Design & Analysis of Algorithms

Laboratory Instructions & Assignments

By: Tamal Chakraborty

Divide & Conquer: Binary Search, Merge Sort, Quick Sort

WEEK 1

Familiarize with your lab environment

- To start working in the Design & Analysis of Algorithms Lab, you need to login to your student account, and then cd to the algo_lab directory. This is your base working directory.
- In this directory you will find four sub-directories **lib, bin, src** and **inc**
- The bin directory would be used to store your executable, and the lib directory would be used to store the c library that you will create
- The src directory would contain all the .c files (c source code) that you will write in this lab
- The inc directory would contain all the .h (#include) files that you would be writing

How to compile your code

- To compile and build the executable type ./build in the algo_lab directory.
- *./build*
- This will compile your c source code, and if there are no errors, create the executable file named demo in the algo_lab/bin directory.
- As an example the Linear Search algorithm has been implemented, so when you run the executable, you will be able to test the Linear Search algorithm.

How to execute your code

- To execute your code, cd to algo_lab/bin directory
- *cd bin*
- Here you will find a file named demo
- Type ./demo now
- *./demo*
- You will find the following Menu in your screen

```
WELCOME TO THE DESIGN AND ANALYSIS OF ALGORITHMS LAB

                    Main Menu

01. Linear Search

99. Exit


Enter Your Choice:
```

Type 1 here and you will be able to execute and test the Linear Search algorithm

How to add your menu item

- As you would have noticed, currently you have only one menu item 01. Linear Search
- Let's say we want to add another menu item for Binary Search, then we will need to update the following files.
- cd to the src directory, open the MenuItemAdder.c file in editor.
- *vi MenuItemAdder.c*
- This file contains only one function with the following code:

```
void addMenuItems() {
    addMenu("Linear Search", lsearchdemo);
}
```

This is the name of the menu item, which is displayed

This is the name of the function which will be called when the menu item is selected

How to add your menu item

- To add another menu item for Binary Search, add the following line of code in the addMenuItems() function:

```
void addMenuItems() {
    addMenu("Linear Search", lsearchdemo);
    addMenu("Binary Search", bsearchdemo);  <———— This line of code needs to be added
}
```

- Save your code now. This will add a new menu item, named Binary Search in the menu, and when the user chooses this menu item bsearchdemo function is called.
- You might wonder who would write the bsearchdemo function? The answer is of course you.
- This function needs to be placed in the AlgoLab.c file. So this file needs to be edited next.

How to add your menu item

- Open the AlgoLab.c file in editor.
- *vi AlgoLab.c*
- You will see that there is a function named lsearchdemo() in this file, this is the function, which is called when the Linear Search menu item is selected.
- Add another method named bsearchdemo() in this file, this function must return nothing (void) and take no arguments.

```
void bsearchdemo() {

    // your code here

}
```

Keep this function empty for now

Remember the entry you added in MenuItemAdder.c
addMenu("Binary Search", bsearchdemo);
The second argument of addMenu, bsearchdemo, is the name of the function which is called when the menu item is selected. We call it the menu callback function. Here, you are providing the implementation of the menu callback function. So you need to make sure that the 2nd argument of addMenu and the name of this function here are same.

- Save your code.

How to add your menu item

- There is only one more step before we can see our menu item. We need to put the signature of the bsearchdemo() function in the algo_lab/inc/AlgoLab.h file
- cd to inc directory and open the AlgoLab.h file in editor
- *vi AlgoLab.h*
- You will see the following code in this file

```
#ifndef _ALGOLAB_H_
#define _ALGOLAB_H_


void lsearchdemo();


#endif
```

How to add your menu item

- Add the signature of the bsearchdemo() function in this file

```
#ifndef _ALGOLAB_H_
#define _ALGOLAB_H_


void lsearchdemo();
void bsearchdemo();
```

this line needs to be added

```
#endif
```

- Save your change and now you are ready to test your new menu item.
- Compile your code (refer to page 4)
- Test your code (refer to page 5)

How to add your menu item

- Now you would see two menu items instead of one.

```
        WELCOME TO THE DESIGN AND ANALYSIS OF ALGORITHMS LAB

                            Main Menu

01. Linear Search

02. Binary Search

99. Exit


Enter Your Choice:
```

How to implement your algorithm

- You might have noticed that we haven’t discussed about implementing the binary search algorithm so far. So let us see how to implement it.
- cd to the algo_lab/src/algorithms directory
- You will find a file named LinearSearch.c, this file implements the LinearSearch algorithm
- We will now create a new file named BinarySearch.c, which will contain the implementation of the Binary Search algorithm.

```
int binarySearch(int a[], int l, int r, int key) {                    // binary search for key in a[l … r]
    if (key < a[l] || key > a[r]) return -1;                          // key is not found
    int m = (l + r) / 2;                                              // m is the mid-point of a
    if (key < a[m]) return binarySearch(a, l, m – 1, key);            // (binary) search in the left half
    else if (key > a[m]) return binarySearch(a, m + 1, r, key);       // (binary) search in the right half
    else return m;                                                    // found the key
}
```

Update the file information

- Before you save your code you need to update the file information, at the beginning of your .c file.

```
//---------------------------------------------------------------------------------------------
//
// Course              :        Design & Analysis of Algorithm Lab
// Code                :        CS591
// Assignment Number   :        <Write the assignment number, e.g. 1.1>
// Author(s)           :        <Write your name>
// Roll Number(s)      :        <Write your roll number>
// Date                :        <Write the date of implementation>
//
//---------------------------------------------------------------------------------------------
```

- Save your code now.

Create the corresponding header file

- cd to the algo_lab/inc/algorithms directory
- You will find a file named LinearSearch.h, this file defines the LinearSearch algorithm
- We will now create a new file named BinarySearch.h, which will contain the definition of the Binary Search algorithm.

```
#ifndef _BINARYSEARCH_H_
#define _BINARYSEARCH_H_


int binarySearch(int a[], int l, int r, int key);


#endif
```

Update the Makefile

- Of course now, we would like to compile our newly added code. Follow the steps below before you go for compilation.
- cd to the algo_lab/src/algorithms directory, you will find a file named Makefile there. Open this file in editor.
- *vi Makefile*
- Find the line SRC = LinearSearch.c in this file.
- Edit the file to add the entry for BinarySearch.c in the following manner:

```
SRC =     LinearSearch.c \          ←add a "slash" in this line
          BinarySearch.c            ←add the entry for BinarySearch.c without a slash
```

- Save your changes.

Call your algorithm from the menu callback

- Now, we are ready to add code to our bsearchdemo method (remember that we had left it empty). cd to the /home/student/algo_lab/src directory and edit the AlgoLab.c file
- Include the BinarySearch.h file
- *#include <algorithms/BinarySearch.h>*
- In the bsearchdemo() function add the code to read an array of elements and the key to be searched from the user (refer to the lsearchdemo function as example).
- Now call the binarySearch function that you implemented in BinarySearch.c from bsearchdemo() (just like the find function is called from the lsearchdemo() function).

```
void bsearchdemo() {
    // read an array a of n elements and a key
    int found = binarySearch(a, 0, n – 1, key);
    // print whether the key was found or not
}
```

- Save your changes.
- Compile & test your code (refer to page 4, 5)

Where is my main() function?

- Cheer up! You **DO NOT** need to write the C main() function in this lab.
- The intention of this lab is to make you focus on writing algorithms, and not to teach C.
- We have taken care to write the main function for you, so that you do not have to bother about such small matters.
- So, do what you do best, which is to implement (in your humble opinion) best-in-class, state-of-the-art, cutting-edge algorithms.
- Just follow the steps in the next page each time you implement a new algorithm and you will rock!!!

Summary

- **To add a new algorithm and a corresponding menu item:**

1. Add an entry for the new menu item in algo_lab/src/MenuItemAdder.c (page 6, 7)
2. Add the corresponding menu callback function (e.g. myalgodemo) in algo_lab/src/AlgoLab.c (page 8)
3. Add the definition of the menu callback function in algo_lab/inc/AlgoLab.h (page 9, 10)
4. Create a new .c file (e.g. MyAlgo.c) in algo_lab/src/algorithms directory. Implement your algorithm in this file, as one or more c functions and don't forget to update the file information. (page 12, 13)
5. Add the definition of the functions implemented in previous step in a new .h file (e.g. MyAlgo.h) in algo_lab/inc/algorithms directory (page 14)
6. Add the entry for your MyAlgo.c file in the SRC = section of the Makefile in algo_lab/src/algorithms directory. You should add your entry in the end (last line) of the section, and remember to add a slash (\) in the previous line. (page 15)
7. Implement the menu callback function (myalgodemo). You should read the inputs for your algorithm from the user here and then call the function in MyAlgo.c which implements your algorithm (page 16)
8. Compile and test your code (page 4, 5).

Assignments

1.1 Implement the quick sort algorithm

```
quickSort(A, l, r)
    if l < r
        p = partition(A, l, r)
        quickSort(A, l, p - 1)
        quickSort(A, p + 1, r)

partition(A, l, r)
    pivot = A[r]
    i = l – 1
    for j = l to r – 1
        if A[j] ≤ pivot
            i = i + 1
            exchange A[i], A[j]
    exchange A[i + 1], A[r]
    return i + 1
```

1.2 Implement the merge sort algorithm

```
mergeSort(A, l, r)
    if l < r
        m = (l + r) / 2
        mergeSort(A, l, m)
        mergeSort(A, m + 1, r)
        merge(A, l, m, r)

merge(A, l, m, r)
    i = l, j = m + 1, B[r – l + 1]
    for k = 0 to r - l
        if        j > r          B[k] = A[i++]
        else if   i > m          B[k] = A[j++]
        else if   A[i] < A[j]    B[k] = A[i++]
        else                     B[k] = A[j++]
    for k = 0 to r - l      A[l + k] = B[k]
```

Dynamic Programming: Matrix Chain Multiplication

WEEK 2

Matrix Chain Multiplication

- Let us suppose that we are given a chain of n matrices A_1, A_2, …, A_n and we have to calculate the matrix product P, such that P = $A_1A_2…A_n$
- We know that multiplication of 2 (n x n) matrices takes $\Theta(n^3)$ time.
- Thus if two matrices A & B of dimensions (p x q) and (q x r) would require p.q.r scalar multiplications.
- Our intension is to multiply n matrices A_1, A_2, …, A_n to find their product P, such that the number of scalar multiplications are minimum.
- Suppose A, B & C are 3 matrices, then their product P can be calculated as P = (A.B).C = A.(B.C)
- That is, multiplying A with B first and then multiplying their product with C gives the same result as multiplying B with C first and then multiplying their product with A.
- But the way we parenthesize this operation has a big impact on the number of scalar multiplications required to compute the product.

Calculating the number of scalar multiplications

- Suppose the dimensions of A, B & C are given as (1 x 2), (2 x 5) & (5 x 1) respectively.
- If we find their product as (A.B).C then the number of scalar multiplications needed are 1.2.5 + 1.5.1 = 15
- If we find their product as A.(B.C) then the number of scalar multiplications needed are 2.5.1 + 1.2.1 = 12
- Thus we observe that parenthesizing the matrices as A.(B.C) evaluates their product with less scalar multiplications.
- The matrix chain multiplication problem can be restated as follows:
- Given n matrices A_1, A_2, …, A_n of dimensions (p_0 x p_1), (p_1 x p_2), …, (p_{n-1} x p_n) respectively, we have to fully parenthesize the product $P = A_1A_2\ldots A_n$ in such a way that it minimizes the number of scalar multiplications.

The structure of optimal solution

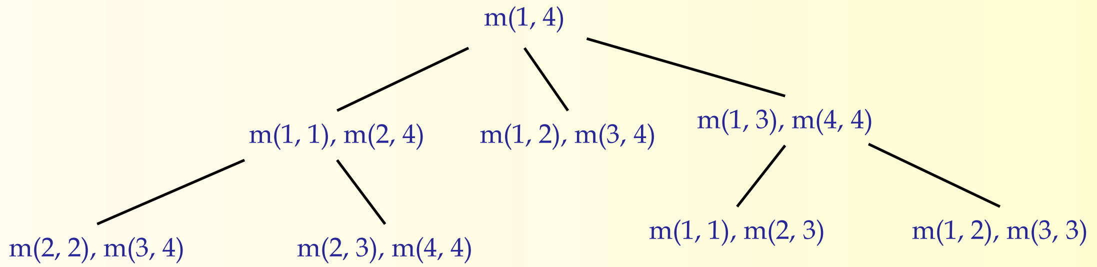
- Let $A_{i,j}$ denote the matrix product $A_i A_{i+1} \ldots A_j$
- Let us suppose that there is an integer k, such that if we parenthesize the product $A_{i,j}$ as $(A_i A_{i+1} \ldots A_k)$. $(A_{k+1} \ldots A_j)$ the number of scalar multiplications are minimum.
- Clearly to compute $A_{i,j}$ one has to compute the product matrices $A_{i,k} = (A_i A_{i+1} \ldots A_k)$ and $A_{k+1,j} = (A_{k+1} \ldots A_j)$ and then multiply them.
- Since the dimension of A_i is $(p_{i-1} \text{ x } p_i)$ the dimensions of $A_{i,k}$ & $A_{k+1,j}$ would be $(p_{i-1} \text{ x } p_k)$ and $(p_k \text{ x } p_j)$ respectively.
- Hence multiplication of $A_{i,k}$ & $A_{k+1,j}$ would require $p_{i-1}p_kp_j$ scalar multiplications.
- Let m(i, j) be the optimal cost for multiplying matrices $A_i A_{i+1} \ldots A_j$ i.e. computing $A_{i,j}$
- Since computation of $A_{i,j}$ requires computation of $A_{i,k}$ & $A_{k+1,j}$ and then $p_{i-1}p_kp_j$ scalar multiplications for evaluating their product, we have:
- $m(i, j) = m(i, k) + m(k + 1, j) + p_{i-1}p_kp_j$
- Clearly $m(i, i) = 0$ & $m(i, i + 1) = p_{i-1}p_ip_{i+1}$
- The above recurrence relation assumes that we know the value of k, which we do not, hence we need to check it for all possible values of k, where $i \le k < j$

A recursive solution

- The matrix chain multiplication problem can be solved by the following recurrence:

$$m(i, j) = \begin{cases} 0 & \text{if } j = i \\ p_{i-1}p_ip_{i+1} & \text{if } j = i + 1 \\ \min\limits_{i \le k < j}\{ m(i, k) + m(k + 1, j) + p_{i-1}p_kp_j \} & \text{if } j > i + 1 \end{cases}$$

- For example, say we want to multiply 4 matrices: (1 x 2), (2 x 3), (3 x4), (4 x 1). Our recursive approach will lead to the following recursion tree:



- As we can see, this is a top down approach, i.e. that is we start with a call to m(1, 4) and then go down the tree until we find a node, whose value is known (e.g. m(1, 1))
- The problem with this approach is that the same values are evaluated again and again, e.g. m(1, 2) is computed twice above, as is m(2, 3) and m(3, 4)

Dynamic Programming: Bottom-up Approach

- The idea behind dynamic programming is to build the solution in a bottom-up manner, and re-use the previously computed values in order to compute a new one.
- Looking at our example we need to find m(1, 4), i.e. a product of a matrix-chain of length 4. We start from the product of the matrix-chain of length 0, i.e. m(1, 1), m(2, 2), m(2, 3), m(4, 4) and the product of the matrix-chain of length 1, i.e. m(1, 2), m(2, 3), m(3, 4) whose values are already known.
- Let us build a table of size n x n (4 x 4 in our example), where the entry (i, j) indicates the number of multiplications required to compute the product of a chain of matrices from A_i to A_j. For example the entry (2, 4) in the table would give us the number of scalar multiplications to compute the product of A_2 to A_4
- We are of course interested in the entries on and above the diagonal.
- So, filling the table with the already known values:

0	6		
	0	24	
		0	12
			0

Matrix chain Product of: A_{1x2} B_{2x3} C_{3x4} & D_{4x1}

m(1, 1) = m(2, 2) = m(3, 3) = m(4, 4) = 0

m(1, 2) = 1.2.3 = 6

m(2, 3) = 2.3.4 = 24

m(3, 4) = 3.4.1 = 12

Filling the table

- The next step is to compute the number of scalar multiplications needed for evaluating products of matrix chains of length 2, i.e. m(1, 3) and m(2, 4) in our example.

$$m(1,3) = \min\begin{pmatrix} \mathrm{m}(1,1) + \mathrm{m}(2,3) + 1.2.4 \\ \mathrm{m}(1,2) + m(3,3) + 1.3.4 \end{pmatrix} = \min\begin{pmatrix} 0 + 24 + 8 \\ 6 + 0 + 12 \end{pmatrix} = 18$$

$$m(2,4) = \min\begin{pmatrix} \mathrm{m}(2,2) + \mathrm{m}(3,4) + 2.3.1 \\ \mathrm{m}(2,3) + m(4,4) + 2.4.1 \end{pmatrix} = \min\begin{pmatrix} 0 + 12 + 6 \\ 24 + 0 + 8 \end{pmatrix} = 18$$

- Filling these entries in the table gives us:

0	6	18	
	0	24	18
		0	12
			0

Computing m(1, 4)

- Finally, the value of m(1, 4) is computed as:

$$m(1,4) = \min\begin{pmatrix} m(1,1) + m(2,4) + 1.2.1 \\ m(1,2) + m(3,4) + 1.3.1 \\ m(1,3) + m(4,4) + 1.4.1 \end{pmatrix} = \min\begin{pmatrix} 0 + 18 + 2 \\ 6 + 12 + 3 \\ 18 + 0 + 4 \end{pmatrix} = 20$$

- So, we fill the final and desired entry in the table:

0	6	18	20
	0	24	18
		0	12
			0

- Thus the minimum number of scalar multiplications to compute our matrix product of the chain of 4 matrices is 20.

Assignment 2.1

- Implement the dynamic programming algorithm for matrix chain multiplications

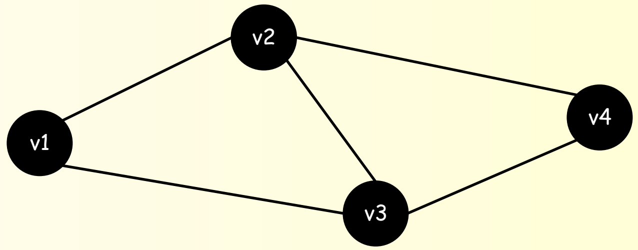
```
mcm() // computes the least number of scalar multiplications to multiply a chain of matrices
    for (i = 1 to n)
            m[i][i] = 0;
            if (i ≠ n) m[i][i + 1] = p[i-1]*p[i]*p[i+1];
    for (l = 2 to n - 1)
            for (i = 1 to n - l )
                    j = i + l;
                    m[i][j] = ∞;
                    for (k = i to j - 1)
                            q = m[i][k] + m[k + 1][j] + p[i-1]*p[k]*p[j];
                            if (q < m[i][j])
                                    m[i][j] = q;
                                    s[i][j] = k;
    return m[1][n];

mcorder(s[][n], i, j) // prints the matrix chain order, initial call should be mcorder(s, 1, n)
    if (i == j) print("A_i");
    else if (j == i + 1) print("(A_i.A_j)");
    else
            print("(");
            mcorder(s, i, s[i][j]);
            mcorder(s, s[i][j] + 1, j);
            print(")");
```

Graph & Graph Algorithms

WEEK 3

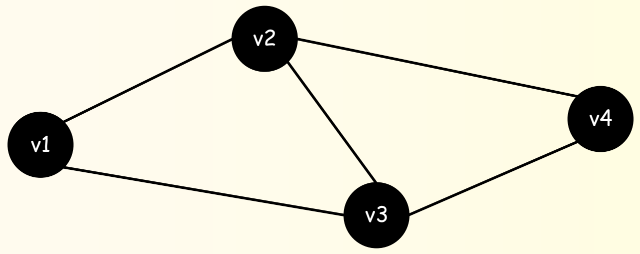
Definitions & Terminology

- A graph is a practical representation of the relationship between some objects, represented in a 2D diagram.
- The set of objects [V = {v1, v2, …}] are called vertices and are represented in 2D by points.
- The set of relationships [E = {e1, e2, …}] between the objects are called edges and are represented on 2D by lines.
- Every edge e_k, is associated with a pair of vertices v_i & v_j, which are called the end-vertices of e_k.



Adjacency Matrix representation of a Graph

- A graph G = (V, E) Can be represented by an Adjacency matrix A = [A_{ij}] according to the following rules:
- $A_{ij} = 1$, if there is an edge between vertex i, j
- $A_{ij} = 0$, if there is no edge between vertex i, j
- For example the graph below can be represented by its adjacency matrix:

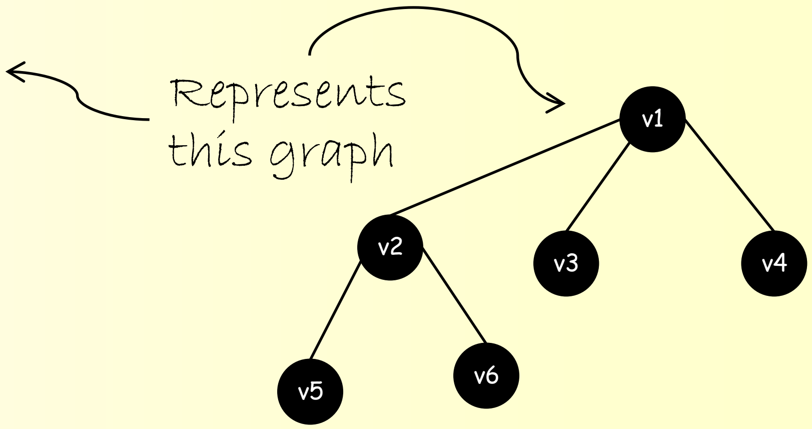


	v1	v2	v3	v4
v1	0	1	1	0
v2	1	0	1	1
v3	1	1	0	1
v4	0	1	1	0

Assignment 3.1: Reading Graph from a file

- Suppose you are given with a file with n lines, each line contains exactly n integers separated by TAB. For example:

```
0	1	1	1	0	0
1	0	0	0	1	1
1	0	0	0	0	0
1	0	0	0	0	0
0	1	0	0	0	0
0	1	0	0	0	0
```



- The file above has 6 lines and each line contains 6 numbers, separated by TAB.
- Read the file, and instantiate a two dimensional array of size n x n (6 x 6) in this example.

Guideline

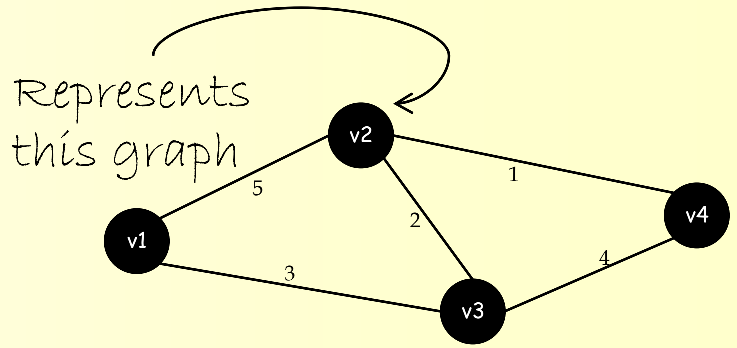
- We will create a function, say readGraph(), which takes two parameters a FILE pointer fp and a 2D array G[][n], n being the number of vertices in the graph. The function will read the file pointed to by fp, and fill-up entries of G.
- The first task in hand is to read a file one line at a time. The getline() function in C can be used to achieve this.
- Then we need to tokenize the line (which contains n integers separated by TAB). The strsep() function can help us achieve it.
- Finally, each token, as read from the file, is received as a pointer to characters, we need to convert it to int using the atoi() function.

```
void readGraph(FILE* fp, int G[][n]) {
    char* line = NULL; size_t size; int i = 0;
    while (getline(&line, &size, fp) != -1) { // read a line from the file
        char* edge; int j = 0;
        while ((edge = strsep(&line, "\t")) != NULL) G[i][j++] = atoi(edge); // read an edge
        i++;
    }
}
```

Assignment 3.2: Reading Edges from the Adjacency Matrix of a Graph

- Suppose you are given a weighted adjacency matrix G of a graph. For example:

$$\begin{matrix} 0 & 5 & 3 & \infty \\ 5 & 0 & 2 & 1 \\ 3 & 2 & 0 & 4 \\ \infty & 1 & 4 & 0 \end{matrix}$$



- The adjacency matrix is given to you as a 2D array G[n][n], and you want to parse it and create an array of structures, which represent the edges of the graph.
- You need to avoid duplicate entries for undirected graphs. For example, in this graph (1, 2) and (2, 1) represent the same edge. So you should report only one edge instead of two.
- Use the value 999999 to represent ∞.

Guideline

- The first task is to create the struct edge. We will use the following definition:

```
typedef struct Edge {
    int u;      // represents the first end-vertex of the edge
    int v;      // represents the second end-vertex of the edge
    int cost;   // represents the cost/weight of the edge
} edge;
```

- Now, let us create a function getEdges(), which takes two parameters, the (weighted) adjacency matrix of the graph G, and an array edges of struct edge (which is empty). The function will parse G and fill-up the entries in edges. Here is the pseudo-code.

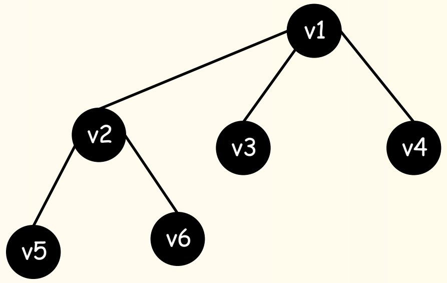
```
getEdges(G[][n], edges[])
    k = 0;   // number of edges in the graph
    for i = 0 to n - 1
        for j = 0 to n - 1
            if $G[i][j] \neq 0$ AND $G[i][j] \neq \infty$          // check if an edge exists
                if j > i OR $G[i][j] \neq G[j][i]$              // avoid duplication
                    edge e;
                    e.u = i; e.v = j; e.cost = G[i][j];   // fill the details of the edge
                    edges[k++] = e;
    return k; // how many edges were found?
```

Traversing a Graph

- One of the most fundamental graph problem is to traverse a graph.
- We have to start from one of the vertices, and then mark each vertex when we visit it. For each vertex we maintain three flags:
  - **Unvisited**
  - **Visited but unexplored**
  - **Visited and completely explored**
- The order in which vertices are explored depends upon the kind of data structure used to store intermediate vertices.
  - Queue (FIFO)
  - Stack (LIFO)

Breadth First Search (BFS)

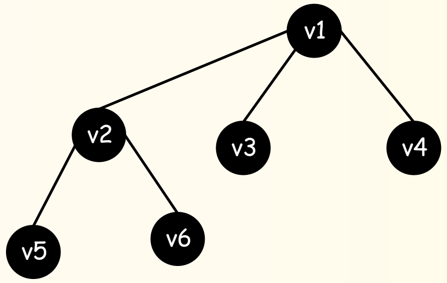
- In this technique we start from a given vertex v and then mark it as visited (but not completely explored).
- A vertex is said to be explored when all the vertices adjacent to it are visited.
- All vertices adjacent to v are visited next, these are new unexplored vertices.
- The vertex v is now completely explored.
- The newly visited vertices which are not completely explored are put at the end of a queue.
- The first vertex of this queue is explored next.
- Exploration continues until no unexplored vertices are left.



For example: BFS(v1) of this graph yields v1,v2,v3,v4,v5,v6

Depth First Search (DFS)

- In this technique we start from a given vertex v and then mark it as visited.
- A vertex is said to be explored when all the vertices adjacent to it are visited.
- An vertex adjacent to v are put at the top of a stack next.
- The top vertex of this stack is explored next.
- Exploration continues until no unexplored vertices are left.
- The search process can be described recursively.



For example: DFS(v1) of this graph yields v1,v2,v5,v6,v3,v4

Assignments

3.3 Implement the BFS algorithm

```
BFS(v) {
    u = v;
    visited[v] = true;
    // visited is an array of vertices visited so far,
    // all elements are 0 initially
    do {
        for all vertices w adjacent to u do {
            if (!visited[w]) {
                add w to q;
                // q is the queue of
                // unexplored vertices
                visited[w] = true;
            }
        }
        if (q empty) return;
        u = extract front element of q;
    } while (true);
}
```

3.4 Implement the DFS algorithm

```
DFS(v)
{
    visited[v] = true;
    // visited is an array of vertices
    // visited so far, all elements are 0 initially
    for all vertices w adjacent to v do
    {
        if (!visited[w])
            DFS(w);
    }
}
```

Putting them all together

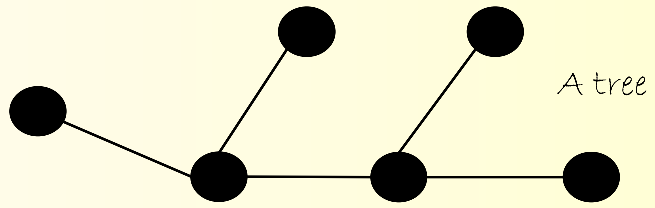
- Create a file named Graph.h in algo_lab/inc/algorithms directory.
- Put the definition of struct edge, n (nr. of vertices) and infinity in that file
- Declare the readGraph, getEdges, BFS & DFS functions there.
- Create a file named Graph.c in algo_lab/src/algorithms directory.
- Implement the readGraph, getEdges, BFS & DFS function there.
- Add two menu items for BFS & DFS.
- When you implement the menu callbacks for BFS & DFS, you must read the graph from a file given by the user. Use the readGraph method to achieve this.

Heap and Priority Queue

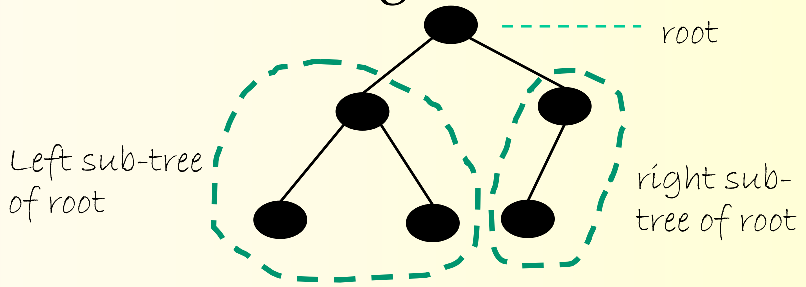
WEEK 4

Trees & Binary Trees

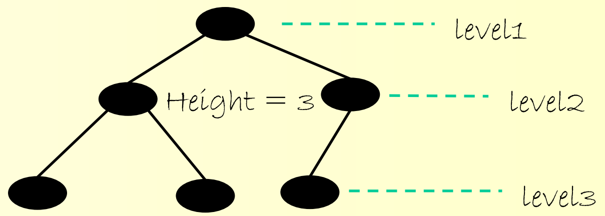
- A tree is a connected graph without any circuits (i.e. a minimally connected graph).



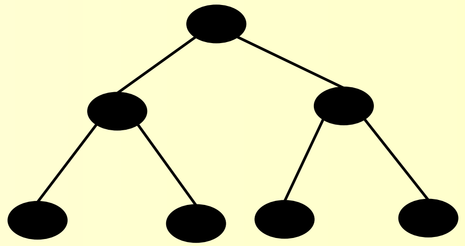
- A binary tree t is a finite (may be empty) set of vertices.
- A non-empty binary tree has a root vertex.
- The remaining vertices (if any) can be partitioned into two binary trees, which are called the left and right sub-trees of t.

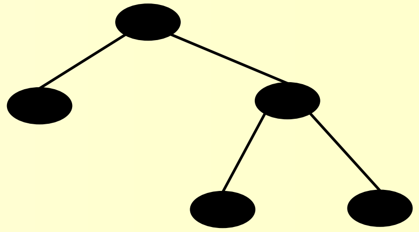


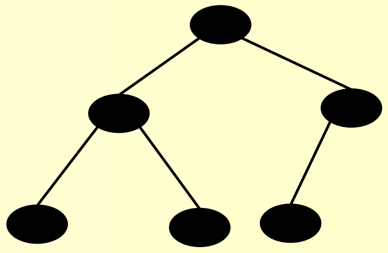
Properties of Binary Tree



- Level
  - By definition root is at level 1
  - Children of root are at level 2
  - Their children are at level 3 and so on…
- Height
  - The height or depth of a binary tree is the number of levels in it
- A binary tree of height h (h ≥ 0) has at least h and at most $2^h - 1$ vertices in it.
- A binary tree of height h that has exactly $2^h - 1$ vertices is called a **full binary tree**.
- If every non-leaf node in a binary tree has non-empty left and right sub-trees, the tree is called a **strictly binary tree**.
- A binary tree is called a **complete binary tree**, if all its levels, except possibly the last have the maximum number of possible vertices, and if all vertices at the last level appear as far left as possible.

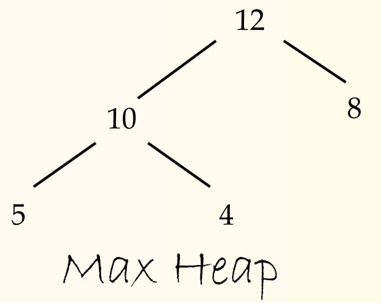

Full binary tree

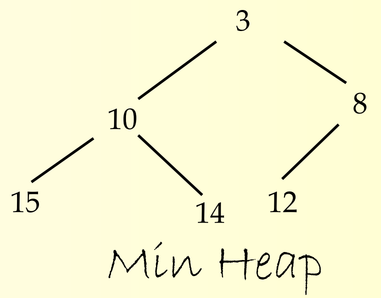

Strictly binary tree


Complete binary tree

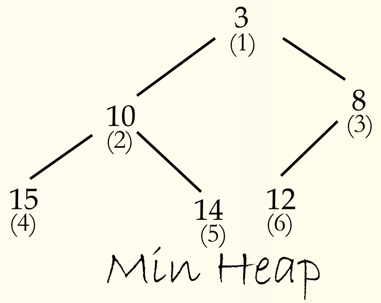
Heap

- A max (min) heap is a complete binary tree such that the root is greater (less) than its children.





- A max (min) heap can be represented by an array, as shown:



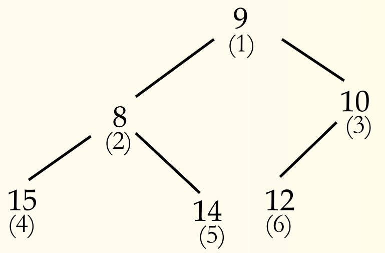
1	**2**	**3**	**4**	**5**	**6**
3	10	8	15	14	12

Array representing the min heap

- Root is at index 1 of the array, left child of root is at index 2, right child of root is at index 3. In general, if index of an element is i, its left child is at 2 * i and right child is at 2 * i + 1

Heapify

- Heapify is an important method for manipulating heaps. In our examples we have illustrated with min heaps, similar method can be employed for max heaps as well.
- The heapify method takes two inputs, an array a[] and an index i of the array. It is assumed that the left and right sub-trees of the binary tree rooted at i are min-heaps, but a[i] may be greater than its children.

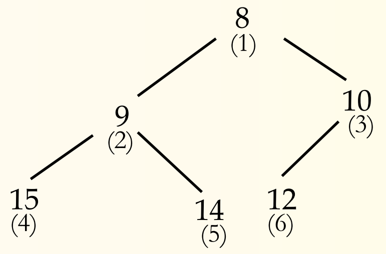


1	2	3	4	5	6
9	8	10	15	14	12

The array a[ ]

e.g. heapify (a, 1)
Notice that a[1] > a[2]
But trees rooted at index 2 and 3 are min-heaps

- We find the min of a[i] and its left and right child. Then a[i] and a[min] are interchanged.

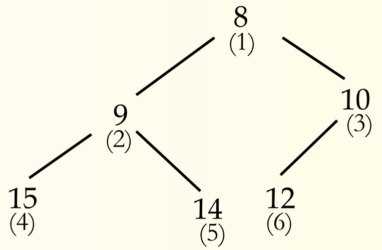


1	2	3	4	5	6
8	9	10	15	14	12

Elements a[1] and a[2] have been interchanged

Heapify

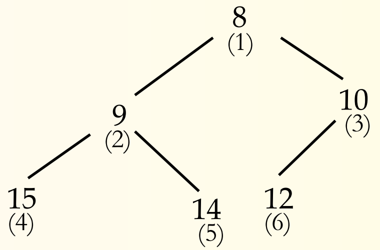
- We now recursively apply the same process to the sub-tree in which the interchange took place.



1	2	3	4	5	6
8	9	10	15	14	12

Now recursively call heapify(a, 2)
But since the sub-tree at index 2 is already a min-heap no further change is necessary

- Thus, we can re-arrange the elements of array a[] by the heapify(a[], i) method, such that the sub-tree rooted at index i becomes a min-heap.



The resultant min-heap rooted at 1

Priority Queue

- A priority queue is a data structure for maintaining a set S of elements, each with an associated value called a key. A min-priority queue supports the following operations.
  - insert(x) // inserts element x in the priority queue
  - extractMin() // removes & returns the element with smallest key
  - decreaseKey(x, k) // decreases the value of element x's key to k
  - findPQ() // search for the element in the priority queue
- A priority queue can be implemented by a heap.

Implementing a Priority Queue of structures

- Say, we want to create a Priority Queue of nodes, where each node is defined as a structure, containing an index and a key value.

```
typedef struct Node {
    int index;
    int key;
} node;
```

- We define the Priority Queue as a structure as well, containing an array (heap) of nodes and a size attribute.

```
typedef struct priorityqueue {
    node* heap[50];
    int size;
} PriorityQueue;
```

Priority Queue: extractMin Operation

- The extractMin() operation removes the first element of the heap, puts the last (nth) element in the heap in the first position, then calls heapify to re-create a heap of n – 1 elements.

```
heapify(node* a[], int i, int n) {
    int l = 2 * i, r = 2 * i + 1, min = i;
    if (l ≤ n AND a[l]→key < a[min]→key)
        min = l;
    if (r ≤ n AND a[r]→key < a[min]→key)
        min = r;
    if (min ≠ i) {
        exchange(a[i], a[min]);
        heapify(a, min, n);
    }
}
```

```
extractMin(PriorityQueue*  pq) {
    if (pq→size == 0) return 0;
    node* n = pq→heap[1];
    pq→heap[1] = pq→heap[pq→size];
    pq→size--;
    heapify(pq→heap, 1, pq→size);
    return n;
}
```

Priority Queue: decreaseKey operation

- The decreaseKey() operation decreases the key of the element i to k. The priority-queue element whose key is to be increased is identified by an index i into the array. The procedure first updates the key of element heap[i] to its new value. Because decreasing the key of heap[i] may violate the min-heap property, the procedure then traverses a path from this node toward the root to find a proper place for the newly decreased key.

```
decreaseKey(PriorityQueue* pq, int i, int val) {
    pq→heap[i]→key = val;
    while (i > 1 AND pq→heap[parent(i)]→key > pq→heap[i]→key) {
            exchange(pq→heap[i], pq→heap[parent(i)]);
            i = parent(i);
    }
}
```

Priority Queue: insert & findPQ Operations

- The insert operation takes as an input the key i of the new element to be inserted into the min-heap. The procedure first expands the min-heap by adding to the tree a new leaf. Then it calls decreaseKey() to set the key of this new node to its correct value and maintain the min-heap property.

```
insert(PriorityQueue* pq, int i, int val) {
    pq→size++;
    node* n = (node*)malloc(sizeof(node));
    n→key = i;
    pq→heap[pq→size] = n;
    decreaseKey(pq, pq→size, val);
}
```

- Since we are implementing the heap as an array, the findPQ operation can be implemented as linear search in the array.

```
int findPQ(PriorityQueue* pq, int ind) { // find the element whose index is ind
    for (i = 1 to pq→size)
            if (pq→heap[i]→index == ind) return i;        // found
    return 0;                                             // not found
}
```

4.1 Assignments

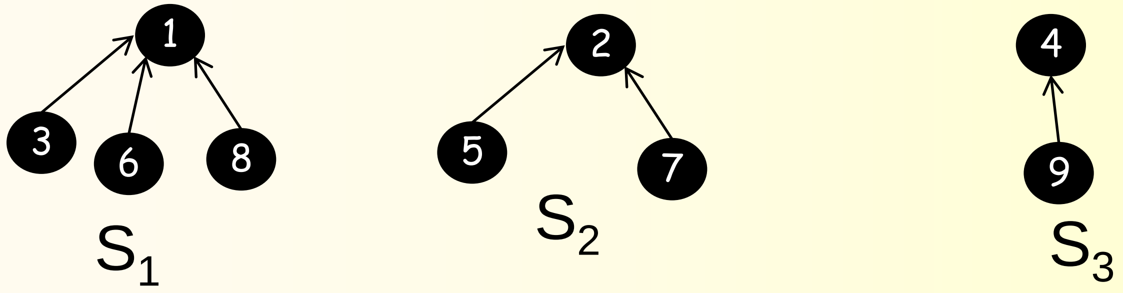
- Implement a PrioirityQueue.h file in algo_lab/inc/algorithms directory. Put the definitions of struct node, struct PriorityQueue and functions extractMin(), insert(), decreaseKey(), findPQ() there.
- Implement a PriorityQueue.c file in algo_lab/src/algorithms directory. Put the implementations of extractMin(), heapify(), insert(), decreaseKey(), findPQ() there.
- Implement a new menu item for demonstrating Priority Queue.
- Test your implementation by accepting a set of elements from the user and then printing their sorted order by first constructing a priority queue with those elements and then repeatedly calling the extractMin() operation.

Disjoint Sets & Job Sequencing

WEEK 5

Disjoint Sets

- Say, we have disjoint sets of numbers 1, 2, 3, …, n
- i.e. if S_i & S_j ($i \neq j$) are two disjoint sets, then there is no element that is in both S_i & S_j
- For example, say there are three sets $S_1 = \{1, 3, 6, 8\}$, $S_2 = \{2, 5, 7\}$, $S_3 = \{4, 9\}$
- We can represent the sets as trees, with the exception that instead of parent nodes pointing to children nodes, the children nodes point to the parent node.

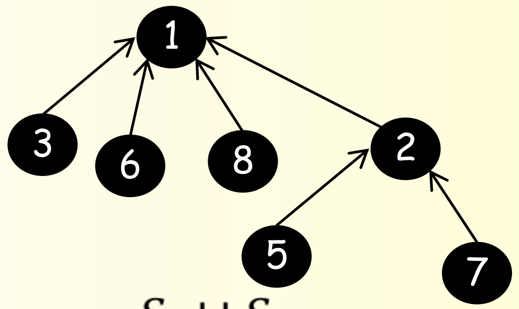


Disjoint Set Operations

- Union
  - If S_i and S_j be two disjoint sets then their union S_i U S_j are all elements x such that x is in S_i or S_j. So S_1 U S_2 in our example = {1, 3, 6, 8, 2, 5, 7}.
- Find
  - Given an element x, find(x) operation tells us to which set x belongs. So 3 is in S_1 and 9 is in S_3

Union & Find

- To obtain union of S_i and S_j we simply make one of the trees a sub-tree of the other.
- Thus to obtain union of two disjoint sets we have to set the parent field of one of the roots to the other root.



$S_1 \cup S_2$

- For simplicity let us ignore the set names and represent the sets by the roots of the trees representing them.
- The operation Find(i) now becomes determining the root of tree containing element i.

Array representation of disjoint sets

- Since the set elements are numbered 1 … n, we can represent the tree nodes using an array p[1 … n].
- The i^{th} element of this array represents the tree nodes that contain element i.
- This array element gives the parent of the corresponding node.
- For root nodes the value of parent is set to -1.
- Thus, we can represent the sets $S_1 = \{1, 3, 6, 8\}$, $S_2 = \{2, 5, 7\}$, $S_3 = \{4, 9\}$ by the following array:

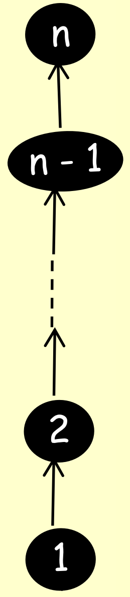
i	1	2	3	4	5	6	7	8	9
p	-1	-1	1	-1	2	1	2	1	4

Algorithms for union & find

```
Union (i, j) { // union of two trees with roots at i & j
        p[i] = j;
}

Find(i) {
        while (p[i] ≥ 0) { i = p[i]; }
        return i;
}
```

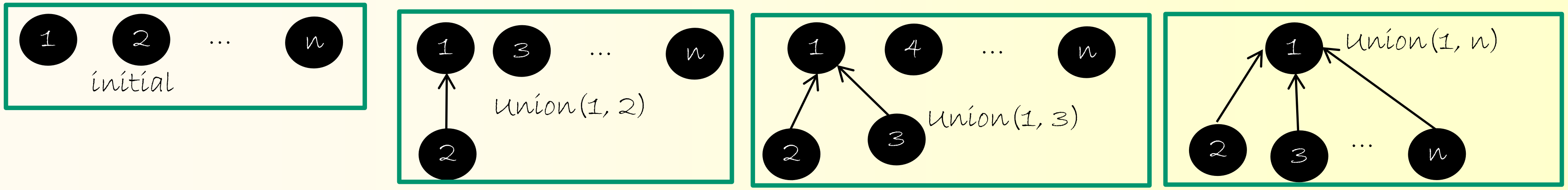
Degenerate Tree

- Say we have n sets, each having one element {1}, {2}, …, {n}, i.e. a forest of n one-element trees.
- Now, if we perform the following sequence of union operations:
  - Union(1, 2), Union(2, 3), …, Union(n – 1, n)
- We will get a degenerate tree as shown in the picture.
- Since the union algorithm runs in constant time, the n – 1 unions will be processed in O(n) time.
- But now, if we perform the following sequence of find operations on the degenerate tree:
  - Find(1), Find(2), …, Find(n)
- Time needed to perform a find at level i of the tree would be O(i)
- Hence the total time needed for all the finds is: $\sum_{i=1}^{n} i = O(n^2)$



Weighting rule for union

- We can improve the performance of our Union and Find algorithms by avoiding the creation of degenerate trees. To accomplish this we will use the weighting rule for Union(i, j)
- Weighting rule for Union(i, j)
  - *If the number of nodes in the tree with root i is less than the number of nodes in the tree with root j, then make j the parent of i, else make i the parent of j.*
- When we use the weighting rule of union to perform the set of unions given before, we obtain the following:



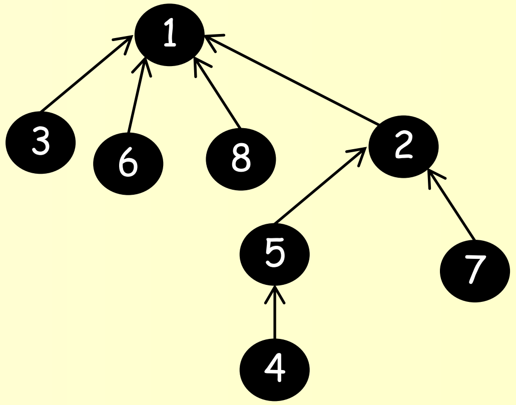
Implementation of weighting rule

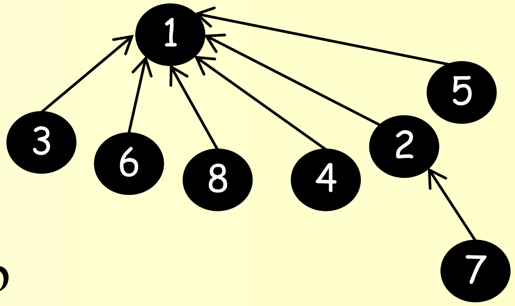
- To implement the weighting the weighting rule we need to know how many nodes are present in a tree.
- To achieve this we maintain a count field in the root of every tree.
- If r is a root node the count(r) equals number of nodes in that tree.
- Since all nodes other than roots have a positive number in the p field, we maintain the count in the p fields of the root as a negative number.
- Thus, we can represent the sets S_1 = {1, 3, 6, 8}, S_2 = {2, 5, 7}, S_3 = {4, 9} by the following array:

i	1	2	3	4	5	6	7	8	9
p	-4	-3	1	-2	2	1	2	1	4

Path Compression

- Further improvements are possible by modifying the find(e) procedure, so as to compress the path from element e to the root.
- For example, let us consider the performance of the following 4 finds in the given tree:
  - Find(4), Find(4), Find(4), Find(4)
- Each Find(4) requires going up three parent link fields, resulting in 12 moves for the 4 finds.
- During the first Find(4) nodes 5, 2 are determined to be in the 4 to the root 1.
- Now if we change the parent fields of 4 and 5 to 1 (2 already has 1 as parent), we get the resulting tree:
- Each of the remaining finds now requires going up only 1 parent link field.
- The total cost is now only 7 moves.





Weighted Union & Path Compression Algorithms

```
weightedUnion(u, v) {
        if (p[u] > p[v]) { p[v] += p[u]; p[u] = v; }
        else { p[u] += p[v]; p[v] = u; }
}

compressedFind(v) {
        if (p[v] < 0) return v;
        p[v] = compressedFind(p[v]);
        return p[v];
}
```

Assignment

5.1 Implement the DisjointSet.C file in algo_lab/src/algorithms directory with the following methods.

```
initDisjointSet(int n);                  // initialize the array p with every element as -1
void weightedUnion(int u, int v);        // weighted union of two disjoint sets rooted at u & v
int compressedFind(int v);               // perform compressedFind  for v
```

- Add the declaration of these methods in algo_lab/inc/algorithms/DisjointSet.h file
- Test your implementation to verify that it works correctly.

Application: Job Sequencing with Deadlines

- We are given a set of n jobs
- Associated with each job is an integer deadline $d_i > 0$ and a profit $p_i > 0$
- For any job the profit is earned if and only if it is completed within deadline
- Basic Assumptions:
  1. Only one job can be processed at a time
  2. Once the job gets started it must be allowed to perform till completed
  3. No job can miss the deadline, i.e. it must complete within the deadline
  4. To complete each job it takes one unit of time
- Our problem is to maximize the profit such that maximum jobs are completed within deadline.
- We have to sequence these jobs in such a way to gain maximum profit.

Job Sequencing Example

- Let n (number of jobs) = 5, p (array of profits) = {10, 5, 15, 1, 20} and d (array of deadlines) = {1, 3, 2, 3, 2}
- First thing to note is that the maximum value of deadline is 3, since each job needs one unit of time, only 3 jobs can be performed. Question is doing which 3 can earn us the greatest profit?
- We will make a greedy choice and try to schedule the job with the highest profit first, i.e. j_5.
- Next we must decide when to do j_5. We have 3 timeslots available with us, 1, 2 & 3. In which timeslot should we schedule j_5?
- Here our policy would be to defer the job to be scheduled as much as possible. Note that the deadline for j_5 is 2, hence we can do j_5 latest by timeslot 2. We choose this timeslot to do j_5.
- Then we will consider the job with the 2nd greatest profit, i.e. j_3. and schedule it in the latest available timeslot, which is timeslot 1.
- We continue in this manner until all the timeslots are exhausted.

Job Sequencing Example

$n = 5, p = \{10, 5, 15, 1, 20\}, d = \{1, 3, 2, 3, 2\}$

- The following table illustrates the sequence of steps executed by our algorithm, for the given example.

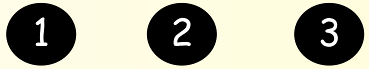
Schedule	Assigned Slots	Job Considered	Action	Profit
{}	none	5	Assign to slot 2	0
{5}	2	3	Assign to slot 1	20
{5, 3}	1, 2	1	Can't schedule, reject	35
{5, 3}	1, 2	2	Assign to slot 3	35
{5, 3, 2}	1, 2, 3	4	reject	40

- The optimum schedule is {5, 3, 2} and the optimum profit is 40.

Job Sequencing Example

$n = 5, p = \{10, 5, 15, 1, 20\}, d = \{1, 3, 2, 3, 2\}$

- To implement this algorithm we must be able to identify the latest available slot for a given deadline.
- To achieve this we partition the jobs in sets of their deadlines.
- Two jobs are in the same set if the latest available slot for both of them is the same.
- For example, initially we have 3 sets (remember the max deadline is 3), because all the slots are free.

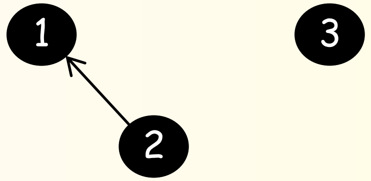


- This is another way of saying that a job with deadline 1 can be done latest in time slot 1, a job with deadline 2 can be done latest by time slot 2 and a job with deadline 3 can be done latest by time slot 3.
- This is represented by an array f[] = {0, 1, 2, 3}. Again, f[1] = 1 is to say that job 1 can be done latest by time slot 1, and so on.

Job Sequencing Example

$n = 5, p = \{10, 5, 15, 1, 20\}, d = \{1, 3, 2, 3, 2\}$

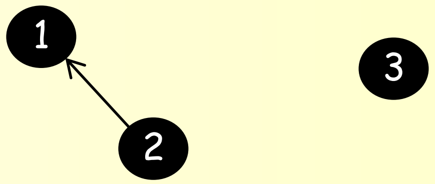
- At first we schedule j_5 in time slot 2, so time slot 2 is no longer available. Now if we get another job with deadline 2, the latest available time slot for that job is 1. But if we are to get a job with deadline 1, the latest available time slot for that job is tie slot 1 as well. So the latest available time slot for a job of deadline 2 and deadline 1 is now the same, i.e. 1.
- We represent this by putting 2 & 1 in the same set. To put them in the same set we perform a weightedUnion of sets {2} & {1}.



this means that a job with deadline 2 and a job with deadline 1 can be done latest by the same time slot

- Since now, a job with deadline 2 can be done latest by time slot 1, we set f[2] = 1, hence f[] = {0, 1, 1, 3}
- Next we consider j_3, which has a deadline 2. We want to know what is the latest available time slot for this job?

Job Sequencing Example

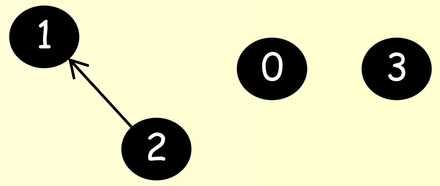


$n = 5, p = \{10, 5, 15, 1, 20\}, d = \{1, 3, 2, 3, 2\}$

- To find out which is the latest available time slot for j_3 which has deadline 2, we first perform a compressedFind of the element 2 in the disjoint set. This gives us 1, since 1 is the root of the tree in which 2 belongs. Now we check the value of f[1], which is again 1. So, we conclude that j_3 can be performed latest by time slot 1.
- But the latest available time slot for any job with deadline 1 is now 0, so we set f[1] = 0. So the array f now looks like {0, 0, 1, 3}
- The next job under our consideration (by greedy choice) would be j_1. But as we had seen it can't be scheduled. But how do we take this decision?
- To decide whether a job can be scheduled, we introduce a fictitious time slot 0.

- If the latest available time slot for a job is 0, then we conclude that it can't be scheduled.

Job Sequencing Example



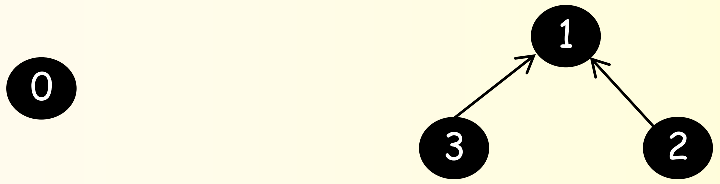
$n = 5, p = \{10, 5, 15, 1, 20\}, d = \{1, 3, 2, 3, 2\}$

- Now, to find out which is the latest available time slot for j_1 which has deadline 1, we first perform a compressedFind of the element 1 in the disjoint set. This gives us 1. Now we check the value of f[1], which is 0. Which means that j_1 can be performed latest by time slot 0. In other words, we reject j_1 as it can't be scheduled.
- The next job under our consideration (by greedy choice) would be $j_{2,}$ which has a deadline of 3.
- To find out the latest available time slot for j_2, we perform a compressedFind of the element 3 in the disjoint set. This gives us 3. Now we check the value of f[3], which is 3. Which means that j_2 can be performed latest by time slot 3. So we schedule it in this time slot.
- Which means that now that latest available time slot for jobs with deadline 3, 2 and 1 are now the same, i.e. 0. As before we represent this by putting them in the same set by performing a weightedUnion(1, 3)

Job Sequencing Example

$n = 5, p = \{10, 5, 15, 1, 20\}, d = \{1, 3, 2, 3, 2\}$

- So our disjoint set looks like the following now.



- Since, no other time slot (apart from 0) is available, no more jobs can be scheduled.
- The optimal schedule is thus {5, 3, 2} giving us an optimal profit of 20 + 15 + 5 = 40.

Representing a Job as a structure

- We will represent each job in our algorithm as a C structure, containing three fields, an id (to uniquely identify a job), a deadline and a profit value.

```
typedef struct Job {
    int id;
    int profit;
    int deadline;
} job;
```

- Suppose we are given an array of struct Job, our greedy strategy dictates that we pick jobs in order of profits, i.e. the highest profit job first, the 2nd highest profit job next and so on.
- So, it would require us to sort the array of jobs in decreasing order of profits to achieve our goal.
- Question is: how do we sort an array of structures in C?

The qsort function to rescue

- We will use the qsort (quick sort) function of C to sort an array of structures.
- The qsort function is defined in stdlib.h, so firstly make sure that you *#include <stdlib.h>* in your code.
- The qsort function takes four parameters: *qsort(base, n, size, cmpFunc);*
- Here base is the base pointer to the array to be sorted, n is the number of elements in the array, size is the size of each element in the array and cmpFunc is the pointer to the function which compares the elements in the array.
- We would of course need to provide the implementation of cmpFunc.
- The cmpFunc takes two constant void pointers to two elements in the array, and returns an integer which is positive if element1 > element2, negative if element1 < element2 and 0 if element1 == element2. For example, our job comparator looks like:

```
int jobCmp(const void* arg1, const void* arg2) {
    job* j1 = (job*)arg1;
    job* j2 = (job*)arg2;              // j1, j2 are pointers to two jobs
    return (j2->profit - j1->profit);// +ve if j2→profit is more, -ve if j2→profit is less, 0 if equal
}
```

Assignment

5.2 Implement the greedy job scheduling with deadlines algorithm

```
JobSchedule(job jobs[]) { // schedule the jobs optimally in j[1 ... k]
        qsort(jobs, n, sizeof(job), jobCmp);          // sort jobs in decreasing order of profits
        b = min(n, maxDeadline);                      // number of jobs that can be performed
        initDisjointSet(b);
        k = 0;
        for (i = 0 to b) f[i] = i;                    // mark all time slots available initially
        for (i = 0 to n - 1) {                        // consider jobs according to greedy choice
                q = compressedFind(min(n, jobs[i].deadline));   // latest available time slot for job i
                if (f[q] ≠ 0) {                       // proceed only if this job can be scheduled
                        j[k++] = jobs[i].id;          // schedule job i
                        m = compressedFind(f[q] - 1);
                        weightedUnion(m,  q);         // jobs with deadline m, q now belong to the same set
                        f[q] = f[m];                  // latest available time slots for deadline m, q are same
                }
        }
}
```

Thank you!!!

"If debugging is the process of removing bugs, then programming must be the process of putting them in."

- Dijkstra