematically try different combinations until a valid solution is found or all possibilities are exhausted. The N-Queens problem demonstrates the effectiveness of backtracking in solving constraint satisfaction problems.

Q2. **What is the time complexity of the N-Queens problem when solved using backtracking?**

A. Worst Case: O(N!) exponential time complexity.

Best Case: No fixed best case; it heavily depends on the specific heuristics and optimizations applied.

Average Case: Generally exponential, influenced by the specific characteristics of the algorithm and the problem instance.