

Advanced algorithms

School of computing

By: ali mostafa Mahmoud
202101194

Abstract

The primary goals and intended learning outcomes of this module (Advanced Algorithms) are covered in detail in my coursework portfolio, along with my solutions to the eight exercises in the CW (problem set) that were required to complete. It also includes information on the specifics of each problem, the libraries or APIs I used to implement the solution, the testing I did (test cases) to make sure the program ran and executed properly, an analysis of the algorithm's effectiveness and justification for its [time and space] complexity, and a detailed explanation of every line or block of code written (relating to each solution). Furthermore, other algorithms, some utilizing various data structures, will be given, along with an explanation of their complexity and the rationale behind my selection of a particular methodology.

introduction

Advanced algorithms, sometimes known as data structures and algorithms (DSA), are the study of fundamental data structures and algorithms that are components of programming paradigms. The term "abstract data types" refers to the categories of data structures that are designed and created to store data, whereas the "algorithms" section deals primarily with mathematical formulas and approaches that are implemented in the form of code to solve problems that involve sorting, searching, manipulating, etc. Stacks, queues, linked lists, hash tables, sorting and searching, trees, graphs, and more make up the advanced algorithms. In this report, these ideas will be covered in further detail. In-depth discussions of these ideas will be included throughout the study. In the end, we will employ and discuss a wide range of algorithms, which include: sorting arrays, in-place heap sort, bubble sort, selection sort, insertion sort, spell checker, matrices, and the Dijkstra algorithm.

Task 1

Overview of the task

The goal of this task is to check if the two unordered arrays have the same sets of numbers

Code explanation

Check equality of arrays

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

using namespace std;

bool checkEqualityOfArrays(const vector<int>& arr1, const vector<int>& arr2) {
    if (arr1.size() != arr2.size()) {
        return false;
    }

    unordered_set<int> set1(arr1.begin(), arr1.end());
    unordered_set<int> set2(arr2.begin(), arr2.end());

    return set1 == set2;
}
```

First, import the needed libraries for the task. Unordered sets are individual components that can be stored in any order. The elements of the input arrays are saved in this code using unordered sets. Vectors are changeable in size; vectors are dynamic arrays. Vectors are passed to the function for comparison in this case as parameters. The function `checkEqualityOfArrays` is used to check if the two arrays have an identical collection of elements in them. The uniqueness of the elements in the arrays is compared by the algorithm using unordered sets. **Main function**

```
vector<int> array1 = {2, 4, 1, 3};
vector<int> array2 = {3, 1, 4, 2};

if (checkEqualityOfArrays(array1, array2)) {
    cout << "Both arrays have the same set of numbers." << std::endl;
} else {
    cout << "Arrays have different sets of numbers." << std::endl;
}

return 0;
```

```
}
```

Two defined arrays are supplied to the `checkEqualityOfArrays` function for comparison in the main function. then a message is printed indicating whether or not the arrays have the same set of integers, depending on the outcome.

Summary

In summary, the code uses unordered sets to compare the uniqueness of elements to efficiently determine if two arrays are equal. It offers a simple and efficient method for figuring out whether two arrays have the same set of numbers in them. When the uniqueness of the elements is the only factor that matters and the order of the components in the arrays is irrelevant, this method can be helpful.

Libaries and api used:

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

Task 2

Overview of the task

The goal is to create a C++ program for the in-place heap-sort algorithm. Compare its execution time to that of the conventional heap sort. The binary heap data structure is the foundation of the comparison-based sorting method known as heap sort. It is comparable to the selection sort, in which the maximum element is located first and positioned at the end. For the last piece, we use the same procedure.

Code explanation

Heapify for place heap sort.

```
void heapify(vector<int>& arr, int n, int i) {
    int largest = i;
    int left = 2 * i + 1;
```

```

int right = 2 * i + 2;

if (left < n && arr[left] > arr[largest])
    largest = left;

if (right < n && arr[right] > arr[largest])
    largest = right;

if (largest != i) {
    swap(arr[i], arr[largest]);
    heapify(arr, n, largest);
}
}

```

First, import the needed libraries for the task. The heapfiy function is used for rooting a subtree at index I of size n in the array. The heapSort function sorts the array by continually extracting the maximum element from the heap after first creating a max heap from the input array.

place heap sort.

```

void inPlaceHeapSort(vector<int>& arr) {
    int n = arr.size();

    for (int i = n / 2 - 1; i >= 0; i--)
        heapify(arr, n, i);

    for (int i = n - 1; i > 0; i--) {
        swap(arr[0], arr[i]);
        heapify(arr, i, 0);
    }
}

```

The heapSort function sorts the array by continually extracting the maximum element from the heap after first creating a max heap from the input array.

Comparison between heap sort and in place heap sort

```

int main() {
    vector<int> arr = {12, 11, 13, 5, 6, 7};
    int arrr[]={12, 11, 13, 5, 6, 7};
}

```

```

int n = sizeof(arr) / sizeof(arr[0]);
inPlaceHeapSort(arr);

auto start = chrono::high_resolution_clock::now();
inPlaceHeapSort(arr);
auto end = chrono::high_resolution_clock::now();
chrono::duration<double> elapsed_seconds = end - start;
cout << "Time taken by in-place heap-sort: " << elapsed_seconds.count() << "s\n";

start = chrono::high_resolution_clock::now();
heapSort(arr, n);
end = chrono::high_resolution_clock::now();
elapsed_seconds = end - start;
cout << "Time taken by standard heap-sort: " << elapsed_seconds.count() << "s\n";

return 0;
}

```

The chrono library is used for comparing the running times of both algorithms. Chrono:high_resolution_clock is a clock that has the smallest tick period available on the system. There is a start and end, which consists of a Chrono:high_resolution_clock that is responsible for calculating the starting time and ending time. After the starting time begins, run the sorting algorithm, and then the end time. Elapsed seconds by subtracting the end from the start. the output of the code shows that in-place heap sort took more time than the standard heap sort

Summary

In summary, the binary heap data structure is used by the effective sorting algorithm known as heap sort. It works well for sorting big datasets and has an $O(n \log n)$ time complexity. The task implemented heap sort in C++ code using both the in-place and standard heap sort algorithms. Libraries and api used :

```

#include <iostream>
#include <vector>
#include <algorithm>
#include <chrono>

```

Task 3

Overview of the task

The goal is to write four different sort algorithms: heap sort, bubble sort, selection sort, and insertion sort. for the four methods to return the total number of comparisons made to the sorted array (arr). The minimal heap sorting method serves as the foundation for the heap_sort implementation. Create a function named test_comparisons() that uses 30 random arrays to execute the preceding 4 functions. Thirty arrays were sorted, thirty arrays arranged inversely, and thirty random arrays. Code has to determine how many comparisons are made and how long it takes to complete each n in seconds. Then On the console, print the findings as tables. Store the outcome in an Excel spreadsheet, with separate sheets for the number of comparisons and execution time. then Plot the outcomes as a chart (two independent charts) in Excel.

Code explnation

```
#include <iostream>

#include <fstream>

#include <chrono> #include <iomanip>

using namespace std;
```

first, import the needed libraries for the code.

```

int bubblicount(int ar[], int n)
{
    int numcomp = 0;
    bool swapcurs;

    for (int i = 0; i < n - 1; i++)
    {
        swapcurs = false;        for (int
x = 0; x < n - i - 1; x++)
        {
            if (ar[x] >
ar[x+1])
            {
                swap(ar[x], ar[x+1]);
            }
            swapcurs = true;
        }
        numcomp++;
    }

    if (!swapcurs)
    {
        break;
    }

    return numcomp;
}

```

One of the simplest sorting methods, bubble sort, involves frequently switching nearby components that are out of order. Due to its relatively high average and worst-case time complexity, this approach is not appropriate for huge data sets. the Bubble Sort algorithm, Move left, compare the items that are next to it, and insert the higher element on the right. The largest

element is initially shifted to the rightmost end in this manner. Once the data is sorted, this procedure is repeated to identify the second largest place, and so on.

```
int selectioncount(int ar[], int n)
{
    int numcomp = 0;
    int min;

    for (int i = 0; i < n - 1; i++)
    {
        min = i;
        for (int j
= i + 1; j < n; j++)
        {
            if (ar[j] <
ar[min])
            {
                min = j;
            }
            numcomp++;
        }
        swap(ar[i], ar[min]);
    }
    return numcomp;
}
```

Selection sort is a straightforward and effective sorting algorithm that functions by repeatedly choosing the largest (or smallest) element from the list's unsorted section and shifting it to the sorted section. The algorithm alternates the initial member of the unsorted section of the list with the smallest (or largest) element it repeatedly selects from the unsorted segment. To sort the complete list, this step is repeated for the remaining unsorted portion.

```
int heap(int arr[], int n, int i)
```

```
{
```

```
    int numcomp = 0;
```

```
    int minvI = i;    int l
```

```
    = 2 * i + 1;    int r =
```

```
    2 * i + 2;    if (l < n)
```

```
    {
```

```
        numcomp++;
```

```
        if (arr[l] < arr[minvI])
```

```
        {
```

```
minvI = l;
```

```
        }    }
```

```
if (r < n)
```

```
{
```

```

    numcomp++;

    if (arr[r] < arr[minvl])
    {
minvl = r;
    } } if
(minvl != i)
    {
        swap(arr[i], arr[minvl]);
numcomp += heap(arr, n, minvl);
    }

    return numcomp;
}

```

```

int heapsocount(int arr[], int n)
{
    int numcomp = 0;    for (int x =
(n - 2) / 2; x >= 0; x--)
    {
        numcomp += heap(arr, n, x);
    }    for (int x = n - 1; x > 0;
x--)
    {
        swap(arr[0], arr[x]);
numcomp += heap(arr, x, 0);
    }

    return numcomp;
}

```

Heap sort is a sorting method based on comparison and the binary heap data structure. It is comparable to the selection sort, in which the minimum element is located first and positioned first. For the remaining components, follow the same procedure. After utilizing the heapify(heap) function to turn the array into a heap data structure, remove each Max-heap root node one at a time and replace it with the heap's last node before heapifying the heap's root. Until the heap's size is more than 1, keep doing this. Create a heap using the supplied input array. Until the heap has just one element, keep doing the following steps: Place the final member in the heap in place of the largest element, the root element. Eliminate the heap's final part, which is now in the proper place. Add to the heap the remaining components. The elements in the input array are rearranged to produce the sorted array.

```
int insertsocount(int ar[], int n)
{
    int s = 0;
    int numcomp = 0;

    for (int i = 1; i < n;
i++)
    {
        s = ar[i];
        for (int
j = i - 1; j > -1; j--)
        {
            numcomp++;

            if (s < ar[j])
            {
swap(ar[j], ar[j+1]);

            }
        }
        else
        {
            break;
        }
    }

    return numcomp;
}
```

A straightforward sorting method called insertion sort creates a sorted array one element at a time. Because it doesn't need any more memory space than the initial array, it is referred to as an "in-place" sorting method. Since the first member in the array is presumed to be sorted, we must begin with the second element. If the second element is smaller than the first after comparing the two, then switch them. To place the third element among the first three in the proper order, move it to the second element, compare it with that element, and then move back to the first element. Proceed with the sorting process, comparing each element with the ones that came before it and switching as necessary to arrange it correctly in the hierarchy of sorted elements. Continue sorting until the full array is done.

```
void reset(int te[], int ar[], int z )
{
    for (int x = 0; x < z;
x++)
    {
        te[x] = ar[x];
    }
}
```

The function reet uses three parameters: two arrays and one integer. The reset function is used for reshuffling the sorted array using a for loop that stores all the numbers of ar array in te array.

```

void exsorts(int te[], int ar[], int x, string artype, string soalg)
{
    reset(te, ar, x);    auto start =
chrono::high_resolution_clock::now();    int st;

    if (soalg == "Heap")
    {
        st = heapsocount(te, x);
    }

    else if (soalg == "Bubble")
    {
        st = bubblcount(te, x);
    }

    else if (soalg == "Selection")
    {
        st = selectioncount(te, x);
    }

    else
    {
        st = insertsocount(te, x);
    }

    auto end = chrono::high_resolution_clock::now();    double exec_time =
chrono::duration_cast<chrono::nanoseconds>(end - start).count() * 1e-9;    dataFile << x << ", "
<< artype << ", " << soalg << ", " << st << ", " << exec_time << "\n";    pTable(x, artype, soalg, st,
exec_time);
}

```

The exsorts function has five parameters. exsorts first, runs the reshuffle function. Second, the if statements check if soalg is equal to the needed sort algorithm. If so it will run the sort algorithm needed. Third, print table function takes five parameters and it is used for printing the table with the required things.

```

void testcompar()
{
    srand(time(0));

    dataFile << "Size of Array,Type of Array,Sorting Algorithm,Number of Comparisons,Execution Time\n";

    cout << left << setw(20) << "Size of Array" << left << setw(20) << "Type of Array"

        << left << setw(20) << "Sorting Algorithm" << left << setw(25) << "Number of Comparisons"

<< left << setw(20) << "Execution Time" << "\n";

    for (int m = 1;m < 31; m++)
    {
        int* randarr = new int[m];

int* sortedarr = new int[m];

int* invarr = new int[m];    int

number1 = 0;    int number2

= 0;    for (int x = 0; x < m;

x++)

    {

        randarr[x] = rand() % (m * 20);

number1 += rand() % (m * 20);

sortedarr[x] = number1;

number2 += rand() % (m * 20);

invarr[m-x-1] = number2;

    }

    rsorts(randarr, "Random", m);

rsorts(sortedarr, "Sorted", m);

```

```
rsorts(invarr, "Inversely-sorted", m);  
  
delete[] randarr;    delete[] sortedarr;  
  
delete[] invarr;  
  
}  
}
```

The testcompar function is used for creating thirty random, sorted and inversely sorted array. then used the run_sorts function 3 times for every type of every to sort the arrays. after that, the arrays get deleted to create new arrays, and so on.

Task 4

Overview of the task

The goal of this task is to write a recursive program that is used for reordering the elements in A such that any element larger than k comes after all elements less than or equal to k. In an array of n values, how complex is your program? Contrast the iterative algorithm's complexity with that of the recursive algorithm.

Code explanation

```
#include <iostream>
#include <vector>
using namespace std;

void rearrangeAr(vector<int>& arr, int k, int start, int end) {
    if (start >= end) return;

    if (arr[start] > k && arr[end] <= k) {
        swap(arr[start], arr[end]);
        rearrangeAr(arr, k, start + 1, end - 1);
    } else if (arr[start] <= k) {
        rearrangeAr(arr, k, start + 1, end);
    } else if (arr[end] > k) {
        rearrangeAr(arr, k, start, end - 1);
    }
}
```

First, import the needed libraries for the task. The function's parameters are the vector arr, the value k, and the vector's start and end indices. Because the array has already been restructured, the function returns if the start index is greater than or equal to the end index. It changes the elements at the start and end indices and calls itself recursively with the new end index if the element at the start index is bigger than k. The function calls itself recursively with the updated start index if the element at the start index is less than or equal to k.

Main function

```

int main() {
    vector<int> arr = {4, 7, 2, 9, 5, 1, 8};
    int k = 5;

    rearrangeAr(arr, k, 0, arr.size() - 1);

    for (int num : arr) {
        cout << num << " ";
    }

    return 0;
}

```

The main function creates a new vector and then runs the rearranger function with the needed parameters. The for loop that is used for printing the array

Time complexity

This recursive algorithm has an $O(n)$ time complexity, where n is the array's element count. That's because every element is only checked once. In contrast, as the iterative method would only need to make one pass through the array to rearrange the items, it would likewise have an $O(n)$ time complexity.

Summary

In summary, the above C++ code effectively rearranges an array by placing elements less than or equal to k at the end and elements larger than a given pivot value, k , at the beginning. Recognizing the interplay between array manipulation and recursion in this code can be beneficial in similar array rearrangement situations.

Libraries and api used:

```

#include <iostream>
#include <vector>

```

Task 5

Task 6

Overview of the task

The goal of this task is to create a spell-checker class that keeps a lexicon of words, W , in a set and implements a method, `check(s)`, that checks the spelling of the string s concerning the set of words, W . If s is in W , then the call to `check(s)` returns a list containing only s , since it is assumed to be spelled correctly in this case; if s is not in W , then the call to `check(s)` returns a list of all the words in W that could be spelled correctly. Your program should be able to handle all the common ways that s might be misspelled in W , such as swapping adjacent characters in a word or adding a single character in between.

Code explanation

Spell checker public

```
#include <iostream>
#include <string>
#include <set>
#include <vector>
using namespace std;

class SpellChecker {
private:
    set<string> lexicon;

public:
    void addToLexicon(const string& word) {
        lexicon.insert(word);
    }

    vector<string> check(const string& word) {
```

```

    if (lexicon.find(word) != lexicon.end()) {
        return {word};
    }

    vector<string> suggestions;
    for (const string& correctWord : lexicon) {
        if (isPossibleTypo(word, correctWord)) {
            suggestions.push_back(correctWord);
        }
    }

    return suggestions;
}

```

First, import the needed libraries for the program. Then a private member lexicon of type `set<string>` is included in the `SpellChecker` class for storing the words in the dictionary. To add a word to the lexicon variable, we will use the `addToLexicon` function. The `check` method accepts a string as input and returns a list containing only the words if it is found in the lexicon; if not, it suggests possible correct spellings based on frequently occurring misspelling instances. There is a `for` loop that loops through the lexicon, and then there is an `if` statement that checks if the `possibleTypo` is true; if so, it will add the word to the suggestions.

Spell checker: private

```

private:

    bool isPossibleTypo(const string& word, const string& correctWord) {
        if (word.size() == correctWord.size()) {
            return isSwapped(word, correctWord) || isInserted(word, correctWord) || isDeleted(word, correctWord) ||
isReplaced(word, correctWord);
        }
        else if (word.size() == correctWord.size() - 1) {
            return isInserted(word, correctWord);
        }
        else if (word.size() == correctWord.size() + 1) {
            return isDeleted(word, correctWord);
        }
        return false;
    }
}

```

```

bool isSwapped(const string& word, const string& correctWord) {
    for (size_t i = 0; i < word.size() - 1; ++i) {
        if (word[i] == correctWord[i + 1] && word[i + 1] == correctWord[i]) {
            return true;
        }
    }
    return false;
}

bool isInserted(const string& word, const string& correctWord) {
    for (size_t i = 0; i < correctWord.size(); ++i) {
        if (word == correctWord.substr(0, i) + correctWord.substr(i + 1)) {
            return true;
        }
    }
    return false;
}

bool isDeleted(const string& word, const string& correctWord) {
    return isInserted(correctWord, word);
}

bool isReplaced(const string& word, const string& correctWord) {
    int mismatches = 0;
    for (size_t i = 0; i < word.size(); ++i) {
        if (word[i] != correctWord[i]) {
            ++mismatches;
        }
    }
    return mismatches == 1;
}
};

```

The SpellChecker class's private methods are used to compare a word with the lexicon's correct words to see whether it's a typo or not. The functions `isPossibleTypo`, `isSwapped`, `isInserted`, `isDeleted`, and `isReplaced` are among them. The `isPossibleTypo` function is responsible for

checking words that are close to the right word but could have typos like letters that have been shifted, added, removed, or replaced. The possible typo function has if statements that check if word and correct word sizes are equal under certain conditions. If so, there will be certain functions that will run under every condition. The isSwapped function is used, for example, to determine if two letters are swapped in a misspelled word that could be close to the correct word. For loop that loops through the word size. Then there is an if statement that checks if the word and the correct word are equal to each other. If so, the function returns true; otherwise, it returns false. The isInserted function is used, for example, to determine if there is an extra letter inserted in the misspelled word or not, which could be close to the correct word. A For loop goes through the correctword variable size, Then there is an if statement that checks if the word and the correct word substring are equal to each other. If so, the function returns true, otherwise, it returns false. The isDeleted function is to check if there is a deleted letter in the misspelled word. IsDeleted just runs the isinserted function with the needed parameters. The isreplaced function is used for checking if items are replaced. Letters in the misspelled word. Create an integer for the mismatches, and then there is a for loop that loops through the word size. Then there is an if statement that checks if the word and correct are not equal to each other. If so, that counts as a mismatch.

Summary

In summary, the above C++ code effectively rearranges an array by placing elements less than or equal to k at the end and elements larger than a given pivot value, k, at the beginning. Recognizing the interplay between array manipulation and recursion in this code can be beneficial in similar array rearrangement situations.

Libraries and api used:

```
#include <iostream>
#include <string>
#include <set>
#include <vector>
```

Task 7

Overview of the task

The goal of this task is to create a program that uses shortest-path routing, in which the number of edges in a path is used to calculate path distance, to create routing tables for each node in a computer network. The connectivity data for each node in the network serves as the problem's input, as seen in the example below: Node A: B, Node A: C, Node A: D. The routing table for the node at address A consists of a set of pairs (X, C), indicating that C is the next node to send a message to (on the shortest path from A to B) to route it from A to X. Given an input list, your software should output the routing table for each network node.

Code explanation

Dijkstra algorithm

```
#include <iostream>
#include <vector>
#include <unordered_map>
#include <queue>
#include <limits>

using namespace std;

void dijkstra(const unordered_map<char, vector<char>>& graph, char startNode) {
    unordered_map<char, int> distance;
    unordered_map<char, char> previous;
    priority_queue<pair<int, char>, vector<pair<int, char>>, greater<pair<int, char>>> pq;

    for (const auto& node : graph) {
        distance[node.first] = node.first == startNode ? 0 : INT_MAX;
        pq.push({distance[node.first], node.first});
    }

    while (!pq.empty()) {
        char current = pq.top().second;
        pq.pop();

        for (char neighbor : graph.at(current)) {
            int alt = distance[current] + 1; // Assuming each edge has a weight of 1 (hop count)
            if (alt < distance[neighbor]) {
                distance[neighbor] = alt;
                previous[neighbor] = current;
                pq.push({distance[neighbor], neighbor});
            }
        }
    }
}
```

First, import the items needed for the task. Dijkstra's algorithm is a greedy algorithm that finds the shortest path from the starting node to the other nodes. The code defines a function called Dijkstra, which takes two parameters: a graph and a beginning node. Then it creates initial distances, initial maps, a priority queue, and previous map variables. The algorithm finds the shortest path by iteratively going over nodes and updating distances. The routing table for the specified starting node is finally output.

Main function

```
int main() {  
    unordered_map<char, vector<char>> graph = {  
        {'A', {'B', 'C', 'D'}},  
        {'B', {'A', 'C'}},  
        {'C', {'A', 'B', 'D'}},  
        {'D', {'A', 'C'}}  
    };  
  
    for (const auto& node : graph) {  
        dijkstra(graph, node.first);  
    }  
  
    return 0;  
}
```

In the main function there is a unordered map created that have nodes a, b, and d. then there is a for loop that will loop through every node and will run dijkstra function.

Summary

In summary, this C++ code illustrates how Dijkstra's algorithm—which finds the shortest pathways between each node and every other node in a graph—is implemented. The method quickly determines the shortest paths by storing paths and distances in priority queues and maps. Understanding and applying Dijkstra's algorithm is crucial for effectively resolving a range of graph-related problems.

Libraries and api used:

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



```
#include <queue>
#include <climits>
```

Task 8

Code explanation

Overview of the task

Task 8 requires using the memory and processing capacity of big matrices, multiplying them can computationally take more time. It is possible to multiply more effectively by using several methods. Multithreading is a productive method of cutting computation time. The goal is to write a C++ program that outputs the results after it takes two matrices and their sizes. For parallel row multiplication, the technique should leverage C++'s pthreads module.

Multiply function

```
#include <iostream>
#include <pthread.h>
#include <vector>

#define MAX_SIZE 100

using namespace std;

int matrixA[MAX_SIZE][MAX_SIZE];
int matrixB[MAX_SIZE][MAX_SIZE];
int result[MAX_SIZE][MAX_SIZE];

int rowsA, colsA, rowsB, colsB;

void* multiply(void* arg) {
```

```

int row = *((int*)arg);
for (int j = 0; j < colsB; ++j) {
    result[row][j] = 0;
    for (int k = 0; k < colsA; ++k) {
        result[row][j] += matrixA[row][k] * matrixB[k][j];
    }
}
pthread_exit(0);
}

```

First, import the needed libraries for the task. `pthread.h` is used for the pthreads and dynamic arrays. `MAX_SIZE` is a constant used to define the maximum size of the matrices. then created 2D arrays (matrix A, matrix B) and a result. The dimensions of the matrices are stored in the variables `rowsA`, `colsA`, `rowsB`, and `colsB`. The multiply function is responsible for each thread's entry point. It takes a pointer to an integer parameter that serves as the matrix's row index for computation. After updating the corresponding elements in the result matrix, the function multiplies the matrix for the given row.

Main function

```

int main() {
    // Input matrices A and B
    cout << "Enter the number of rows and columns for Matrix A: ";
    cin >> rowsA >> colsA;
    cout << "Enter the elements of Matrix A:\n";
    for (int i = 0; i < rowsA; ++i) {
        for (int j = 0; j < colsA; ++j) {
            cin >> matrixA[i][j];
        }
    }

    cout << "Enter the number of rows and columns for Matrix B: ";
    cin >> rowsB >> colsB;
    cout << "Enter the elements of Matrix B:\n";
    for (int i = 0; i < rowsB; ++i) {
        for (int j = 0; j < colsB; ++j) {
            cin >> matrixB[i][j];
        }
    }
}

```

```

// Initialize pthread variables
pthread_t threads[MAX_SIZE];
int threadArgs[MAX_SIZE];

// Create threads for parallel multiplication
for (int i = 0; i < rowsA; ++i) {
    threadArgs[i] = i;
    pthread_create(&threads[i], NULL, multiply, (void*)&threadArgs[i]);
}

// Join threads
for (int i = 0; i < rowsA; ++i) {
    pthread_join(threads[i], NULL);
}

// Display the result matrix
cout << "Resultant Matrix:\n";
for (int i = 0; i < rowsA; ++i) {
    for (int j = 0; j < colsB; ++j) {
        cout << result[i][j] << " ";
    }
    cout << endl;
}

return 0;
}

```

The primary function receives the user input and is responsible for generating threads, connecting threads, and displaying the matrix that results. The user enters his desired dimensions and elements of matrices A and B in the main function. The 2d arrays, matrixA and matrixB, hold the input. After that, threads are created for parallel multiplication using a loop. Using the pthread_create method, a thread is generated for every row in matrix A. The multiply function receives two arguments: the thread identifier and the row index.

Summary

The C++ code for this task used Pthreads to implement matrix multiplication. through utilizing multi-core systems, parallelizing the matrix multiplication process, and enhancing performance.

It is crucial to understand matrix multiplication and Pthreads to develop effective parallel algorithms.

Libraries and api used:

```
#include <iostream>
#include <pthread.h>
#include <vector>
```

Code outputs:

Task 1

```
Both arrays have the same set of numbers.

...Program finished with exit code 0
Press ENTER to exit console. □
```

Task 2

```
Time taken by in-place heap-sort: 8.78e-07s  
Time taken by standard heap-sort: 6.4e-07s
```

```
...Program finished with exit code 0  
Press ENTER to exit console.
```

Task 3

Task 4

```
4 1 2 5 9 7 8  
...Program finished with exit code 0  
Press ENTER to exit console.
```

Task 6

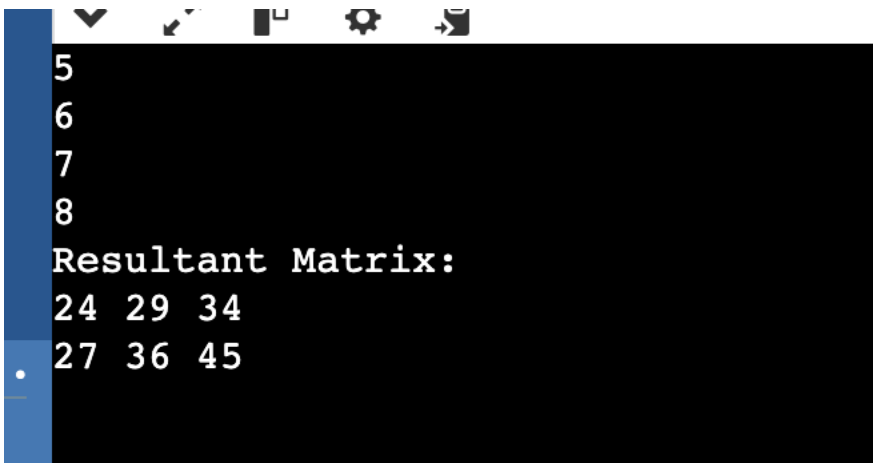
```
Did you mean: world  
...Program finished with exit code 0  
Press ENTER to exit console.
```

Task 7

```
To reach node D, next hop is: A
Routing table for node A:
To reach node B, next hop is: A
To reach node C, next hop is: A
To reach node D, next hop is: A
```

Task 8

```
Enter the number of rows and columns for Matrix A: 2 3
Enter the elements of Matrix A:
2 3
7
9
0
6
Enter the number of rows and columns for Matrix B: 2 3
Enter the elements of Matrix B:
```



The screenshot shows a terminal window with a dark background and a blue vertical bar on the left. At the top, there is a toolbar with icons for a dropdown menu, a cursor, a square, a gear, and a right-pointing arrow. The terminal text shows the numbers 5, 6, 7, and 8 on separate lines. Below these is the text "Resultant Matrix:", followed by two rows of numbers: "24 29 34" and "27 36 45". A cursor is visible at the end of the second row.

```
5
6
7
8
Resultant Matrix:
24 29 34
27 36 45
```

Conclusion

To sum it up, I finished all the assignments and learned essential information about the applications of each data structure and method. I learned about a lot of new algorithms that I was unfamiliar

with in this coursework. I can declare with confidence that I have learned the time and space complexity of each algorithm, as well as its efficiency, how it is implemented when to use it (i.e., the most common ways to use it), why [it is better than other solutions] (i.e., what makes it the most optimal solution), and what can be changed or improved to better suit my needs (i.e., adhering to the task's requirements). The activities that I was given tested my thinking and helped me improve my problem-solving abilities. Additionally, these activities assisted me in gaining fresh knowledge and expertise in the sector. Additionally, these tasks assisted me in developing new DSA and coding abilities and experiences. They have altered my understanding of and approach to code analysis.

For every assignment, I described the issue, outlined a strategy for resolving it, put the solution (code with explanation) into practice, clarified the intricacies of the algorithms employed, presented potential alternatives and their advantages and disadvantages, and tested the software. Every activity addressed a particular component of this module, guaranteeing that all topics and learning goals were covered

Appendix:

Task 1

```
#include <iostream>
#include <unordered_set>
#include <vector>
using namespace std;

bool checkEqualityOfArrays(const vector<int>& arr1, const vector<int>& arr2) {
    if (arr1.size() != arr2.size()) {
        return false;
    }

    unordered_set<int> set1(arr1.begin(), arr1.end());
    unordered_set<int> set2(arr2.begin(), arr2.end());

    return set1 == set2;
}

int main() {
    vector<int> array1 = {2, 4, 1, 3};
    vector<int> array2 = {3, 1, 4, 2};
```

```

if (checkEqualityOfArrays(array1, array2)) {
    cout << "Both arrays have the same set of numbers." << std::endl;
} else {
    cout << "Arrays have different sets of numbers." << std::endl;
}

return 0;
}

```

Task 2

```

using namespace std;

#include <iostream>
#include <vector>
#include <algorithm>
#include <chrono>
//standard heap sort algorithm
void heapify1 (int arr[], int n, int i) {
    int largest = i;
    int left = 2 * i + 1;
    int right = 2 * i + 2;

    if (left < n && arr[left] > arr[largest])
        largest = left;

    if (right < n && arr[right] > arr[largest])
        largest = right;

    if (largest != i) {
        std::swap(arr[i], arr[largest]);
        heapify1(arr, n, largest);
    }
}

```



```

void heapSort(int arr[], int n) {
    for (int i = n / 2 - 1; i >= 0; i--)
        heapify1 (arr, n, i);

    for (int i = n - 1; i > 0; i--) {
        std::swap(arr[0], arr[i]);
        heapify1(arr, i, 0);
    }
}

//in place heap sort
void heapify(vector<int>& arr, int n, int i) {
    int largest = i;
    int left = 2 * i + 1;
    int right = 2 * i + 2;

    if (left < n && arr[left] > arr[largest])
        largest = left;

    if (right < n && arr[right] > arr[largest])
        largest = right;

    if (largest != i) {
        swap(arr[i], arr[largest]);
        heapify(arr, n, largest);
    }
}

void inPlaceHeapSort(vector<int>& arr) {
    int n = arr.size();

    for (int i = n / 2 - 1; i >= 0; i--)
        heapify(arr, n, i);

    for (int i = n - 1; i > 0; i--) {
        swap(arr[0], arr[i]);
        heapify(arr, i, 0);
    }
}

```

```

    }
}

int main() {
    vector<int> arr = {12, 11, 13, 5, 6, 7};
    int arrr[]={12, 11, 13, 5, 6, 7};
    int n = sizeof(arrr) / sizeof(arrr[0]);
    inPlaceHeapSort(arrr);

    auto start = chrono::high_resolution_clock::now();
    inPlaceHeapSort(arr);
    auto end = chrono::high_resolution_clock::now();
    chrono::duration<double> elapsed_seconds = end - start;
    cout << "Time taken by in-place heap-sort: " << elapsed_seconds.count() << "s\n";

    start = chrono::high_resolution_clock::now();
    heapSort(arrr, n);
    end = chrono::high_resolution_clock::now();
    elapsed_seconds = end - start;
    cout << "Time taken by standard heap-sort: " << elapsed_seconds.count() << "s\n";

    return 0;
}

```

Task 3

```

#include <iostream>
#include <fstream>
#include <chrono>
#include <iomanip>
using namespace std;

ofstream dataFile("sort_results.csv");

int heap(int arr[], int n, int i)

```

```

{
    int numcomp = 0;
    int minvl = i;
    int l = 2 * i + 1;
    int r = 2 * i + 2;
    if (l < n)
    {
        numcomp++;
        if (arr[l] < arr[minvl])
        {
            minvl = l;
        }
    }
    if (r < n)
    {
        numcomp++;
        if (arr[r] < arr[minvl])
        {
            minvl = r;
        }
    }
    if (minvl != i)
    {
        swap(arr[i], arr[minvl]);
        numcomp += heap(arr, n, minvl);
    }
    return numcomp;
}

```

```

int heapsocount(int arr[], int n)
{
    int numcomp = 0;
    for (int x = (n - 2) / 2; x >= 0; x--)
    {
        numcomp += heap(arr, n, x);
    }
    for (int x = n - 1; x > 0; x--)

```

```

{
    swap(arr[0], arr[x]);
    numcomp += heap(arr, x, 0);
}
return numcomp;
}

```

```

int bubblecount(int ar[], int n)
{
    int numcomp = 0;
    bool swapcurs;
    for (int i = 0; i < n - 1; i++)
    {
        swapcurs = false;
        for (int x = 0; x < n - i - 1; x++)
        {
            if (ar[x] > ar[x+1])
            {
                swap(ar[x], ar[x+1]);
                swapcurs = true;
            }
            numcomp++;
        }
        if (!swapcurs)
        {
            break;
        }
    }
    return numcomp;
}

```

```

int selectioncount(int ar[], int n)
{
    int numcomp = 0;
    int min;
    for (int i = 0; i < n - 1; i++)
    {

```

```
    min = i;
    for (int j = i + 1; j < n; j++)
    {
        if (ar[j] < ar[min])
        {
            min = j;
        }
        numcomp++;
    }
    swap(ar[i], ar[min]);
}
return numcomp;
}
```

```
int insertsocount(int ar[], int n)
{
    int s = 0;
    int numcomp = 0;

    for (int i = 1; i < n; i++)
    {
        s = ar[i];
        for (int j = i - 1; j > -1; j--)
        {
            numcomp++;
            if (s < ar[j])
            {
                swap(ar[j], ar[j+1]);
            }
            else
            {
                break;
            }
        }
    }
    return numcomp;
}
```

```
void reset(int te[], int ar[], int z )
```

```
{  
    for (int x = 0; x < z; x++)  
    {  
        te[x] = ar[x];  
    }  
}
```

```
void pTable(int n, string artype, string sotype, int rt, double extime)
```

```
{  
    cout << left << setw(20) << n << left << setw(20) << artype << left << setw(20) << sotype << left << setw(25) << rt  
<< left << setw(20) << extime << "\n";  
}
```

```
void exsorts(int te[], int ar[], int x, string artype, string soalg)
```

```
{  
    reset(te, ar, x);  
    auto start = chrono::high_resolution_clock::now();  
    int st;  
    if (soalg == "Heap")  
    {  
        st = heapsocount(te, x);  
    }  
    else if (soalg == "Bubble")  
    {  
        st = bubblcount(te, x);  
    }  
    else if (soalg == "Selection")  
    {  
        st = selectioncount(te, x);  
    }  
    else  
    {  
        st = insertsocount(te, x);  
    }  
    auto end = chrono::high_resolution_clock::now();
```

```

double exec_time = chrono::duration_cast<chrono::nanoseconds>(end - start).count() * 1e-9;
dataFile << x << "," << artype << "," << soalg << "," << st << "," << exec_time << "\n";
pTable(x, artype, soalg, st, exec_time);
}

void rsorts(int ar[], string artype, int n)
{
    int* temp = new int[n];
    exsorts(temp, ar, n, artype, "Heap");
    exsorts(temp, ar, n, artype, "Bubble");
    exsorts(temp, ar, n, artype, "Selection");
    exsorts(temp, ar, n, artype, "Insertion");
}

void testcompar()
{
    srand(time(0));

    dataFile << "Size of Array,Type of Array,Sorting Algorithm,Number of Comparisons,Execution Time\n";
    cout << left << setw(20) << "Size of Array" << left << setw(20) << "Type of Array"
    << left << setw(20) << "Sorting Algorithm" << left << setw(25) << "Number of Comparisons"
    << left << setw(20) << "Execution Time" << "\n";

    for (int m = 1; m < 31; m++)
    {
        int* randarr = new int[m];
        int* sortedarr = new int[m];
        int* invarr = new int[m];
        int number1 = 0;
        int number2 = 0;
        for (int x = 0; x < m; x++)
        {
            randarr[x] = rand() % (m * 20);
            number1 += rand() % (m * 20);
            sortedarr[x] = number1;
            number2 += rand() % (m * 20);
            invarr[m-x-1] = number2;
        }
    }
}

```

```

    }

    rsorts(randarr, "Random", m);
    rsorts(sortedarr, "Sorted", m);
    rsorts(invarr, "Inversely-sorted", m);

    delete[] randarr;
    delete[] sortedarr;
    delete[] invarr;
}

}

int main()
{
    testcompar();
    dataFile.close();
    return 0;
}

```

Task 4

```

#include <iostream>
#include <vector>
using namespace std;

void rearrangeAr(vector<int>& arr, int k, int start, int end) {
    if (start >= end) return;

    if (arr[start] > k && arr[end] <= k) {
        swap(arr[start], arr[end]);
        rearrangeAr(arr, k, start + 1, end - 1);
    } else if (arr[start] <= k) {
        rearrangeAr(arr, k, start + 1, end);
    } else if (arr[end] > k) {
        rearrangeAr(arr, k, start, end - 1);
    }
}

```



```

}

int main() {
    vector<int> arr = {4, 7, 2, 9, 5, 1, 8};
    int k = 5;

    rearrangeAr(arr, k, 0, arr.size() - 1);

    for (int num : arr) {
        cout << num << " ";
    }

    return 0;
}

```

Task 6

```

#include <iostream>
#include <string>
#include <set>
#include <vector>
using namespace std;

class SpellChecker {
private:
    set<string> lexicon;

public:
    void addToLexicon(const string& word) {
        lexicon.insert(word);
    }

    vector<string> check(const string& word) {
        if (lexicon.find(word) != lexicon.end()) {
            return {word};
        }
    }
}

```

```

vector<string> suggestions;
for (const string& correctWord : lexicon) {
    if (isPossibleTypo(word, correctWord)) {
        suggestions.push_back(correctWord);
    }
}

return suggestions;
}

private:

bool isPossibleTypo(const string& word, const string& correctWord) {
    if (word.size() == correctWord.size()) {
        return isSwapped(word, correctWord) || isInserted(word, correctWord) || isDeleted(word, correctWord) ||
isReplaced(word, correctWord);
    }
    else if (word.size() == correctWord.size() - 1) {
        return isInserted(word, correctWord);
    }
    else if (word.size() == correctWord.size() + 1) {
        return isDeleted(word, correctWord);
    }
    return false;
}

bool isSwapped(const string& word, const string& correctWord) {
    for (size_t i = 0; i < word.size() - 1; ++i) {
        if (word[i] == correctWord[i + 1] && word[i + 1] == correctWord[i]) {
            return true;
        }
    }
    return false;
}

bool isInserted(const string& word, const string& correctWord) {
    for (size_t i = 0; i < correctWord.size(); ++i) {

```

```

        if (word == correctWord.substr(0, i) + correctWord.substr(i + 1)) {
            return true;
        }
    }
    return false;
}

```

```

bool isDeleted(const string& word, const string& correctWord) {
    return isInserted(correctWord, word);
}

```

```

bool isReplaced(const string& word, const string& correctWord) {
    int mismatches = 0;
    for (size_t i = 0; i < word.size(); ++i) {
        if (word[i] != correctWord[i]) {
            ++mismatches;
        }
    }
    return mismatches == 1;
}
};

```

```

int main() {
    SpellChecker spellChecker;
    spellChecker.addToLexicon("hello");
    spellChecker.addToLexicon("world");
    spellChecker.addToLexicon("goodbye");

    vector<std::string> suggestions = spellChecker.check("wold");
    for (const std::string& suggestion : suggestions) {
        cout << "Did you mean: " << suggestion << endl;
    }

    return 0;
}

```

Task 7

```
#include <iostream>
#include <vector>
#include <unordered_map>
#include <queue>
#include <climits>

using namespace std;

void dijkstra(const unordered_map<char, vector<char>>& graph, char startNode) {
    unordered_map<char, int> distance;
    unordered_map<char, char> previous;
    priority_queue<pair<int, char>, vector<pair<int, char>>, greater<pair<int, char>>> pq;

    for (const auto& node : graph) {
        distance[node.first] = node.first == startNode ? 0 : INT_MAX;
        pq.push({distance[node.first], node.first});
    }

    while (!pq.empty()) {
        char current = pq.top().second;
        pq.pop();

        for (char neighbor : graph.at(current)) {
            int alt = distance[current] + 1; // Assuming each edge has a weight of 1 (hop count)
            if (alt < distance[neighbor]) {
                distance[neighbor] = alt;
                previous[neighbor] = current;
                pq.push({distance[neighbor], neighbor});
            }
        }
    }
}
```

```

// Output routing table
cout << "Routing table for node " << startNode << ":\n";
for (const auto& node : distance) {
    if (node.first != startNode) {
        cout << "To reach node " << node.first << ", next hop is: " << previous[node.first] << endl;
    }
}
}

int main() {
    unordered_map<char, vector<char>>> graph = {
        {'A', {'B', 'C', 'D'}},
        {'B', {'A', 'C'}},
        {'C', {'A', 'B', 'D'}},
        {'D', {'A', 'C'}}
    };

    for (const auto& node : graph) {
        dijkstra(graph, node.first);
    }

    return 0;
}

```

Task 8

```

#include <iostream>
#include <pthread.h>
#include <vector>

#define MAX_SIZE 100

using namespace std;

```

```

int matrixA[MAX_SIZE][MAX_SIZE];
int matrixB[MAX_SIZE][MAX_SIZE];
int result[MAX_SIZE][MAX_SIZE];

int rowsA, colsA, rowsB, colsB;

void* multiply(void* arg) {
    int row = *((int*)arg);
    for (int j = 0; j < colsB; ++j) {
        result[row][j] = 0;
        for (int k = 0; k < colsA; ++k) {
            result[row][j] += matrixA[row][k] * matrixB[k][j];
        }
    }
    pthread_exit(0);
}

int main() {
    // Input matrices A and B
    cout << "Enter the number of rows and columns for Matrix A: ";
    cin >> rowsA >> colsA;
    cout << "Enter the elements of Matrix A:\n";
    for (int i = 0; i < rowsA; ++i) {
        for (int j = 0; j < colsA; ++j) {
            cin >> matrixA[i][j];
        }
    }

    cout << "Enter the number of rows and columns for Matrix B: ";
    cin >> rowsB >> colsB;
    cout << "Enter the elements of Matrix B:\n";
    for (int i = 0; i < rowsB; ++i) {
        for (int j = 0; j < colsB; ++j) {
            cin >> matrixB[i][j];
        }
    }
}

```

```
// Initialize pthread variables
pthread_t threads[MAX_SIZE];
int threadArgs[MAX_SIZE];

// Create threads for parallel multiplication
for (int i = 0; i < rowsA; ++i) {
    threadArgs[i] = i;
    pthread_create(&threads[i], NULL, multiply, (void*)&threadArgs[i]);
}

// Join threads
for (int i = 0; i < rowsA; ++i) {
    pthread_join(threads[i], NULL);
}

// Display the result matrix
cout << "Resultant Matrix:\n";
for (int i = 0; i < rowsA; ++i) {
    for (int j = 0; j < colsB; ++j) {
        cout << result[i][j] << " ";
    }
    cout << endl;
}

return 0;
}
```