DAA Prac 05

Name: Vijiyant Tanaji shejwalkar

Reg N0: 2020BIT057

1. Binary Search

```
#include<iostream>
using namespace std;
int main(){
   int arr[9]={1,2,3,4,5,6,7,8,9};
   int size=sizeof(arr)/sizeof(arr[0]);
   int l=0,e=size-1,mid=(l+e)/2;
   cout<<"Enter a number from 1 to 9:";
   int n;
   cin>>n;
   while(l<=e){
      if(arr[mid]==n){
         cout<<"Index of entered number is:"<<mid<<endl;
         break;
      }
      else if(arr[mid]>n){
         e=mid-1;
         mid=(e+l)/2;
      }
      else{
         l=mid+1;
         mid=(e+l)/2;
      }
```

```
    }
}
```

2. Merge Sort

```
// C++ program for Merge Sort
#include <iostream>
using namespace std;

// Merges two subarrays of array[].
// First subarray is arr[begin..mid]
// Second subarray is arr[mid+1..end]
void merge(int array[], int const left, int const mid,
                                int const right)
{
        auto const subArrayOne = mid - left + 1;
        auto const subArrayTwo = right - mid;

        // Create temp arrays
        auto *leftArray = new int[subArrayOne],
                        *rightArray = new int[subArrayTwo];

        // Copy data to temp arrays leftArray[] and rightArray[]
        for (auto i = 0; i < subArrayOne; i++)
                        leftArray[i] = array[left + i];
        for (auto j = 0; j < subArrayTwo; j++)
                        rightArray[j] = array[mid + 1 + j];
```

```
auto indexOfSubArrayOne
                = 0, // Initial index of first sub-array
                indexOfSubArrayTwo
                = 0; // Initial index of second sub-array
int indexOfMergedArray
                = left; // Initial index of merged array


// Merge the temp arrays back into array[left..right]
while (indexOfSubArrayOne < subArrayOne
                && indexOfSubArrayTwo < subArrayTwo) {
                if (leftArray[indexOfSubArrayOne]
                                <= rightArray[indexOfSubArrayTwo]) {
                                array[indexOfMergedArray]
                                                = leftArray[indexOfSubArrayOne];
                                indexOfSubArrayOne++;
                }
                else {
                                array[indexOfMergedArray]
                                                = rightArray[indexOfSubArrayTwo];
                                indexOfSubArrayTwo++;
                }
                indexOfMergedArray++;
}
// Copy the remaining elements of
// left[], if there are any
while (indexOfSubArrayOne < subArrayOne) {
                array[indexOfMergedArray]
                                = leftArray[indexOfSubArrayOne];
                indexOfSubArrayOne++;
```

```
                indexOfMergedArray++;
        }
        // Copy the remaining elements of
        // right[], if there are any
        while (indexOfSubArrayTwo < subArrayTwo) {
                array[indexOfMergedArray]
                        = rightArray[indexOfSubArrayTwo];
                indexOfSubArrayTwo++;
                indexOfMergedArray++;
        }
        delete[] leftArray;
        delete[] rightArray;
}

void mergeSort(int array[], int const begin, int const end)
{
        if (begin >= end)
                return; // Returns recursively

        auto mid = begin + (end - begin) / 2;
        mergeSort(array, begin, mid);
        mergeSort(array, mid + 1, end);
        merge(array, begin, mid, end);
}

void printArray(int A[], int size)
{
        for (auto i = 0; i < size; i++)
                cout << A[i] << " ";
```

```
}

int main()
{
        int arr[] = { 12, 11, 13, 5, 6, 7 };
        auto arr_size = sizeof(arr) / sizeof(arr[0]);

        cout << "Given array is \n";
        printArray(arr, arr_size);

        mergeSort(arr, 0, arr_size - 1);

        cout << "\nSorted array is \n";
        printArray(arr, arr_size);
        return 0;
}
```

3. Quicksort

```
/* C++ implementation of QuickSort */
#include <bits/stdc++.h>
using namespace std;

// A utility function to swap two elements
void swap(int* a, int* b)
{
        int t = *a;
        *a = *b;
```

```
        *b = t;
}

/* This function takes last element as pivot, places
the pivot element at its correct position in sorted
array, and places all smaller (smaller than pivot)
to left of pivot and all greater elements to right
of pivot */
int partition(int arr[], int low, int high)
{
        int pivot = arr[high]; // pivot
        int i
                = (low
                - 1); // Index of smaller element and indicates
                                        // the right position of pivot found so far

        for (int j = low; j <= high - 1; j++) {
                // If current element is smaller than the pivot
                if (arr[j] < pivot) {
                        i++; // increment index of smaller element
                        swap(&arr[i], &arr[j]);
                }
        }
        swap(&arr[i + 1], &arr[high]);
        return (i + 1);
}

/* The main function that implements QuickSort
arr[] --> Array to be sorted,
```

```
low --> Starting index,
high --> Ending index */
void quickSort(int arr[], int low, int high)
{
	if (low < high) {
		/* pi is partitioning index, arr[p] is now
		at right place */
		int pi = partition(arr, low, high);

		// Separately sort elements before
		// partition and after partition
		quickSort(arr, low, pi - 1);
		quickSort(arr, pi + 1, high);
	}
}

/* Function to print an array */
void printArray(int arr[], int size)
{
	int i;
	for (i = 0; i < size; i++)
		cout << arr[i] << " ";
	cout << endl;
}

int main()
{
	int arr[] = { 10, 7, 8, 9, 1, 5 };
	int n = sizeof(arr) / sizeof(arr[0]);
```

```
	quickSort(arr, 0, n - 1);
	cout << "Sorted array: \n";
	printArray(arr, n);
	return 0;
}
```

4. Strassen's Matrix multiplication

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

#define ROW_1 4
#define COL_1 4

#define ROW_2 4
#define COL_2 4

void print(string display, vector<vector<int> > matrix,
		int start_row, int start_column, int end_row,
		int end_column)
{
	cout << endl << display << " =>" << endl;
	for (int i = start_row; i <= end_row; i++) {
		for (int j = start_column; j <= end_column; j++) {
			cout << setw(10);
			cout << matrix[i][j];
		}
		cout << endl;
```

```
        }
        cout << endl;
        return;
}

void add_matrix(vector<vector<int> > matrix_A,
                                        vector<vector<int> > matrix_B,
                                        vector<vector<int> >& matrix_C,
                                        int split_index)
{
        for (auto i = 0; i < split_index; i++)
                for (auto j = 0; j < split_index; j++)
                        matrix_C[i][j]
                                        = matrix_A[i][j] + matrix_B[i][j];
}

vector<vector<int> >
multiply_matrix(vector<vector<int> > matrix_A,
                                        vector<vector<int> > matrix_B)
{
        int col_1 = matrix_A[0].size();
        int row_1 = matrix_A.size();
        int col_2 = matrix_B[0].size();
        int row_2 = matrix_B.size();

        if (col_1 != row_2) {
                cout << "\nError: The number of columns in Matrix "
                                        "A must be equal to the number of rows in "
                                        "Matrix B\n";
```

```
                return {};
}

vector<int> result_matrix_row(col_2, 0);
vector<vector<int> > result_matrix(row_1,

result_matrix_row);

if (col_1 == 1)
                result_matrix[0][0]
                                = matrix_A[0][0] * matrix_B[0][0];
else {
                int split_index = col_1 / 2;

                vector<int> row_vector(split_index, 0);
                vector<vector<int> > result_matrix_00(split_index,

                                row_vector);
                vector<vector<int> > result_matrix_01(split_index,

                                row_vector);
                vector<vector<int> > result_matrix_10(split_index,

                                row_vector);
                vector<vector<int> > result_matrix_11(split_index,

                                row_vector);

                vector<vector<int> > a00(split_index, row_vector);
                vector<vector<int> > a01(split_index, row_vector);
                vector<vector<int> > a10(split_index, row_vector);
```

```
        vector<vector<int> > a11(split_index, row_vector);
        vector<vector<int> > b00(split_index, row_vector);
        vector<vector<int> > b01(split_index, row_vector);
        vector<vector<int> > b10(split_index, row_vector);
        vector<vector<int> > b11(split_index, row_vector);

        for (auto i = 0; i < split_index; i++)
            for (auto j = 0; j < split_index; j++) {
                a00[i][j] = matrix_A[i][j];
                a01[i][j] = matrix_A[i][j + split_index];
                a10[i][j] = matrix_A[split_index + i][j];
                a11[i][j] = matrix_A[i + split_index]
[j + split_index];
                b00[i][j] = matrix_B[i][j];
                b01[i][j] = matrix_B[i][j + split_index];
                b10[i][j] = matrix_B[split_index + i][j];
                b11[i][j] = matrix_B[i + split_index]
[j + split_index];
            }

        add_matrix(multiply_matrix(a00, b00),
                multiply_matrix(a01, b10),
                result_matrix_00, split_index);
        add_matrix(multiply_matrix(a00, b01),
                multiply_matrix(a01, b11),
                result_matrix_01, split_index);
        add_matrix(multiply_matrix(a10, b00),
```

```
                multiply_matrix(a11, b10),
                result_matrix_10, split_index);
    add_matrix(multiply_matrix(a10, b01),
                multiply_matrix(a11, b11),
                result_matrix_11, split_index);

    for (auto i = 0; i < split_index; i++)
        for (auto j = 0; j < split_index; j++) {
            result_matrix[i][j]
                    = result_matrix_00[i][j];
            result_matrix[i][j + split_index]
                    = result_matrix_01[i][j];
            result_matrix[split_index + i][j]
                    = result_matrix_10[i][j];
            result_matrix[i + split_index]
                                        [j + split_index]
                    = result_matrix_11[i][j];
        }

    result_matrix_00.clear();
    result_matrix_01.clear();
    result_matrix_10.clear();
    result_matrix_11.clear();
    a00.clear();
    a01.clear();
    a10.clear();
    a11.clear();
    b00.clear();
    b01.clear();
```

```
                b10.clear();
                b11.clear();
        }
        return result_matrix;
}

int main()
{
        vector<vector<int> > matrix_A = { { 1, 1, 1, 1 },
        { 2, 2, 2, 2 },
        { 3, 3, 3, 3 },
        { 2, 2, 2, 2 } };

        print("Array A", matrix_A, 0, 0, ROW_1 - 1, COL_1 - 1);

        vector<vector<int> > matrix_B = { { 1, 1, 1, 1 },
        { 2, 2, 2, 2 },
        { 3, 3, 3, 3 },
        { 2, 2, 2, 2 } };

        print("Array B", matrix_B, 0, 0, ROW_2 - 1, COL_2 - 1);

        vector<vector<int> > result_matrix(
                multiply_matrix(matrix_A, matrix_B));
```

```
        print("Result Array", result_matrix, 0, 0, ROW_1 - 1,
                        COL_2 - 1);
}

// Time Complexity: O(n^3)
```