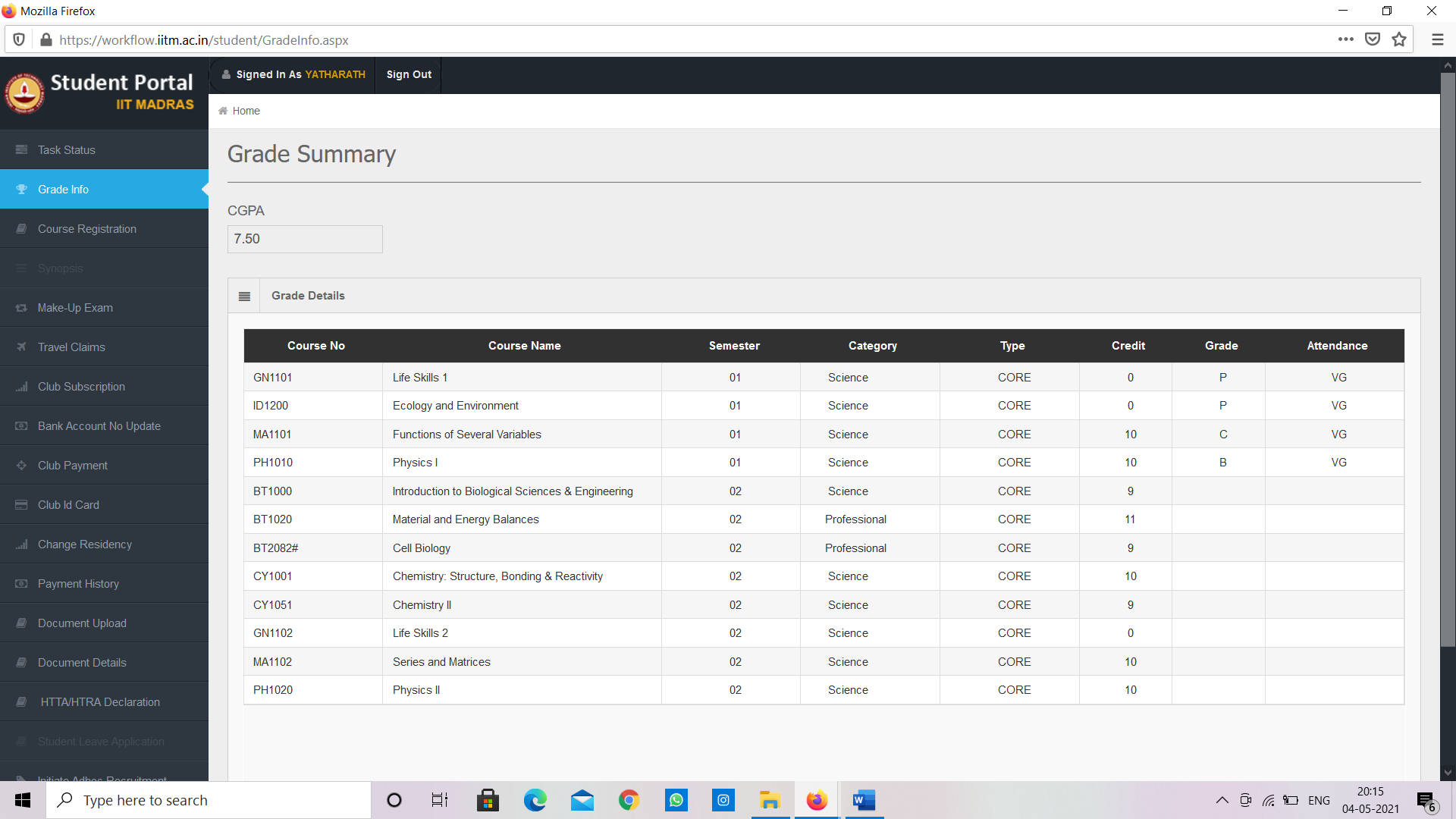
**Name:** Yatharath

**Roll No:** BS20B039

**PHONE:** 9958158847

**CGPA:** 7.5

**Email:** [bs20b039@smail.iitm.ac.in](mailto:bs20b039@smail.iitm.ac.in)



**TASK 1**

* Memory Hierarchy in modern computers

Why memory Hierarchy?

Previously, the designing of a computer system was done without memory hierarchy, and the speed gap among the main memory as well as the CPU registers enhances because of the huge disparity in access time, which will cause the lower performance of the system. So, the enhancement was mandatory. The enhancement of this was designed in the memory hierarchy model due to the system’s performance increase as it ensures a balance between access time, cost and speed.

About memory hierarchy

On a broad scale Memory hierarchy is the division of the system’s memory in segments in order to create a balance between access time, cost and speed by assigning different kind of tasks to different segments, tasks are divided in such a way that we don’t need go much down in the hierarchy. Memory is divided into segments namely CPU registers, cache memory, main memory, magnetic tapes, magnetic discs.

Volatile memory: Memory which becomes inactive as we shut down the power is volatile memory. Registers, Cache memory, main memory are the examples of volatile memory.

Non-Volatile memory: Memory which is sort of permanent and remains even after the power is turned off. For e.g. Magnetic tapes and magnetic discs.

This Memory Hierarchy Design is divided into 2 main types:

1. **External Memory or Secondary Memory –**  
   Comprising of Magnetic Disk, Optical Disk, Magnetic Tape i.e. peripheral storage devices which are accessible by the processor via I/O Module.
2. **Internal Memory or Primary Memory –**  
   Comprising of Main Memory, Cache Memory & CPU registers. This is directly accessible by the processor.

We can infer the following characteristics of Memory Hierarchy Design:

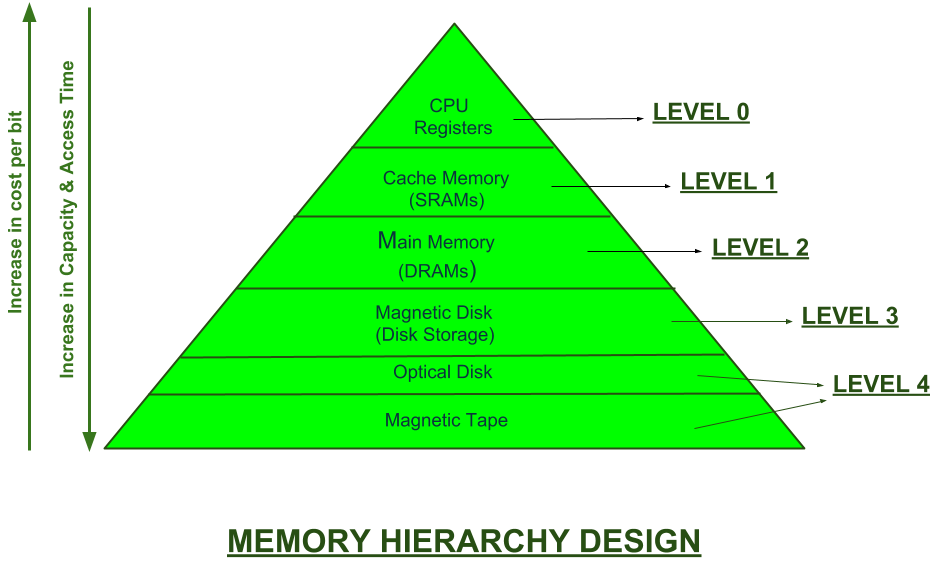
1. **Capacity:**  
   It is the global volume of information the memory can store. As we move from top to bottom in the Hierarchy, the capacity increases.
2. **Access Time:**  
   It is the time interval between the read/write request and the availability of the data. As we move from top to bottom in the Hierarchy, the access time increases.
3. **Performance:**  
   If, like earlier times the computer system is designed without Memory Hierarchy design, the speed gap increases between the CPU registers and Main Memory due to large difference in access time. This results in lower performance of the system and thus, enhancement was required. This enhancement was made in the form of Memory Hierarchy Design because of which the performance of the system increases. One of the

most significant ways to increase system performance is minimizing how far down the memory hierarchy one has to go to manipulate data.

1. **Cost per bit:**  
   As we move from bottom to top in the Hierarchy, the cost per bit increases i.e. Internal Memory is costlier than External Memory.

The image shown below describes memory hierarchy model of modern computers.

In the image shown below as we move from bottom to top, cost per bit increases, capacity decreases, access time decreases, for registers access time is of the order of nano seconds.



* About RAM machine model and External memory model.
* RAM machine model of computation calculates the total run time of an algorithm by calculating the number of steps involved in running a particular program, it does sum up the number of operations involved. An average computer performs 10^8 operations per second hence a program is efficient for even large inputs as it performs less than 10^8 operations per second.

An example of such computation: If we look at a basic program of calculating sum of given N natural numbers, we can have two ways. One way can be writing a simple algorithm by stating a formula for the required sum which we know is (N)\*(N+1)/2, this algorithm performs only one operation which can be easily done within a second. There can be another fancy way, in this we may use a loop which runs N times and in each step it performs an operation sum = sum + i where sum is a variable used to store the required sum and i is an iterator, this program will for sure perform N operations, however if N becomes larger than 10^8 then our program becomes inefficient.

* External machine model: The external-memory model is sometimes also known as the I/O model or disk access model (DAM). It captures a two-layered memory hierarchy. The model is based on a computer with a CPU connected directly to a fast cache of size M, which is connected to a much larger and slower disk. Both the cache and disk are divided into blocks of size B; the cache thus holds M/B blocks, while the disk can hold many more. The CPU can only operate directly on the data stored in cache. Algorithms can make memory transfer operations, which read a block from disk to cache, or write a block from cache to disk. The cost of an algorithm is the number of memory transfers required. operations on cached data are considered free. Clearly any algorithm that has running time T(N) in the RAM model can be trivially converted into an external-memory algorithm that requires no more than T(N) memory transfers, by ignoring locality. We want to do better than this. Ideally, we would like to achieve T(N)/B, but this optimum is often hard to achieve.

REFERENCES

* Geeks for geeks
* Courses.csail.mit.edu

**TASK 2**

Why cache oblivious model?

* As mentioned in the task 1, external memory model works on calculating the number of transfers between the two level of memory hierarchy for measuring the time taken to run an algorithm. It is done in order to make an algorithm run in very less time as compared to RAM machine model.
* Achieving this becomes difficult as it evolves taking in parameters like block size (for dividing the cache) which is different for different systems and devices.
* In order to tackle this problem, cache oblivious model has been designed, as the name suggests this model is oblivious about cache i.e it doesn’t care about what is going around in cache by tackling problems with the help of algorithms.
* Another advantage of this model over RAM machine model is that algorithms in cache oblivious model of time complexity O(n) can be even faster than algorithms of time complexity O(logn) in RAM machine model, this is due to the dispersed RAM memory access.
* About cache oblivious model
* Unlike external memory model cache oblivious model doesn’t take in the parameter like block size.
* In computing, a **cache-oblivious algorithm** is an algorithm designed to take advantage of a CPU cache without having the size of the cache as an explicit parameter. An **optimal cache-oblivious algorithm** is a cache-oblivious algorithm that uses the cache optimally (in an asymptotic sense, ignoring constant factors). Thus, a cache-oblivious algorithm is designed to perform well, without modification, on multiple machines with different cache sizes, or for a memory hierarchy with different levels of cache having different sizes. Cache-oblivious algorithms are contrasted with explicit blocking*,* as in loop nested optimization, which explicitly breaks a problem into blocks that are optimally sized for a given cache.
* Typically, a cache-oblivious algorithm works by a recursive divide-and-conquer, where the problem is divided into smaller and smaller subproblems. Eventually, one reaches a subproblem size that fits into cache, regardless of the cache size. For example, an optimal cache-oblivious matrix multiplication is obtained by recursively dividing each matrix into four sub-matrices to be multiplied, multiplying the submatrices eventually making up block optimally sized for cache and then performing operations.
* Heap sorting algorithm

Tree data structure in brief

**Trees:** Unlike Arrays, Linked Lists, Stack and queues, which are linear data structures, trees are hierarchical data structures.  
**Tree Vocabulary:**The topmost node is called root of the tree. The elements that are directly under an element are called its children. The element directly above something is called its parent. For example, ‘a’ is a child of ‘f’, and ‘f’ is the parent of ‘a’. Finally, elements with no children are called leaves.

tree

----

j <-- root

/ \

f k

/ \ \

a h z <-- leaves

Basic implementation

#include <bits/stdc++.h>

using namespace std;

struct Node {

    int data;

    struct Node\* left;

    struct Node\* right;

    // val is the key or the value that

    // has to be added to the data part

    Node(int val)

    {

        data = val;

        // Left and right child for node

        // will be initialized to null

        left = NULL;

        right = NULL;

    }

};

int main()

{

    /\*create root\*/

    struct Node\* root = new Node(1);

    /\* following is the tree after above statement

             1

            / \

          NULL NULL

    \*/

    root->left = new Node(2);

    root->right = new Node(3);

    /\* 2 and 3 become left and right children of 1

                    1

                  /    \

                 2       3

               /  \     /  \

            NULL NULL  NULL NULL

    \*/

    root->left->left = new Node(4);

    /\* 4 becomes left child of 2

               1

            /     \

           2       3

          / \     / \

         4  NULL NULL NULL

        / \

     NULL NULL

    \*/

    return 0;

}

# **Binary Tree Data Structure**

A tree whose elements have at most 2 children is called a binary tree. Since each element in a binary tree can have only 2 children, we typically name them the left and right child.



* HeapSort

Heap sort is a comparison based sorting technique based on Binary Heap data structure. It is similar to selection sort where we first find the minimum element and place the minimum element at the beginning. We repeat the same process for the remaining elements.

**What is binary heap?**   
Let us first define a Complete Binary Tree. A complete binary tree is a binary tree in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible.  
A binary heap is a Complete Binary Tree where items are stored in a special order such that value in a parent node is greater(or smaller) than the values in its two children nodes. The former is called as max heap and the latter is called min-heap. The heap can be represented by a binary tree or array.

**Why array-based representation for Binary Heap?**   
Since a Binary Heap is a Complete Binary Tree, it can be easily represented as an array and the array-based representation is space-efficient. If the parent node is stored at index I, the left child can be calculated by 2 \* I + 1 and right child by 2 \* I + 2 (assuming the indexing starts at 0).

**Heap Sort Algorithm for sorting in increasing order:**   
**1.** Build a max heap from the input data.   
**2.** At this point, the largest item is stored at the root of the heap. Replace it with the last item of the heap followed by reducing the size of heap by 1. Finally, heapify the root of the tree.   
**3.** Repeat step 2 while size of heap is greater than 1.

**How to build the heap?**   
Heapify procedure can be applied to a node only if its children nodes are heapified. So the heapification must be performed in the bottom-up order.  
Let’s understand with the help of an example:

Input data: 4, 10, 3, 5, 1

4(0)

/ \

10(1) 3(2)

/ \

5(3) 1(4)

The numbers in bracket represent the indices in the array

representation of data.

Applying heapify procedure to index 1:

4(0)

/ \

10(1) 3(2)

/ \

5(3) 1(4)

Applying heapify procedure to index 0:

10(0)

/ \

5(1) 3(2)

/ \

4(3) 1(4)

The heapify procedure calls itself recursively to build heap

in top down manner.

|  |
| --- |
| // C++ program for implementation of Heap Sort  #include <iostream>    using namespace std;    // To heapify a subtree rooted with node i which is  // an index in arr[]. n is size of heap  void heapify(int arr[], int n, int i)  {      int largest = i; // Initialize largest as root      int l = 2 \* i + 1; // left = 2\*i + 1      int r = 2 \* i + 2; // right = 2\*i + 2        // If left child is larger than root      if (l < n && arr[l] > arr[largest])          largest = l;        // If right child is larger than largest so far      if (r < n && arr[r] > arr[largest])          largest = r;        // If largest is not root      if (largest != i) {          swap(arr[i], arr[largest]);            // Recursively heapify the affected sub-tree          heapify(arr, n, largest);      }  }    // main function to do heap sort  void heapSort(int arr[], int n)  {      // Build heap (rearrange array)      for (int i = n / 2 - 1; i >= 0; i--)          heapify(arr, n, i);        // One by one extract an element from heap      for (int i = n - 1; i > 0; i--) {          // Move current root to end          swap(arr[0], arr[i]);            // call max heapify on the reduced heap          heapify(arr, i, 0);      }  }    /\* A utility function to print array of size n \*/  void printArray(int arr[], int n)  {      for (int i = 0; i < n; ++i)          cout << arr[i] << " ";      cout << "\n";  }    // Driver code  int main()  {      int arr[] = { 12, 11, 13, 5, 6, 7 };      int n = sizeof(arr) / sizeof(arr[0]);        heapSort(arr, n);        cout << "Sorted array is \n";      printArray(arr, n);  } |

**Output**

Sorted array is

5 6 7 11 12 13

* Merge sort

Merge Sort is a divide and conquer algorithm. It divides the input array into two halves, calls itself for the two halves, and then merges the two sorted halves. **The merge () function** is used for merging two halves. The merge (arr, l, m, r) is a key process that assumes that arr[l..m] and arr[m+1..r] are sorted and merges the two sorted sub-arrays into one.



|  |
| --- |
| // C++ program for Merge Sort  #include <iostream>  using namespace std;    // Merges two subarrays of arr[].  // First subarray is arr[l..m]  // Second subarray is arr[m+1..r]  void merge(int arr[], int l, int m, int r)  {      int n1 = m - l + 1;      int n2 = r - m;        // Create temp arrays      int L[n1], R[n2];        // Copy data to temp arrays L[] and R[]      for (int i = 0; i < n1; i++)          L[i] = arr[l + i];      for (int j = 0; j < n2; j++)          R[j] = arr[m + 1 + j];        // Merge the temp arrays back into arr[l..r]        // Initial index of first subarray      int i = 0;        // Initial index of second subarray      int j = 0;        // Initial index of merged subarray      int k = l;        while (i < n1 && j < n2) {          if (L[i] <= R[j]) {              arr[k] = L[i];              i++;          }          else {              arr[k] = R[j];              j++;          }          k++;      }        // Copy the remaining elements of      // L[], if there are any      while (i < n1) {          arr[k] = L[i];          i++;          k++;      }        // Copy the remaining elements of      // R[], if there are any      while (j < n2) {          arr[k] = R[j];          j++;          k++;      }  }    // l is for left index and r is  // right index of the sub-array  // of arr to be sorted \*/  void mergeSort(int arr[],int l,int r){      if(l>=r){          return;//returns recursively      }      int m =l+ (r-l)/2;      mergeSort(arr,l,m);      mergeSort(arr,m+1,r);      merge(arr,l,m,r);  }    // UTILITY FUNCTIONS  // Function to print an array  void printArray(int A[], int size)  {      for (int i = 0; i < size; i++)          cout << A[i] << " ";  }    // Driver code  int main()  {      int arr[] = { 12, 11, 13, 5, 6, 7 };      int 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;  }    // This code is contributed by Mayank Tyagi |

**Output**

Given array is

12 11 13 5 6 7

Sorted array is

5 6 7 11 12 13

**Time Complexity:** Sorting arrays on different machines. Merge Sort is a recursive algorithm and time complexity can be expressed as following recurrence relation.   
T(n) = 2T(n/2) + θ(n)

The above recurrence can be solved either using the Recurrence Tree method or the Master method. It falls in case II of Master Method and the solution of the recurrence is θ(nLogn). Time complexity of Merge Sort is  θ(nLogn) in all 3 cases (worst, average and best) as merge sort always divides the array into two halves and takes linear time to merge two halves.

**References: geeks for geeks, Wikipedia.**

Task 3

* Depth first traversal
* Depth First Search is one of the main graph algorithms.
* Depth First Search finds the lexicographical first path in the graph from a source vertex uu to each vertex. Depth First Search will also find the shortest paths in a tree (because there only exists one simple path), but on general graphs this is not the case.
* The algorithm works in O(m+n) time where nn is the number of vertices and mm is the number of edges.

The idea behind DFS is to go as deep into the graph as possible, and backtrack once you are at a vertex without any unvisited adjacent vertices.

It is very easy to describe / implement the algorithm recursively: We start the search at one vertex. After visiting a vertex, we further perform a DFS for each adjacent vertex that we haven't visited before. This way we visit all vertices that are reachable from the starting vertex.

We perform a DFS and classify the encountered edges using the following rules:

If vv is not visited:

* Tree Edge - If vv is visited after uu then edge (u,v)(u,v) is called a tree edge. In other words, if vv is visited for the first time and uu is currently being visited then (u,v)(u,v) is called tree edge. These edges form a DFS tree and hence the name tree edges.

If vv is visited before uu:

* Back edges - If vv is an ancestor of uu, then the edge (u,v)(u,v) is a back edge. vv is an ancestor exactly if we already entered vv, but not exited it yet. Back edges complete a cycle as there is a path from ancestor vv to descendant uu (in the recursion of DFS) and an edge from descendant uu to ancestor vv (back edge), thus a cycle is formed. Cycles can be detected using back edges.
* Forward Edges - If vv is a descendant of uu, then edge (u,v)(u,v) is a forward edge. In other words, if we already visited and exited vv and entry[u]<entry[v]entry[u]<entry[v] then the edge (u,v)(u,v) forms a forward edge.
* Cross Edges: if vv is neither an ancestor or descendant of uu, then edge (u,v)(u,v) is a cross edge. In other words, if we already visited and exited vv and entry[u]>entry[v]entry[u]>entry[v] then (u,v)(u,v) is a cross edge.

Note: Forward edges and cross edges only exist in directed graphs.

**Implementation**

**vector<vector<int>> adj; // graph represented as an adjacency list**

**int n; // number of vertices**

**vector<bool> visited;**

**void dfs(int v) {**

**visited[v] = true;**

**for (int u : adj[v]) {**

**if (!visited[u])**

**dfs(u);**

**}**

**}**

This is the most simple implementation of Depth First Search. As described in the applications it might be useful to also compute the entry and exit times and vertex color. We will color all vertices with the color 0, if we haven't visited them, with the color 1 if we visited them, and with the color 2, if we already exited the vertex.

Here is a generic implementation that additionally computes those:

**vector<vector<int>> adj; // graph represented as an adjacency list**

**int n; // number of vertices**

**vector<int> color;**

**vector<int> time\_in, time\_out;**

**int dfs\_timer = 0;**

**void dfs(int v) {**

**time\_in[v] = dfs\_timer++;**

**color[v] = 1;**

**for (int u : adj[v])**

**if (color[u] == 0)**

**dfs(u);**

**color[v] = 2;**

**time\_out[v] = dfs\_timer++;**

**}**

* Breadth first Traversal
* Breadth first search is one of the basic and essential searching algorithms on graphs.
* As a result of how the algorithm works, the path found by breadth first search to any node is the shortest path to that node, i.e the path that contains the smallest number of edges in unweighted graphs.
* The algorithm works in O(n+m)O(n+m) time, where nn is number of vertices and mm is the number of edges.

## Description of the algorithm

* The algorithm takes as input an unweighted graph and the id of the source vertex ss. The input graph can be directed or undirected, it does not matter to the algorithm.
* The algorithm can be understood as a fire spreading on the graph: at the zeroth step only the source ss is on fire. At each step, the fire burning at each vertex spreads to all of its neighbors. In one iteration of the algorithm, the "ring of fire" is expanded in width by one unit (hence the name of the algorithm).
* More precisely, the algorithm can be stated as follows: Create a queue qq which will contain the vertices to be processed and a Boolean array used[]used[] which indicates for each vertex, if it has been lit (or visited) or not.
* Initially, push the source ss to the queue and set used[s]=trueused[s]=true, and for all other vertices vv set used[v]=falseused[v]=false. Then, loop until the queue is empty and in each iteration, pop a vertex from the front of the queue. Iterate through all the edges going out of this vertex and if some of these edges go to vertices that are not already lit, set them on fire and place them in the queue.
* As a result, when the queue is empty, the "ring of fire" contains all vertices reachable from the source ss, with each vertex reached in the shortest possible way. You can also calculate the lengths of the shortest paths (which just requires maintaining an array of path lengths d[]d[]) as well as save information to restore all of these shortest paths (for this, it is necessary to maintain an array of "parents" p[]p[], which stores for each vertex the vertex from which we reached it).

## Implementation

We write code for the described algorithm in C++.

**vector<vector<int>> adj; // adjacency list representation**

**int n; // number of nodes**

**int s; // source vertex**

**queue<int> q;**

**vector<bool> used(n);**

**vector<int> d(n), p(n);**

**q.push(s);**

**used[s] = true;**

**p[s] = -1;**

**while (!q.empty()) {**

**int v = q.front();**

**q.pop();**

**for (int u : adj[v]) {**

**if (!used[u]) {**

**used[u] = true;**

**q.push(u);**

**d[u] = d[v] + 1;**

**p[u] = v;**

**}**

**}**

**}**

If we have to restore and display the shortest path from the source to some vertex uu, it can be done in the following manner:

**if (!used[u]) {**

**cout << "No path!";**

**} else {**

**vector<int> path;**

**for (int v = u; v != -1; v = p[v])**

**path.push\_back(v);**

**reverse(path.begin(), path.end());**

**cout << "Path: ";**

**for (int v : path)**

**cout << v << " ";**

**}**

**References: cp-algorithms.com**

LEARNING OUTCOMES

* Basic understanding of the memory hierarchy and a basic idea what actually goes on in the background of the computer architecture.
* First important observation of mine from this project was that the method which I used to measure the time complexity falls in RAM machine model which involves calculating the total no of steps.
* Basic understanding of the cache oblivious model and the problems faced with the other model.
* I was unable to explain heap and merge sort question and dfs, bfs question through cache oblivious model I wrote whatever I was able to learn in the given span of time. But it was a great learning experience.