AMERICAN INTERNATIONAL UNIVERSITY-BANGLADESH (AIUB)
FACULTY OF SCIENCE & TECHNOLOGY
DEPARTMENT OF COMPUTER SCIENCE

LAB MANUAL 1-6

CSC2211 Algorithms

Fall 2019-2020

Designed by *Faria Nawshin* and *Md. Manzurul Hasan*

AMERICAN INTERNATIONAL UNIVERSITY-BANGLADESH (AIUB)
FACULTY OF SCIENCE & TECHNOLOGY
DEPARTMENT OF COMPUTER SCIENCE

LAB MANUAL 01

CSC2211 Algorithms

Fall 2019-2020

Designed by *Faria Nawshin* and *Md. Manzurul Hasan*

TITLE

A review of Linear Searching and Different Sorting Algorithms

PREREQUISITE

- Have a clear understanding of Arrays
- Have a basic idea about Data Structure
- Have a basic idea about Pseudocode

OBJECTIVE

- To understand the significance of Algorithms
- To be able to implement Linear Search
- To be able to implement Bubble Sort
- To be able to implement Selection Sort
- To be able to implement Insertion Sort

THEORY

Algorithm

An algorithm is a step by step method of solving a problem. It is commonly used for data processing, calculation and other related computer and mathematical operations. An algorithm is also used to manipulate data in various ways, such as inserting a new data item, searching for a particular item or sorting an item.

Technically, computers use algorithms to list the detailed instructions for carrying out an operation. For example, to compute an employee's paycheck, the computer uses an algorithm. To accomplish this task, appropriate data must be entered into the system. In terms of efficiency, various algorithms are able to accomplish operations or problem solving easily and quickly.

In an algorithm, each instruction is identified and the order in which they should be carried out is planned. Algorithms are often used as a starting point for creating a computer program, and they are sometimes written as a flowchart or in pseudocode.

Searching Algorithms

Searching Algorithms are designed to check for an element or retrieve an element from any data structure where it is stored. Based on the type of search operation, these algorithms are generally classified into two categories:

Sequential Search: In this, the list or array is traversed sequentially and every element is checked. For example: Linear Search.

Interval Search: These algorithms are specifically designed for searching in sorted data-structures. This type of searching algorithms are much more efficient than Linear Search as they repeatedly target the center of the search structure and divide the search space in half. For Example: Binary Search.

Linear Search

- A linear search scans one item at a time, without jumping to any item.
- The worst case complexity is O(n), sometimes known an O(n) search
- Time taken to search elements keeps increasing as the numbers of elements are increased.

Example:

```
Input: arr[] = {10, 20, 80, 30, 60, 50,110, 100, 130, 170}
x = 110;
Output: 6
Element x is present at index 6

Input: arr[] = {10, 20, 80, 30, 60, 50,110, 100, 130, 170}
x = 175;
Output: -1
Element x is not present in arr[].
```

Pseudocode for Linear Search

```
procedure linear_search (list, value)

   for each item in the list
      if match item == value
         return the item's location
      end if
   end for

end procedure
```

C++ Code to Implement Linear Search

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

int SearchItem(int arr[], int n, int x)
{
    int i;
    for (i = 0; i < n; i++)
        if (arr[i] == x)
            return i;
    return -1;
}

int main()
{
    int n,x,i;
    cout<<"Enter size of array: ";
    cin>>n;
    int arr[n];
    cout<<"Enter Array Elements: ";
    for(i=0;i<n;i++)
    {
        cin>>arr[i];
    }
    cout<<"Enter Item to search for: ";
    cin>>x;

    int result = SearchItem(arr, n, x);
    if (result == -1)
        cout<<x<<" is not present in array";
    else
        cout<<x<<" is present at index " <<result;
   return 0;
}
```

Designed by *Faria Nawshin* and *Md. Manzurul Hasan*

Sorting Algorithms

A Sorting Algorithm is used to rearrange a given array or list elements according to a comparison operator on the elements. The comparison operator is used to decide the new order of element in the respective data structure.

Example:

Input: 7, 6, 1, 5, 3, 4, 9, 5

Output: 1, 3, 4, 5, 5, 6, 7, 9

Bubble Sort:

Bubble sort is a simple sorting algorithm. This sorting algorithm is comparison-based algorithm in which each pair of adjacent elements is compared and the elements are swapped if they are not in order. This algorithm is not suitable for large data sets as its average and worst case complexity are of O(n2) where n is the number of items.

Pseudocode for Bubble Sort

```
begin BubbleSort(list)

   for all elements of list
      if list[i] > list[i+1]
         swap(list[i], list[i+1])
      end if
   end for

   return list

end BubbleSort
```

C++ Code to Implement Bubble Sort

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

void Swap(int *x, int *y)
{
    int temp = *x;
    *x = *y;
    *y = temp;
}
```

Designed by *Faria Nawshin* and *Md. Manzurul Hasan*

```
void bubbleSort(int arr[], int n)
{
   int i, j;
   for (i = 0; i < n-1; i++)
    for (j = 0; j < n-i-1; j++)
           if (arr[j] > arr[j+1])
              Swap(&arr[j], &arr[j+1]);
}


void printArray(int arr[], int n)
{
    int i;
    for (i=0; i < n; i++)
        cout<<arr[i]<<" ";
    cout<<endl;
}


int main()
{
    int n,x,i;
    cout<<"Enter size of array: ";
    cin>>n;
    int arr[n];
    cout<<"Enter Array Elements: ";
    for(i=0;i<n;i++)
    {
        cin>>arr[i];
    }
    bubbleSort(arr,n);
    printArray(arr,n);
   return 0;
}
```

Selection Sort

The Selection sort algorithm is based on the idea of finding the minimum or maximum element in an unsorted array and then putting it in its correct position in a sorted array.

Pseudocode for Selection Sort

```
SELECTION-SORT(A)

      for j ← 1 to n-1

            smallest ← j

                 for i ← j + 1 to n

                         if A[ i ] < A[ smallest ]

                                 smallest ← i

                  Exchange A[ j ] ↔ A[ smallest ]
```

C++ Code to Implement Selection Sort

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


void Swap(int *x, int *y)
{
    int temp = *x;
    *x = *y;
    *y = temp;
}


void selectionSort(int arr[], int n)
{
    int i, j, min_idx;

    for (i = 0; i < n-1; i++)
    {
        min_idx = i;
        for (j = i+1; j < n; j++)
        if (arr[j] < arr[min_idx])
```

```
            min_idx = j;

        Swap(&arr[min_idx], &arr[i]);
    }
}


void printArray(int arr[], int n)
{
    int i;
    for (i=0; i < n; i++)
        cout<<arr[i]<<" ";
    cout<<endl;
}


int main()
{
    int n,x,i;
    cout<<"Enter size of array: ";
    cin>>n;
    int arr[n];
    cout<<"Enter Array Elements: ";
    for(i=0;i<n;i++)
    {
        cin>>arr[i];
    }
    selectionSort(arr,n);
    printArray(arr,n);
   return 0;
}
```

Insertion Sort

Insertion sort is based on the idea that one element from the input elements is consumed in each iteration to find its correct position i.e, the position to which it belongs in a sorted array.

It iterates the input elements by growing the sorted array at each iteration. It compares the current element with the largest value in the sorted array. If the current element is greater, then it leaves the element in its place and moves on to the next element else it finds its correct position in the sorted array and moves it to that position. This is done by shifting all the elements, which are larger than the current element, in the sorted array to one position ahead.

Pseudocode for Insertion Sort

INSERTION-SORT(A)

for j = 2 to n

    key ← A [j]

        j ← i – 1

        while i > 0 and A[i] > key

                A[i+1] ← A[i]

                i ← i – 1

        A[j+1] ← key

C++ Code to Implement Insertion Sort

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

void Swap(int *x, int *y)
{
    int temp = *x;
    *x = *y;
    *y = temp;
}


void insertionSort(int arr[], int n)
{
    int i, key, j;
    for (i = 1; i < n; i++)
    {
        key = arr[i];
        j = i - 1;

        while (j >= 0 && arr[j] > key)
        {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
    }
}
```

Designed by *Faria Nawshin* and *Md. Manzurul Hasan*

```
void printArray(int arr[], int n)
{
    int i;
    for (i=0; i < n; i++)
        cout<<arr[i]<<" ";
    cout<<endl;
}


int main()
{
    int n,x,i;
    cout<<"Enter size of array: ";
    cin>>n;
    int arr[n];
    cout<<"Enter Array Elements: ";
    for(i=0;i<n;i++)
    {
        cin>>arr[i];
    }
    insertionSort(arr,n);
    printArray(arr,n);
   return 0;
}
```

Counting Sort Algorithm

Counting sort is a sorting technique based on keys between a specific range. It works by counting the number of objects having distinct key values (kind of hashing). Then doing some arithmetic to calculate the position of each object in the output sequence.

Let us understand it with the help of an example.

```
For simplicity, consider the data in the range 0 to 9.
Input data: 1, 4, 1, 2, 7, 5, 2
  1) Take a count array to store the count of each unique
object.
  Index:     0  1  2  3  4  5  6  7  8  9
  Count:     0  2  2  0   1  1  0  1  0  0

  2) Modify the count array such that each element at each index
  stores the sum of previous counts.
  Index:     0  1  2  3  4  5  6  7  8  9
  Count:     0  2  4  4  5  6  6  7  7  7
```

The modified count array indicates the position of each object in the output sequence.

3) Output each object from the input sequence followed by decreasing its count by 1.
Process the input data: 1, 4, 1, 2, 7, 5, 2. Position of 1 is 2. Put data 1 at index 2 in output. Decrease count by 1 to place next data 1 at an index 1 smaller than this index.

Pseudocode for Counting Sort

```
CountingSort(A)

  for i = 0 to k do
     c[i] = 0

  //Storing Count of each element

  for j = 0 to n do
     c[A[j]] = c[A[j]] + 1

  // Change C[i] such that it contains actual position of these
elements in output array

  for i = 1 to k do
     c[i] = c[i] + c[i-1]

  //Build Output array from C[i]

  for j = n-1 down to 0 do
     B[ c[A[j]]-1 ] = A[j]
     c[A[j]] = c[A[j]] - 1
end func
```

C++ Code to Implement Counting Sort

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

int k=0;

void Counting_Sort(int A[],int B[],int n)
{
     int C[k+1];

     for(int i=0;i<=k;i++)
     {
          C[i]=0;
     }
     for(int j=1;j<=n;j++)
     {
          C[A[j]]++;
     }
     for(int i=1;i<=k;i++)
     {
          C[i]+=C[i-1];
     }
     for(int j=n;j>=1;j--)
     {
          B[C[A[j]]]=A[j];

          C[A[j]]=C[A[j]]-1;
     }
}
int main()
{
     int n;
     cout<<"Enter the size of the array :";
     cin>>n;

     int A[n],B[n];

     for(int i=1;i<=n;i++)
     {
          cin>>A[i];
          if(A[i]>k)
          {
               /*It will modify k if an element
               occurs whose value is greater than k*/
               k=A[i];
```

```
            }
        }
        Counting_Sort(A,B,n);
        /*It will print the sorted sequence on the
        console*/
        for(int i=1;i<=n;i++)
        {
            cout<<B[i]<<" ";
        }

        cout<<endl;
        return 0;
}
```

AMERICAN INTERNATIONAL UNIVERSITY-BANGLADESH (AIUB)
FACULTY OF SCIENCE & TECHNOLOGY
DEPARTMENT OF COMPUTER SCIENCE

LAB MANUAL 02

CSC2211 Algorithms

Fall 2019-2020

Designed by *Faria Nawshin* and *Md. Manzurul Hasan*

TITLE

An Introduction to Recursion and Recursive Algorithm

PREREQUISITE

- Have a clear understanding of Stack
- Know how to analyze the runtime of algorithms
- Have a clear understanding of return statement

OBJECTIVE

- To know about Recursion and Recursive Definition
- To know about Recursive Algorithm
- To know about Master Theorem
- To know about Quick Sort
- To know about Merge Sort

THEORY

Recursion

The process in which a function calls itself directly or indirectly is called recursion and the corresponding function is called as recursive function. Using recursive algorithm, certain problems can be solved quite easily. Examples of such problems are Towers of Hanoi (TOH), Inorder/Preorder/Postorder Tree Traversals, DFS of Graph, etc.

Recursive Definition

A recursive definition of a set always consists of three distinct clauses:

What is base condition in recursion?

In the recursive program, the solution to the base case is provided and the solution of the bigger problem is expressed in terms of smaller problems.

Designed by *Faria Nawshin* and *Md. Manzurul Hasan*

```
int fact(int n)
{
    if (n < = 1) // base case
        return 1;
    else
        return n*fact(n-1);
}
```

In the above example, base case for n < = 1 is defined and larger value of number can be solved by converting to smaller one till base case is reached.

Why Stack Overflow error occurs in recursion?

If the base case is not reached or not defined, then the stack overflow problem may arise. Let us take an example to understand this.

```
int fact(int n)
{
    // wrong base case (it may cause stack overflow)
    if (n == 100)
        return 1;

    else
        return n*fact(n-1);
}
```

If fact(10) is called, it will call fact(9), fact(8), fact(7) and so on but the number will never reach 100. So, the base case is not reached. If the memory is exhausted by these functions on the stack, it will cause a stack overflow error.

Recursive Algorithm

A recursive algorithm is an algorithm which calls itself with "smaller (or simpler)" input values, and which obtains the result for the current input by applying simple operations to the returned value for the smaller (or simpler) input. More generally if a problem can be solved utilizing solutions to smaller versions of the same problem, and the smaller versions reduce to easily solvable cases, then one can use a recursive algorithm to solve that problem. For example, the elements of a recursively defined set, or the value of a recursively defined function can be obtained by a recursive algorithm.

If a set or a function is defined recursively, then a recursive algorithm to compute its members or values mirrors the definition. Initial steps of the recursive algorithm correspond to the basis

clause of the recursive definition and they identify the basis elements. They are then followed by steps corresponding to the inductive clause, which reduce the computation for an element of one generation to that of elements of the immediately preceding generation.

In general, recursive computer programs require more memory and computation compared with iterative algorithms, but they are simpler and for many cases a natural way of thinking about the problem.

Direct Recursive:

Algorithm A is said to be direct recursive if it calls itself directly.

Indirect Recursive:

Algorithm A is said to be indirect recursive if it calls another algorithm which in turn calls A.

Binary Search Algorithm:

Binary search is a fast search algorithm with run-time complexity of O(log n). This search algorithm works on the principle of divide and conquer. For this algorithm to work properly, **the data collection should be in the sorted form.**

Binary search looks for a particular item by comparing the middle most item of the collection. If a match occurs, then the index of item is returned. If the middle item is greater than the item, then the item is searched in the sub-array to the left of the middle item. Otherwise, the item is searched for in the sub-array to the right of the middle item. This process continues on the sub-array as well until the size of the subarray reduces to zero

Designed by *Faria Nawshin* and *Md. Manzurul Hasan*

Pseudocode for Binary Search

```
// initially called with low = 0, high = N - 1

  BinarySearch_Right(A[0..N-1], value, low, high) {
      if (high < low)
          return low
      mid = (low + high) / 2
      if (A[mid] > value)
          return BinarySearch_Right(A, value, low, mid-1)
      else
          return BinarySearch_Right(A, value, mid+1, high)
  }
```

C++ Code to Implement Binary Search

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

void insertionSort(int arr[], int n)
{
    int i, key, j;
    for (i = 1; i < n; i++)
    {
        key = arr[i];
        j = i - 1;

        while (j >= 0 && arr[j] > key)
        {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
    }
}

void printArray(int arr[], int n)
{
    int i;
    for (i=0; i < n; i++)
        cout<<arr[i]<<" ";
    cout<<endl;
}

int binarySearch(int arr[], int l, int r, int x)
```

```
{
    if (r >= l) {
        int mid = l + (r - l) / 2;

        // If the element is present at the middle
        // itself
        if (arr[mid] == x)
            return mid;

        // If element is smaller than mid, then
        // it can only be present in left subarray
        if (arr[mid] > x)
            return binarySearch(arr, l, mid - 1, x);

        // Else the element can only be present
        // in right subarray
        return binarySearch(arr, mid + 1, r, x);
    }

    // We reach here when element is not
    // present in array
    return -1;
}

int main()
{
    int n,x,i;
    cout<<"Enter size of array: ";
    cin>>n;
    int arr[n];
    cout<<"Enter Array Elements: ";
    for(i=0;i<n;i++)
    {
        cin>>arr[i];
    }
    insertionSort(arr, n);
    cout<<"After sorting..."<<endl;
    printArray(arr,n);
    cout<<endl;
    cout<<"Enter Item to search for: ";
    cin>>x;

    int result = binarySearch(arr, 0, n - 1, x);
    if (result == -1)
        cout<<x<<" is not present in array";
    else
```

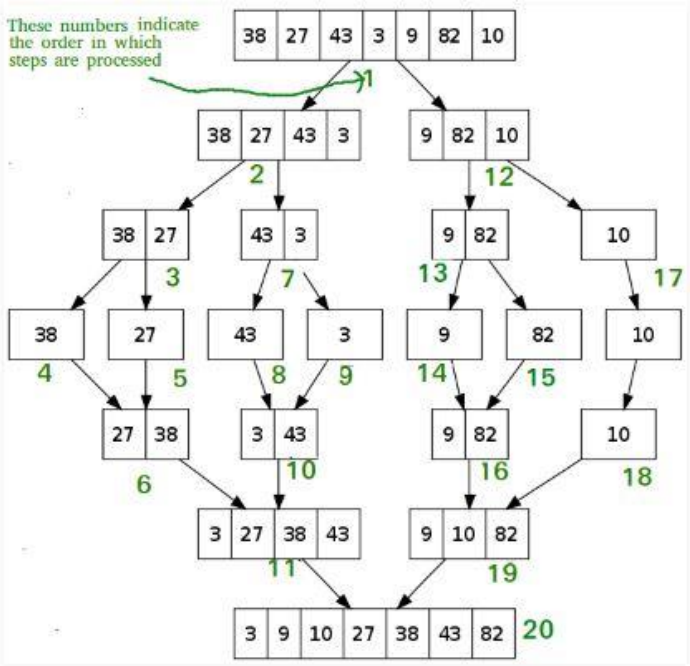
```
        cout<<x<<" is present at index " <<result;
    return 0;

}
```

Merge Sort Algorithm

Merge Sort is a Divide and Conquer algorithm. It divides input array in 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 key process that assumes that arr[l..m] and arr[m+1..r] are sorted and merges the two sorted sub-arrays into one. See following C implementation for details.

The following diagram shows the complete merge sort process for an example array {38, 27, 43, 3, 9, 82, 10}. If we take a closer look at the diagram, we can see that the array is recursively divided in two halves till the size becomes 1. Once the size becomes 1, the merge processes comes into action and starts merging arrays back till the complete array is merged.



Designed by *Faria Nawshin* and *Md. Manzurul Hasan*

Pseudocode for Merge Sort

```
MERGE-SORT(A, p, r)
1  if p < r
2      q = ⌊(p + r)/2⌋
3      MERGE-SORT(A, p, q)
4      MERGE-SORT(A, q + 1, r)
5      MERGE(A, p, q, r)

MERGE(A, p, q, r)
1  n1 = q − p + 1
2  n2 = r − q
3  let L[1..n1 + 1] and R[1..n2 + 1] be new arrays
4  for i = 1 to n1
5      L[i] = A[p + i − 1]
6  for j = 1 to n2
7      R[j] = A[q + j]
8  L[n1 + 1] = ∞
9  R[n2 + 1] = ∞
10 i = 1
11 j = 1
12 for k = p to r
13     if L[i] ≤ R[j]
14         A[k] = L[i]
15         i = i + 1
16     else A[k] = R[j]
17         j = j + 1
```

C++ Code to Implement Merge Sort

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

void merge(int arr[], int l, int m, int r)
{
    int i, j, k;
    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 (i = 0; i < n1; i++)
        L[i] = arr[l + i];
    for (j = 0; j < n2; j++)
        R[j] = arr[m + 1+ j];

    /* Merge the temp arrays back into arr[l..r]*/
    i = 0; // Initial index of first subarray
```

```
    j = 0; // Initial index of second subarray
    k = l; // Initial index of merged subarray
    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)
    {
        // Same as (l+r)/2, but avoids overflow for
        // large l and h
        int m = l+(r-l)/2;
```

```
        // Sort first and second halves
        mergeSort(arr, l, m);
        mergeSort(arr, m+1, r);

        merge(arr, l, m, r);
    }
}

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

int main()
{
    int n,x,i;
    cout<<"Enter size of array: ";
    cin>>n;
    int arr[n];
    cout<<"Enter Array Elements: ";
    for(i=0;i<n;i++)
    {
        cin>>arr[i];
    }
    mergeSort(arr, 0, n - 1);

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

}
```

Quick Sort Algorithm

Quick sort is a highly efficient sorting algorithm and is based on partitioning of array of data into smaller arrays. A large array is partitioned into two arrays one of which holds values smaller than the specified value, say pivot, based on which the partition is made and another array holds values greater than the pivot value.

Quick sort partitions an array and then calls itself recursively twice to sort the two resulting subarrays. This algorithm is quite efficient for large-sized data sets as its average and worst case complexity are of $O(n^2)$, where n is the number of items.

C++ to Implement Quick Sort

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

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

    for (int j = low; j <= high- 1; j++)
    {
        // If current element is smaller than or
        // equal to 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 n,x,i;
    cout<<"Enter size of array: ";
    cin>>n;
    int arr[n];
    cout<<"Enter Array Elements: ";
    for(i=0;i<n;i++)
    {
        cin>>arr[i];
    }
    quickSort(arr, 0, n-1);
    cout<<"Sorted array: "<<endl;
    printArray(arr, n);
```

```
    return 0;
}
```

AMERICAN INTERNATIONAL UNIVERSITY-BANGLADESH (AIUB)
FACULTY OF SCIENCE & TECHNOLOGY
DEPARTMENT OF COMPUTER SCIENCE

LAB MANUAL 03

CSC2211 Algorithms

Fall 2019-2020

Designed by *Faria Nawshin* and *Md. Manzurul Hasan*

TITLE

An Introduction to Greedy Algorithm

OBJECTIVE

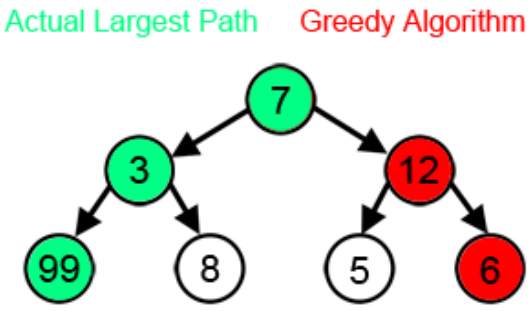
- To know about Greedy Algorithm
- To know about the Structure of Greedy Algorithm
- To know about Activity Selection Algorithm
- To know about Fractional Knapsack Problem

THEORY

Greedy Strategy:

A greedy algorithm is a simple, intuitive algorithm that is used in optimization problems. The algorithm makes the optimal choice at each step as it attempts to find the overall optimal way to solve the entire problem. Greedy algorithms are quite successful in some problems, such as Huffman encoding which is used to compress data, or Dijkstra's algorithm, which is used to find the shortest path through a graph.

However, in many problems, a greedy strategy does not produce an optimal solution. For example, in the example below, the greedy algorithm seeks to find the path with the largest sum. It does this by selecting the largest available number at each step. The greedy algorithm fails to find the largest sum, however, because it makes decisions based only on the information it has at any one step, without regard to the overall problem.



Designed by *Faria Nawshin* and *Md. Manzurul Hasan*

Structure of a Greedy Algorithm

Greedy algorithms take all of the data in a particular problem, and then set a rule for which elements to add to the solution at each step of the algorithm. In the animation above, the set of data is all of the numbers in the graph, and the rule was to select the largest number available at each level of the graph. The solution that the algorithm builds is the sum of all of those choices.

If both of the properties below are true, a greedy algorithm can be used to solve the problem.

Greedy choice property: A global (overall) optimal solution can be reached by choosing the optimal choice at each step.

Optimal substructure: A problem has an optimal substructure if an optimal solution to the entire problem contains the optimal solutions to the sub-problems.

Greedy Algorithm to find Minimum number of Coins

Given a value V, if we want to make change for V Taka, and we have infinite supply of each of the denominations in Bangladeshi currency, i.e., we have infinite supply of { 1, 2, 5, 10, 20, 50, 100, 500, 1000} valued coins/notes, what is the minimum number of coins and/or notes needed to make the change?

Examples:

```
Input: V = 70
Output: 2
We need a 50 Taka note and a 20 Taka note.

Input: V = 121
Output: 3
We need a 100 Taka note, a 20 Taka note and a 1 Taka note.
```

Designed by *Faria Nawshin* and *Md. Manzurul Hasan*

Activity Selection Algorithm

We will learn about Activity Selection Problem, a greedy way to find the maximum number of activities a person or machine can perform, assuming that the person or machine involved can only work on a single activity at a time.

Points to remember

For this algorithm we have a list of activities with their starting time and finishing time.

- Our goal is to select maximum number of non-conflicting activities that can be performed by a person or a machine, assuming that the person or machine involved can work on a single activity at a time.
- Any two activities are said to be non-conflicting if starting time of one activity is greater than or equal to the finishing time of the other activity.
- In order to solve this problem we first sort the activities as per their finishing time in ascending order.
- Then we select non-conflicting activities.

Problem

Consider the following 8 activities with their starting and finishing time.

Activity	a1	a2	a3	a4	a5	a6	a7	a8
start	1	0	1	4	2	5	3	4
finish	3	4	2	6	9	8	5	5

Steps:

- sort the activities as per finishing time in ascending order
- select the first activity
- select the new activity if its starting time is greater than or equal to the previously selected activity
- REPEAT step 3 till all activities are checked

Designed by *Faria Nawshin* and *Md. Manzurul Hasan*

Pseudocode for Activity Selection Algorithm

```
Set i = 0;    //pointing at first element
for j = 1 to n-1 do
  if start time of j >= finish time of i then
    Print j
    Set i = j

  endif

endfor
```

C++ Code to Implement Activity Selection Algorithm

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

struct Activity
{
    int startTime, finishTime;
};

void SWAP(struct Activity *p, struct Activity *q)
{
    struct Activity t;
    t=*p;
    *p=*q;
    *q=t;
}

void bubbleSort(struct Activity a[], int n)
{
    int pass,i;
    for(pass=1;pass<n;pass++)
    {
        for(i=0;i<n;i++)
        {
            if(a[i].finishTime>a[i+1].finishTime)
            {
                SWAP(&a[i],&a[i+1]);
            }
        }
    }
}
```

```
void activitySelection(struct Activity a[], int n)
{
   int i,j;
   i=0;
   cout<<a[0].startTime<<" "<<a[0].finishTime<<endl;
   for(j=i+1;j<n;j++)
   {
       if(a[j].startTime>=a[i].finishTime)
       {
            cout<<a[j].startTime<<" "<<a[j].finishTime<<endl;
            i=j;
       }
   }

}

int main()
{
    int n,i;
    cout<<"Enter Number of Activities: ";
    cin>>n;
    struct Activity act[n];
    for(i=0;i<n;i++)
    {
        cout<<"Enter start time: ";
        cin>>act[i].startTime;
        cout<<"Enter finish time: ";
        cin>>act[i].finishTime;
    }

    bubbleSort(act,n);
    /*cout<<"After sorting"<<endl;
    for(i=0;i<n;i++)
    {
        cout<<act[i].startTime<<" "<<act[i].finishTime<<endl;
    }*/
    cout<<"Selected Activities: "<<endl;
    activitySelection(act,n);

}
```

Fractional Knapsack Problem

We will learn about fractional knapsack problem, a greedy algorithm. In this problem the objective is to fill the knapsack with items to get maximum benefit (value or profit) without crossing the weight capacity of the knapsack. And we are also allowed to take an item in fractional part.

Points to remember

- In this problem we have a Knapsack that has a weight limit W
- There are items i1, i2, ..., in each having weight w1, w2, … wn and some benefit (value or profit) associated with it v1, v2, ..., vn
- Our objective is to maximise the benefit such that the total weight inside the knapsack is at most W
- And we are also allowed to take an item in fractional part

Problem

Assume that we have a knapsack with max weight capacity, W = 16.

Our objective is to fill the knapsack with items such that the benefit (value or profit) is maximum.

Consider the following items and their associated weight and value

ITEM	WEIGHT	VALUE
i1	6	6
i2	10	2
i3	3	1
i4	5	8
i5	1	3
i6	3	5

Designed by *Faria Nawshin* and *Md. Manzurul Hasan*

Steps:

- Compute density = (value/weight)
- Sort the items as per the value density in descending order
- Take as much item as possible not already taken in the knapsack

Compute density = (value/weight)

ITEM	WEIGHT	VALUE	DENSITY
i1	6	6	1.000
i2	10	2	0.200
i3	3	1	0.333
i4	5	8	1.600
i5	1	3	3.000
i6	3	5	1.667

Sort the items as per density in descending order

ITEM	WEIGHT	VALUE	DENSITY
i5	1	3	3.000
i6	3	5	1.667

Designed by *Faria Nawshin* and *Md. Manzurul Hasan*

i4	5	8	1.600
i1	6	6	1.000
i3	3	1	0.333
i2	10	2	0.200

Now we will pick items such that our benefit is maximum and total weight of the selected items is at most W.

Values after calculation

ITEM	WEIGHT	VALUE	TOTAL WEIGHT	BENEFIT
i5	1	3	1.000	3.000
i6	3	5	4.000	8.000
i4	5	8	9.000	16.000
i1	6	6	15.000	22.000
i3	1	0.333	16.000	22.333

So, total weight in the knapsack = 16 and total value inside it = 22.333336

C++ Code to Implement Fractional Knapsack Problem

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

struct Item {
  char id[5];
  int weight;
  int value;
  float density;
};

void fractionalKnapsack(Item items[], int n, int W);

int main() {
  int i, j;

  //list items
  Item items[6] = {
    {"i1",  6, 6, 0},
    {"i2", 10, 2, 0},
    {"i3",  3, 1, 0},
    {"i4",  5, 8, 0},
    {"i5",  1, 3, 0},
    {"i6",  3, 5, 0}
  };

  //temp item
  Item temp;

  //number of items
  int n = 6;

  //max weight limit of knapsack
  int W = 16;

  //compute desity = (value/weight)
  for(i = 0; i < n; i++) {
    items[i].density = float(items[i].value) / items[i].weight;
  }

  //sort by density in descending order
  for(i = 1; i < n; i++) {
    for(j = 0; j < n - i; j++) {
      if(items[j+1].density > items[j].density) {
```

```
        temp = items[j+1];
        items[j+1] = items[j];
        items[j] = temp;
      }
    }
  }

  fractionalKnapsack(items, n, W);

  return 0;
}

void fractionalKnapsack(Item items[], int n, int W) {
  int i, wt;
  float value;

  float totalWeight = 0, totalBenefit = 0;

  for(i = 0; i < n; i++) {
    if(items[i].weight + totalWeight <= W) {

      totalWeight += items[i].weight;
      totalBenefit += items[i].value;

      cout<<"Selected Item: "<<items[i].id<<" \tWeight:
"<<items[i].weight<<" \tValue: "<<items[i].value<<"\tTotal
Weight: "<<totalWeight<<"\tTotal Benefit: "<<totalBenefit
<<endl;
      }
      else {
      wt = (W - totalWeight);
      value = wt * (float(items[i].value) / items[i].weight);

      totalWeight += wt;
      totalBenefit += value;

      cout<<"Selected Item: "<<items[i].id<<"\tWeight:
"<<wt<<"\tValue: "<<value<<"\tTotal Weight:
"<<totalWeight<<"\tTotal Benefit: "<<totalBenefit<<endl;

      break;
    }
  }

  cout<<"Total Weight: "<<totalWeight<<endl;
  cout<<"Total Benefit: "<<totalBenefit<<endl;
```

Designed by *Faria Nawshin* and *Md. Manzurul Hasan*

```
}
```

Designed by *Faria Nawshin* and *Md. Manzurul Hasan*

AMERICAN INTERNATIONAL UNIVERSITY-BANGLADESH (AIUB)
FACULTY OF SCIENCE & TECHNOLOGY
DEPARTMENT OF COMPUTER SCIENCE

LAB MANUAL 04

CSC2211 Algorithms

Fall 2019-2020

Designed by *Faria Nawshin* and *Md. Manzurul Hasan*

TITLE

An Introduction to Dynamic Programming

OBJECTIVE

- To know about Dynamic Programming
- To know about Properties of Dynamic Programming
- To know about 0/1 knapsack Algorithm
- To know about Matrix Chain Multiplication Algorithm (MCM)
- To know about Longest Common Subsequence Algorithm (LCS)

THEORY

Dynamic Programming

Dynamic Programming is mainly an optimization over plain recursion. Wherever we see a recursive solution that has repeated calls for same inputs, we can optimize it using Dynamic Programming. The idea is to simply store the results of sub problems, so that we do not have to re-compute them when needed later. This simple optimization reduces time complexities from exponential to polynomial. For example, if we write simple recursive solution for Fibonacci Numbers, we get exponential time complexity and if we optimize it by storing solutions of sub problems, time complexity reduces to linear.

A code for it using pure recursion:

```
int fib (int n) {
      if (n < 2)
          return 1;
      return fib(n-1) + fib(n-2);
  }
```

Using Dynamic Programming approach with memoization:

```
void fib () {
       fibresult[0] = 1;
       fibresult[1] = 1;
       for (int i = 2; i<n; i++)
          fibresult[i] = fibresult[i-1] + fibresult[i-2];
   }
```

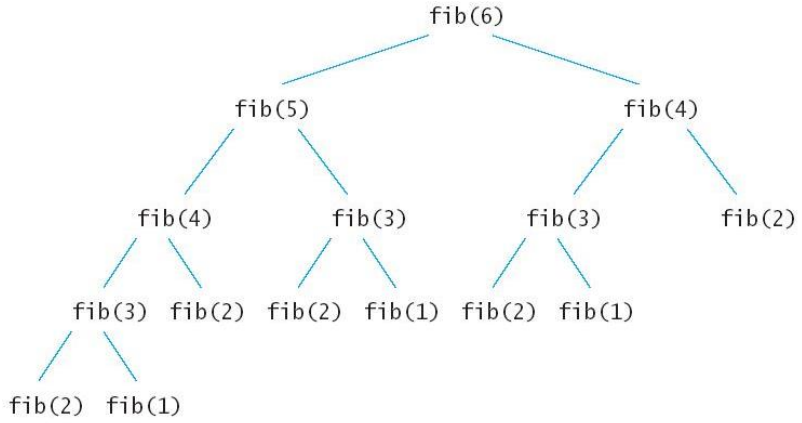
Designed by *Faria Nawshin* and *Md. Manzurul Hasan*

In the recursive code, a lot of values are being recalculated multiple times. We could do good with calculating each unique quantity only once. Take a look at the image to understand that how certain values were being recalculated in the recursive way:

Properties of Dynamic Programming

1) Overlapping Subproblems:

Like Divide and Conquer, Dynamic Programming combines solutions to sub-problems. Dynamic Programming is mainly used when solutions of same subproblems are needed again and again. In dynamic programming, computed solutions to subproblems are stored in a table so that these don't have to be recomputed. So Dynamic Programming is not useful when there are no common (overlapping) subproblems because there is no point storing the solutions if they are not needed again. For example, Binary Search doesn't have common subproblems. If we take an example of following recursive program for Fibonacci Numbers, there are many subproblems which are solved again and again.



We can see that the function fib(3) is being called 3 times. If we would have stored the value of fib(3), then instead of computing it again, we could have reused the old stored value. There are following two different ways to store the values so that these values can be reused:

- **Memoization (Top Down)**
- **Tabulation (Bottom Up)**

a) **Memoization (Top Down):**
The memoized program for a problem is similar to the recursive version with a small modification that it looks into a lookup table before computing solutions. We initialize a lookup array with all initial values as NIL. Whenever we need the solution to a subproblem, we first look into the lookup table. If the precomputed value is there then we return that value, otherwise, we calculate the value and put the result in the lookup table so that it can be reused later.

b) **Tabulation (Bottom Up):**
The tabulated program for a given problem builds a table in bottom up fashion and returns the last entry from table. For example, for the same Fibonacci number, we first calculate fib(0) then fib(1) then fib(2) then fib(3) and so on. So literally, we are building the solutions of subproblems bottom-up.

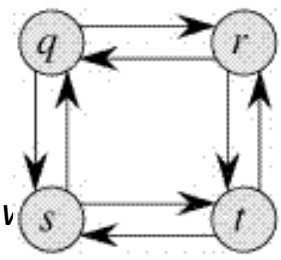
2) Optimal Substructure:

A given problems has Optimal Substructure Property if optimal solution of the given problem can be obtained by using optimal solutions of its sub problems.

For example, the Shortest Path problem has following optimal substructure property:

If a node x lies in the shortest path from a source node u to destination node v then the shortest path from u to v is combination of shortest path from u to x and shortest path from x to v. The standard All Pair Shortest Path algorithms like Floyd–Warshall and Bellman–Ford are typical examples of Dynamic Programming.

On the other hand, the Longest Path problem doesn't have the Optimal Substructure property. Here by Longest Path we mean longest simple path (path without cycle) between two nodes. Consider the following unweighted graph given in the CLRS book. There are two longest paths from q to t: q→r→t and q→s→t. Unlike shortest paths, these longest paths do not have the optimal substructure property. For example, the longest path q→r→t is not a combination of longest path from q to r and longest path from r to t, because the longest path from q to r is q→s→t→r and the longest path from r to t is r→q→s→t.



Designed by *Faria Nawshin* and *Md. Manzurul Hasan*

Difference between Greedy Algorithm and Dynamic Programming

- Greedy algorithm is one which finds feasible solution at every stage with the hope of finding optimal solution whereas Dynamic programming is one which breaks the problems into series of overlapping sub-problems.
- Greedy algorithm never reconsiders its choices whereas Dynamic programming may consider the previous state.
- Greedy algorithm is less efficient whereas Dynamic programming is more efficient.
- Greedy algorithm has a local choice of the sub-problems whereas Dynamic programming would solve the all sub-problems and then select one that would lead to an optimal solution.
- Greedy algorithm takes decision in one time whereas Dynamic programming takes decision at every stage.
- Greedy algorithm work based on choice property whereas Dynamic programming work based on principle of optimality.
- Greedy algorithm follows the top-down strategy whereas Dynamic programming follows the bottom-up strategy.

0/1 Knapsack Algorithm

We will be learning about 0/1 Knapsack problem. In this dynamic programming problem we have n items each with an associated weight and value (benefit or profit). The objective is to fill the knapsack with items such that we have a maximum profit without crossing the weight limit of the knapsack. Since this is a 0 1 knapsack problem hence we can either take an entire item or reject it completely. We cannot break an item and fill the knapsack.

Point to remember

- In this problem we have a Knapsack that has a weight limit W.
- There are items i1, i2, ..., in each having weight w1, w2, … wn and some benefit (value or profit) associated with it v1, v2, ... vn
- Our objective is to maximise the benefit such that the total weight inside the knapsack is at most W.
- Since this is a 0 1 Knapsack problem algorithm so, we can either take an entire item or reject it completely. We can not break an item and fill the knapsack.

Designed by *Faria Nawshin* and *Md. Manzurul Hasan*

Example

Assume that we have a knapsack with max weight capacity W = 5
Our objective is to fill the knapsack with items such that the benefit (value or profit) is maximum.

Following table contains the items along with their value and weight.

item	1	2	3	4
value	100	20	60	40
weight	3	2	4	1

Total items n = 4
Total capacity of the knapsack W = 5

Now we create a value table V[i,w] where, i denotes number of items and w denotes the weight of the items.
Rows denote the items and columns denote the weight.
As there are 4 items so, we have 5 rows from 0 to 4.
And the weight limit of the knapsack is W = 5 so, we have 6 columns from 0 to 5

We fill the first row i = 0 with 0. This means when 0 item is considered weight is 0.

Then we fill the first column w = 0 with 0. This means when weight is 0 then items considered is 0.

After calculation, the value table V

V[i,w]	w = 0	1	2	3	4	5
i = 0	0	0	0	0	0	0
1	0	0	0	100	100	100
2	0	0	20	100	100	120
3	0	0	20	100	100	120

Designed by *Faria Nawshin* and *Md. Manzurul Hasan*

4	0	40	40	100	140	140

Maximum value earned
Max Value = V[n,W]
= V[4,5]
= 140

C++ Code to Implement 0/1 Knapsack Algorithm

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

int getMax(int x, int y)
{
    if(x>y)
    {
        return x;
    }
    else
        return y;
}

void knapsack(int v[], int wt[], int n, int M)
{
    int K[n+1][M+1];
    int i,w;
    for(w=0;w<=M;w++)
    {
        K[0][w]=0;
    }
    for(i=0;i<=n;i++)
    {
        K[i][0]=0;
    }
    for(i=1;i<=n;i++)
    {
        for(w=1;w<=M;w++)
        {
            if(wt[i]<=w)
            {
                K[i][w] = getMax(K[i-1][w], v[i] + K[i-1][w -
wt[i]]);
            }
```

```
            else
                K[i][w]= K[i-1][w];
        }
    }
    cout<<"Maximum profit: "<<K[n][M]<<endl;
}

int main()
{
    int n,i,M;
    cout<<"Enter number of items: ";
    cin>>n;
    int wt[n], v[n];

    for(i=0;i<n;i++)
    {
        cout<<"Enter weight for item "<<i+1<<": ";
        cin>>wt[i];
        cout<<"Enter value for item "<<i+1<<": ";
        cin>>v[i];
    }
    cout<<"Enter Knapsack Capacity: ";
    cin>>M;
    knapsack(v,wt,n,M);
}
```

Matrix Chain Multiplication Algorithm

Given a sequence of matrices, find the most efficient way to multiply these matrices together. The problem is not actually to perform the multiplications, but merely to decide in which order to perform the multiplications.

We have many options to multiply a chain of matrices because matrix multiplication is associative. In other words, no matter how we parenthesize the product, the result will be the same. For example, if we had four matrices A, B, C, and D, we would have:

(ABC)D = (AB)(CD) = A(BCD) = ...

However, the order in which we parenthesize the product affects the number of simple arithmetic operations needed to compute the product, or the efficiency. For example, suppose A is a 10 × 30 matrix, B is a 30 × 5 matrix, and C is a 5 × 60 matrix. Then,

(AB)C = (10×30×5) + (10×5×60) = 1500 + 3000 = 4500 operations
A(BC) = (30×5×60) + (10×30×60) = 9000 + 18000 = 27000 operations.

Clearly the first parenthesization requires less number of operations.

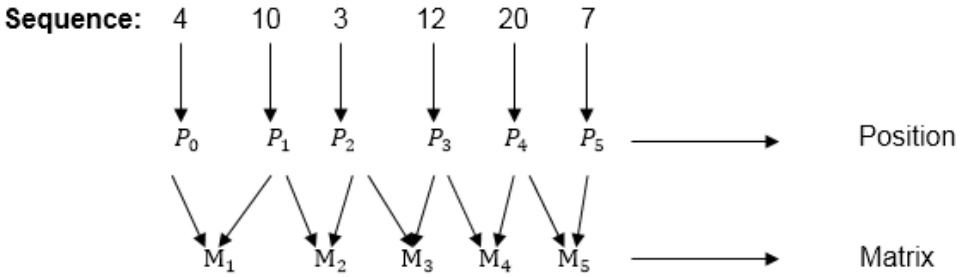
Given an array p[] which represents the chain of matrices such that the ith matrix Ai is of dimension p[i-1] x p[i]. We need to write a function MatrixChainOrder() that should return the minimum number of multiplications needed to multiply the chain.

Example:

We are given the sequence {4, 10, 3, 12, 20, and 7}. The matrices have size 4 x 10, 10 x 3, 3 x 12, 12 x 20, 20 x 7. We need to compute M [i,j], 0 ≤ i, j≤ 5. We know M [i, i] = 0 for all i.

$M[i,j] = \min_{i<=k<j}\{M[i,k] + M[k+1, j]+P[i-1]*P[k]*P[j]\}$

1	2	3	4	5	
0					1
	0				2
		0			3
			0		4
				0	5



We will fill the table diagonally. After following the above formula

1	2	3	4	5	
0	120	264	1080	1344	1
	0	360	1320	1350	2
		0	720	1140	3
			0	1680	4
				0	5

Manzurul Hasan

C++ code to Implement Matrix Chain Multiplication Algorithm

```
#include<iostream>
#include<limits>

using namespace std;

int MatrixChainMultiplication(int p[], int n)
{
    cout<<n<<endl;
    int m[n][n];
    int i, j, k, L, q;

    for (i=1; i<=n; i++)
        m[i][i] = 0;
    for (L=2; L<=n; L++)
    {
        for (i=1; i<=n-L+1; i++)
        {
            j = i+L-1;
            m[i][j] = INT_MAX;  //assigning to maximum value

            for (k=i; k<=j-1; k++)
            {
                q = m[i][k] + m[k+1][j] + p[i-1]*p[k]*p[j];
                if (q < m[i][j])
                {
                    m[i][j] = q;
                }
            }
        }
    }

    return m[1][n];

}

int main()
{
```

```
    int n,i;
    cout<<"Enter number of matrices\n";
    cin>>n;

    int arr[n+1];

    cout<<"Enter dimensions \n";

    for(i=0;i<=n;i++)
    {
        cout<<"Enter d"<<i<<" = ";
        cin>>arr[i];
    }

    cout<<"Minimum number of multiplications is
"<<MatrixChainMultiplication(arr, n);

    return 0;
}
```

Longest Common Subsequence

Given two sequences, find the length of longest subsequence present in both of them. A subsequence is a sequence that appears in the same relative order, but not necessarily contiguous

Examples:
LCS for input Sequences “ABCDGH” and “AEDFHR” is “ADH” of length 3
LCS for input Sequences “AGGTAB” and “GXTXAYB” is “GTAB” of length 4

			A	B	C	D	G	H
		-1	0	1	2	3	4	5
	-1	0	0	0	0	0	0	0
A	0	0	1	1	1	1	1	1
E	1	0	1	1	1	1	1	1
D	2	0	1	1	1	2	2	2
F	3	0	1	1	1	2	2	2
H	4	0	1	1	1	2	2	3
R	5	0	1	1	1	2	2	3

Designed by *Faria Nawshin* and *Md. Manzurul Hasan*

Algorithm for Longest Common Subsequence

```
 // m is the length of the first String
 // n is the length of the second String
 for(i = 0; i less than or equal to m; i++) {
   for(j = 0; j less than or equal to n; j++) {
     if ( i == 0)  { // Entries on first row
        mat[i][j] = 0;
     }
     else if ( j == 0)  { // Entries on first column
        mat[i][j] = 0;
    }
    else if (first[i] == second[j]) {  // Same character on both
strings
        mat[i][j] = mat[i-1][j-1]+1;
    }
    else if (first[i] != second[j]) {  // Different characters
on both strings
        mat[i][j] = max((mat[i-1][j]),(mat[i][j-1]));
    }
  } // end for j
 } // end for i
```

C++ Code to Implement Longest Common Subsequence

```
/* Dynamic Programming implementation of LCS problem */
#include<iostream>
#include<cstring>
#include<cstdlib>
using namespace std;

/* Returns length of LCS for X[0..m-1], Y[0..n-1] */
void lcs( char *X, char *Y, int m, int n )
{
int L[m+1][n+1];

/* Following steps build L[m+1][n+1] in bottom up fashion. Note
     that L[i][j] contains length of LCS of X[0..i-1] and
Y[0..j-1] */
for (int i=0; i<=m; i++)
{
     for (int j=0; j<=n; j++)
     {
     if (i == 0 || j == 0)
```

```
            L[i][j] = 0;
        else if (X[i-1] == Y[j-1])
            L[i][j] = L[i-1][j-1] + 1;
        else
            L[i][j] = max(L[i-1][j], L[i][j-1]);
        }
}

// Following code is used to print LCS
int index = L[m][n];

// Create a character array to store the lcs string
char lcs[index+1];
lcs[index] = ''; // Set the terminating character

// Start from the right-most-bottom-most corner and
// one by one store characters in lcs[]
int i = m, j = n;
while (i > 0 && j > 0)
{
    // If current character in X[] and Y are same, then
    // current character is part of LCS
    if (X[i-1] == Y[j-1])
    {
        lcs[index-1] = X[i-1]; // Put current character in
result
        i--; j--; index--;   // reduce values of i, j and
index
    }

    // If not same, then find the larger of two and
    // go in the direction of larger value
    else if (L[i-1][j] > L[i][j-1])
        i--;
    else
        j--;
}

// Print the lcs
cout << "LCS of " << X << " and " << Y << " is " << lcs;
}

/* Driver program to test above function */
int main()
{
char X[] = "AGGTAB";
```

```
char Y[] = "GXTXAYB";
int m = strlen(X);
int n = strlen(Y);
lcs(X, Y, m, n);
return 0;
}
```

AMERICAN INTERNATIONAL UNIVERSITY-BANGLADESH (AIUB)
FACULTY OF SCIENCE & TECHNOLOGY
DEPARTMENT OF COMPUTER SCIENCE

LAB MANUAL 05

CSC2211 Algorithms

Fall 2019-2020

Designed by *Faria Nawshin* and *Md. Manzurul Hasan*

TITLE

An Introduction to Graph and Graph Traversal Algorithms

PREREQUISITE

- Have a clear understanding of two dimensional Array and Pointer
- Should know how to pass 2D array in a function

OBJECTIVE

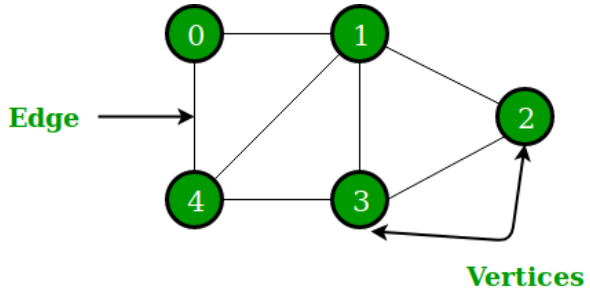
- To know about Representation of Graph
- To know about Breadth First Search (BFS)
- To know about Depth First Search (DFS)

THEORY

Graph

A Graph is a non-linear data structure consisting of nodes and edges. The nodes are sometimes also referred to as vertices and the edges are lines or arcs that connect any two nodes in the graph. More formally a Graph can be defined as,

A Graph consists of a finite set of vertices (or nodes) and set of Edges which connect a pair of nodes.



In the above Graph, the set of vertices V = {0,1,2,3,4} and the set of edges E = {01, 12, 23, 34, 04, 14, 13}.

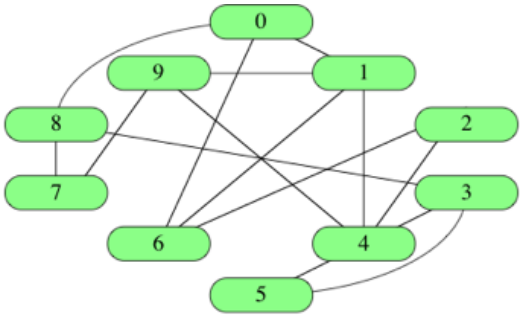
Designed by *Faria Nawshin* and *Md. Manzurul Hasan*

Graphs are used to solve many real-life problems. Graphs are used to represent networks. The networks may include paths in a city or telephone network or circuit network. Graphs are also used in social networks like linkedIn, Facebook.

Representation of Graph

There are several ways to represent graphs, each with its advantages and disadvantages. Some situations, or algorithms that we want to run with graphs as input, call for one representation, and others call for a different representation. Here, we'll see three ways to represent graphs.

1. **Vertex Lists:**
   It is common to identify vertices not by name (such as "Audrey," "Boston," or "sweater") but instead by a number. That is, we typically number the |V| |V| |V|vertical bar, V, vertical bar vertices from 0 to |V|−1 |V|-1 |V|−1vertical bar, V, vertical bar, minus, 1. Here's the social network graph with its 10 vertices identified by numbers rather than names:



   Edge lists:
   One simple way to represent a graph is just a list, or array, of |E| edges, which we call an edge list. To represent an edge, we just have an array of two vertex numbers, or an array of objects containing the vertex numbers of the vertices that the edges are incident on. If edges have weights, add either a third element to the array or more information to the object, giving the edge's weight. For example, here's how we represent an edge list for the social network graph:

   [[0,1], [0,6], [0,8], [1,4], [1,6], [1,9], [2,4], [2,6], [3,4], [3,5],

   [3,8], [4,5], [4,9], [7,8], [7,9]]

Designed by *Faria Nawshin* and *Md. Manzurul Hasan*

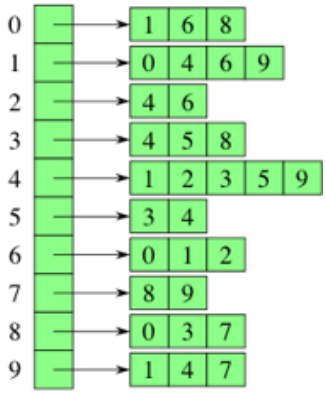
2. **Adjacency Matrices:**

   For a graph with |V| vertices, an adjacency matrix is a |V|×|V| matrix of 0s and 1s, where the entry in row *i* and column *j* is 1 if and only if the edge *(i,j)* is in the graph. If you want to indicate an edge weight, put it in the row *I*, column *j* entry, and reserve a special value (perhaps null) to indicate an absent edge. Here's the adjacency matrix for the social network graph:

	0	1	2	3	4	5	6	7	8	9
0	0	1	0	0	0	0	1	0	1	0
1	1	0	0	0	1	0	1	0	0	1
2	0	0	0	0	1	0	1	0	0	0
3	0	0	0	0	1	1	0	0	1	0
4	0	1	1	1	0	1	0	0	0	1
5	0	0	0	1	1	0	0	0	0	0
6	1	1	1	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0	1	1
8	1	0	0	1	0	0	0	1	0	0
9	0	1	0	0	1	0	0	1	0	0

3. **Adjacency List:**

   Representing a graph with adjacency lists combines adjacency matrices with edge lists. For each vertex *i* store an array of the vertices adjacent to it. We typically have an array of |V| adjacency lists, one adjacency list per vertex. Here's an adjacency-list representation of the social network graph:



C++ Code to Represent Graph using Adjacency Matrix

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

int vertArr[20][20]; //the adjacency matrix initially 0
int count = 0;

void displayMatrix(int v) {
  int i, j;
  for(i = 0; i < v; i++) {
    for(j = 0; j < v; j++) {
      cout << vertArr[i][j] << " ";
    }
    cout << endl;
  }
}

void add_edge(int u, int v) { //function to add edge into the matrix
  vertArr[u][v] = 1;
  vertArr[v][u] = 1;
}

int main() {
int v = 6; //there are 6 vertices in the graph
add_edge(0, 4);
add_edge(0, 3);
add_edge(1, 2);
add_edge(1, 4);
add_edge(1, 5);
add_edge(2, 3);
add_edge(2, 5);
add_edge(5, 3);
add_edge(5, 4);
displayMatrix(v);
}
```

Designed by *Faria Nawshin* and *Md. Manzurul Hasan*

Graph Traversal Algorithms:

Graph traversal means visiting every vertex and edge exactly once in a well-defined order. While using certain graph algorithms, you must ensure that each vertex of the graph is visited exactly once. The order in which the vertices are visited are important and may depend upon the algorithm or question that you are solving.

During a traversal, it is important that you track which vertices have been visited. The most common way of tracking vertices is to mark them.

Breadth First Search (BFS):

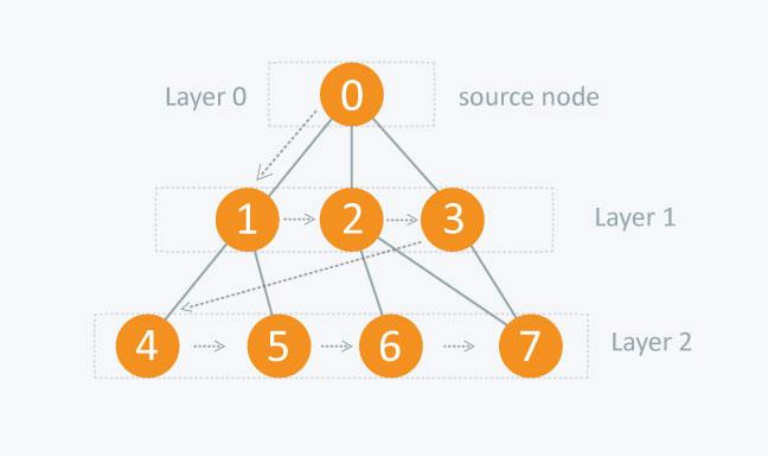
There are many ways to traverse graphs. BFS is the most commonly used approach.

BFS is a traversing algorithm where you should start traversing from a selected node (source or starting node) and traverse the graph layerwise thus exploring the neighbour nodes (nodes which are directly connected to source node). You must then move towards the next-level neighbor nodes.

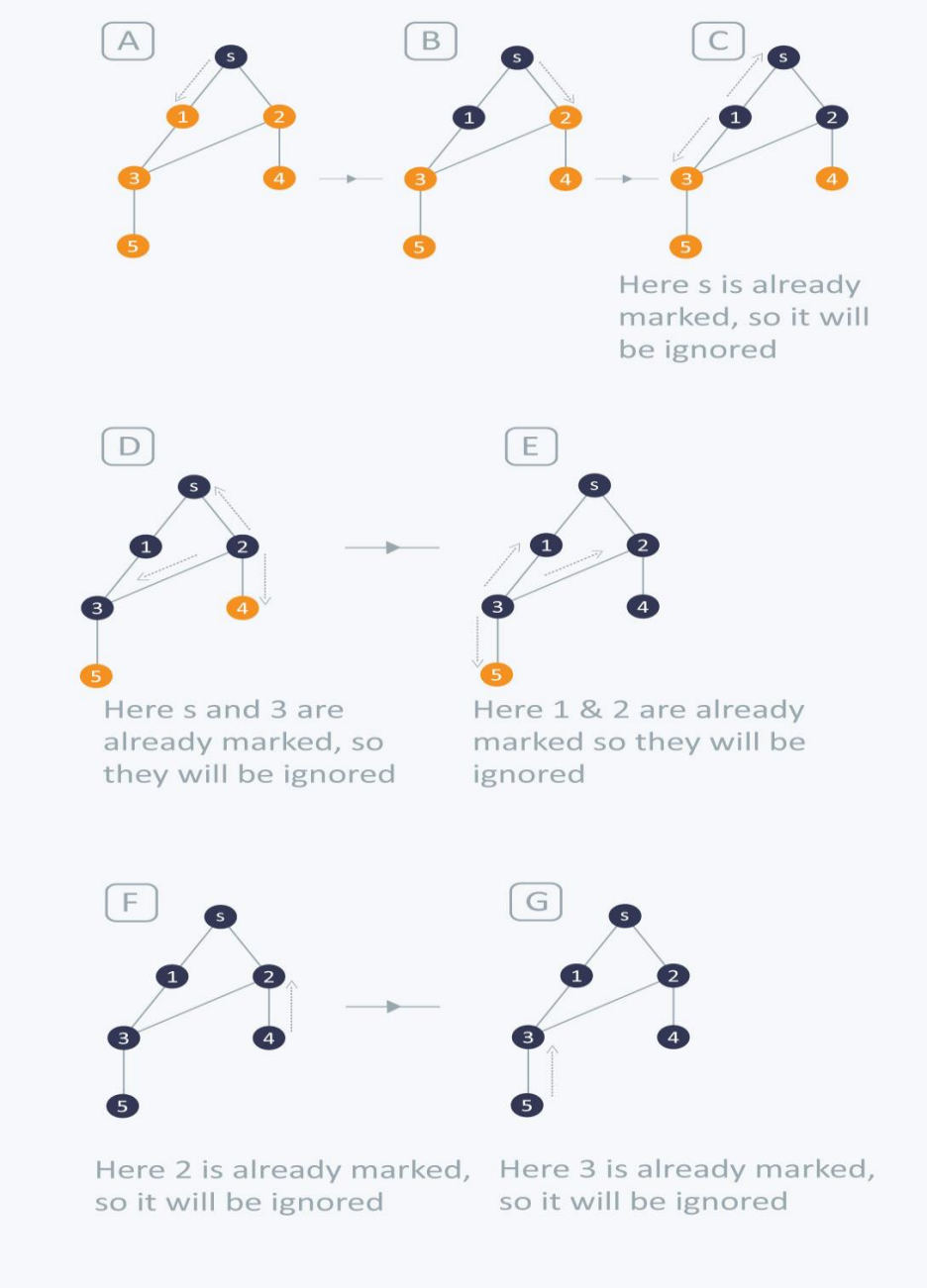
As the name BFS suggests, you are required to traverse the graph breadthwise as follows:

1. First move horizontally and visit all the nodes of the current layer
2. Move to the next layer

Consider the following diagram.



Traversing Process:



Designed by *Faria Nawshin* and *Md. Manzurul Hasan*

Pseudocode for BFS

```
BFS (G, s)          //Where G is the graph and s is the source node
      let Q be queue.
      Q.enqueue( s ) //Inserting s in queue until all its neighbour
vertices are marked.

      mark s as visited.
      while ( Q is not empty)
        //Removing that vertex from queue,whose neighbour will be
visited now
           v  =  Q.dequeue( )

          //processing all the neighbours of v
          for all neighbours w of v in Graph G
               if w is not visited
                        Q.enqueue( w ) //Stores w in Q to further
visit its neighbour
                        mark w as visited.
```

Depth First Search (DFS)

The DFS algorithm is a recursive algorithm that uses the idea of backtracking. It involves exhaustive searches of all the nodes by going ahead, if possible, else by backtracking.

Here, the word backtrack means that when you are moving forward and there are no more nodes along the current path, you move backwards on the same path to find nodes to traverse. All the nodes will be visited on the current path till all the unvisited nodes have been traversed after which the next path will be selected.

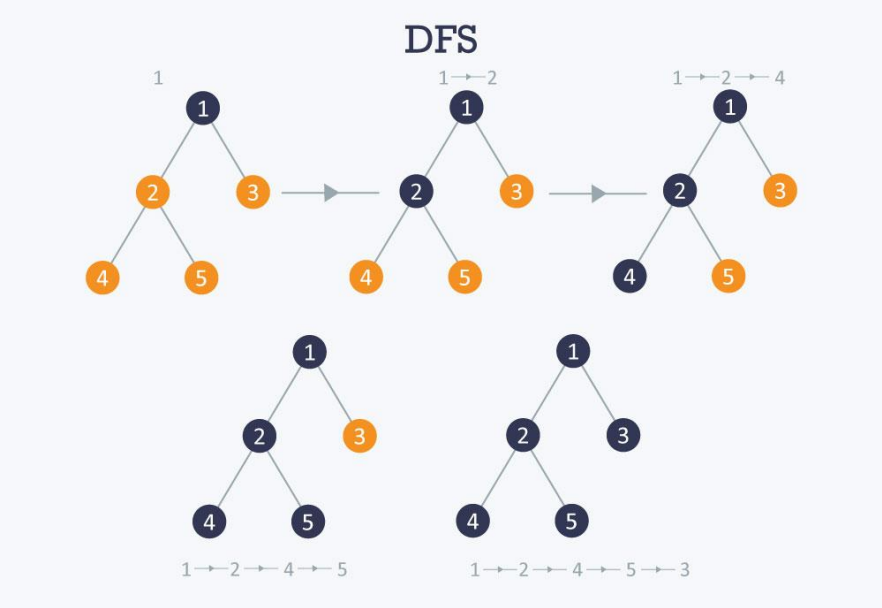
This recursive nature of DFS can be implemented using stacks. The basic idea is as follows:

Pick a starting node and push all its adjacent nodes into a stack.

Pop a node from stack to select the next node to visit and push all its adjacent nodes into a stack.

Repeat this process until the stack is empty. However, ensure that the nodes that are visited are marked. This will prevent you from visiting the same node more than once. If you do not mark the nodes that are visited and you visit the same node more than once, you may end up in an infinite loop.

Designed by *Faria Nawshin* and *Md. Manzurul Hasan*



Pseudocode for DFS

```
DFS-iterative (G, s):        //Where G is graph and s is source vertex
      let S be stack
      S.push( s )            //Inserting s in stack
      mark s as visited.
      while ( S is not empty):
          //Pop a vertex from stack to visit next
          v  =  S.top( )
         S.pop( )
         //Push all the neighbours of v in stack that are not visited
        for all neighbours w of v in Graph G:
            if w is not visited :
                     S.push( w )
                    mark w as visited
```

Designed by *Faria Nawshin* and *Md. Manzurul Hasan*

AMERICAN INTERNATIONAL UNIVERSITY-BANGLADESH (AIUB)
FACULTY OF SCIENCE & TECHNOLOGY
DEPARTMENT OF COMPUTER SCIENCE

LAB MANUAL 06

CSC2211 Algorithms

Fall 2019-2020

Designed by *Faria Nawshin* and *Md. Manzurul Hasan*

TITLE

An Introduction to Tree and Greedy Graph Algorithms

PREREQUISITE

- Have a clear understanding of two dimensional Array and Pointer
- Should know how to pass 2D array in a function

OBJECTIVE

- To know about Tree Data Structure
- To know about Minimum Spanning Tree Algorithm
- To know about Shortest Path Algorithm

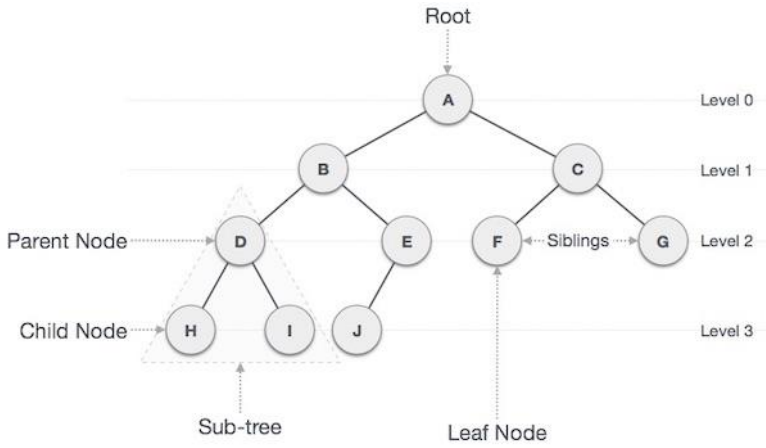
THEORY

Tree

Tree represents the nodes connected by edges. We will discuss binary tree or binary search tree specifically.

Binary Tree is a special datastructure used for data storage purposes. A binary tree has a special condition that each node can have a maximum of two children. A binary tree has the benefits of both an ordered array and a linked list as search is as quick as in a sorted array and insertion or deletion operation are as fast as in linked list.

Designed by *Faria Nawshin* and *Md. Manzurul Hasan*



Important Terms

Following are the important terms with respect to tree.

- **Path** − Path refers to the sequence of nodes along the edges of a tree.
- **Root** − The node at the top of the tree is called root. There is only one root per tree and one path from the root node to any node.
- **Parent** − Any node except the root node has one edge upward to a node called parent.
- **Child** − The node below a given node connected by its edge downward is called its child node.
- **Leaf** − The node which does not have any child node is called the leaf node.
- **Subtree** − Subtree represents the descendants of a node.
- **Visiting** − Visiting refers to checking the value of a node when control is on the node.
- **Traversing** − Traversing means passing through nodes in a specific order.
- **Levels** − Level of a node represents the generation of a node. If the root node is at level 0, then its next child node is at level 1, its grandchild is at level 2, and so on.
- **keys** − Key represents a value of a node based on which a search operation is to be carried out for a node.

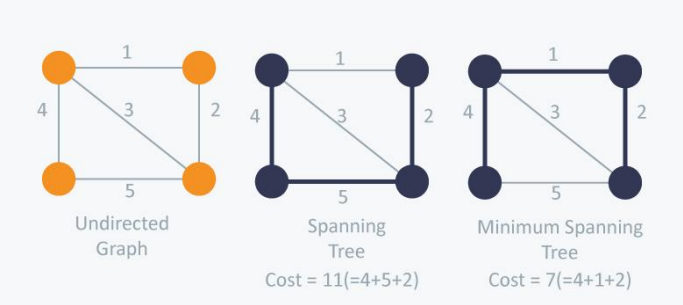
Greedy Graph Algorithm

Spanning Tree:

Given an undirected and connected graph G= (V,E), a spanning tree of the graph G is a tree that spans G (that is, it includes every vertex of G) and is a subgraph of (every edge in the tree belongs to G)

Minimum Spanning Tree:

The cost of the spanning tree is the sum of the weights of all the edges in the tree. There can be many spanning trees. Minimum spanning tree is the spanning tree where the cost is minimum among all the spanning trees. There also can be many minimum spanning trees.



There are two famous algorithms for finding the Minimum Spanning Tree:

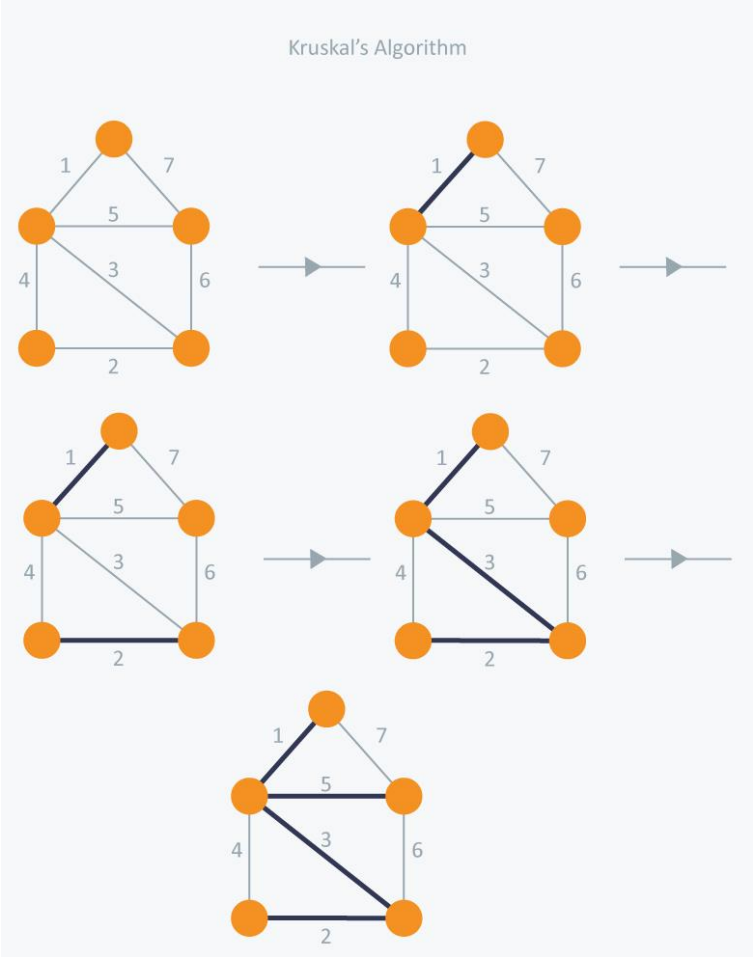
Kruskal's Algorithm:

Kruskal's Algorithm builds the spanning tree by adding edges one by one into a growing spanning tree. Kruskal's algorithm follows greedy approach as in each iteration it finds an edge which has least weight and add it to the growing spanning tree.

Algorithm Steps:

- Sort the graph edges with respect to their weights.
- Start adding edges to the MST from the edge with the smallest weight until the edge of the largest weight.
- Only add edges which doesn't form a cycle , edges which connect only disconnected components.

Designed by *Faria Nawshin* and *Md. Manzurul Hasan*



In Kruskal's algorithm, at each iteration we will select the edge with the lowest weight. So, we will start with the lowest weighted edge first i.e., the edges with weight 1. After that we will select the second lowest weighted edge i.e., edge with weight 2. Notice these two edges are totally disjoint. Now, the next edge will be the third lowest weighted edge i.e., edge with weight 3, which connects the two disjoint pieces of the graph. Now, we are not allowed to pick the edge with weight 4, that will create a cycle and we can't have any cycles. So we will select the fifth lowest weighted edge i.e., edge with weight 5. Now the other two edges will create cycles so we will ignore them. In the end, we end up with a minimum spanning tree with total cost 11 (= 1 + 2 + 3 + 5).

Pseudocode for Kruskal's Algorithm:

```
KRUSKAL(G):
A = ∅
For each vertex v ∈ G.V:
    MAKE-SET(v)
```

Designed by *Faria Nawshin* and *Md. Manzurul Hasan*

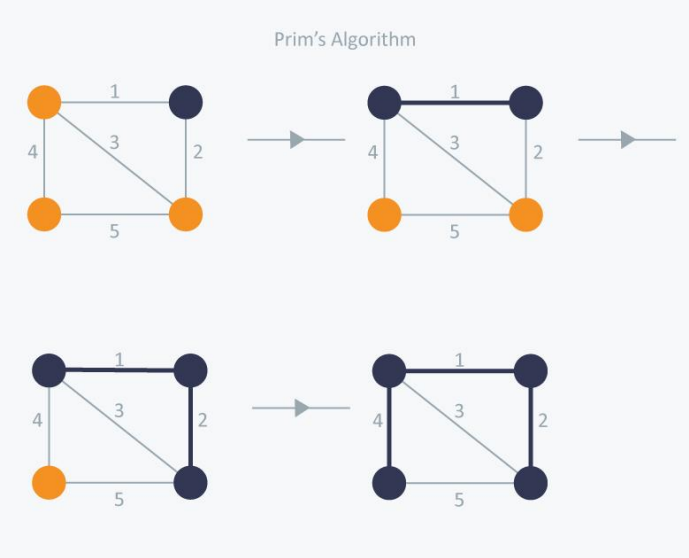
```
For each edge (u, v) ∈ G.E ordered by increasing order by weight(u,
v):
    if FIND-SET(u) ≠ FIND-SET(v):
    A = A ∪ {(u, v)}
    UNION(u, v)
return A
```

Prim's Algorithm:

Prim's Algorithm also use Greedy approach to find the minimum spanning tree. In Prim's Algorithm we grow the spanning tree from a starting position. Unlike an edge in Kruskal's, we add vertex to the growing spanning tree in Prim's.

Algorithm Steps:

- Maintain two disjoint sets of vertices. One containing vertices that are in the growing spanning tree and other that are not in the growing spanning tree.
- Select the cheapest vertex that is connected to the growing spanning tree and is not in the growing spanning tree and add it into the growing spanning tree. This can be done using Priority Queues. Insert the vertices, that are connected to growing spanning tree, into the Priority Queue.
- Check for cycles. To do that, mark the nodes which have been already selected and insert only those nodes in the Priority Queue that are not marked.



Designed by *Faria Nawshin* and *Md. Manzurul Hasan*

In Prim's Algorithm, we will start with an arbitrary node (it doesn't matter which one) and mark it. In each iteration we will mark a new vertex that is adjacent to the one that we have already marked. As a greedy algorithm, Prim's algorithm will select the cheapest edge and mark the vertex. So we will simply choose the edge with weight 1. In the next iteration we have three options, edges with weight 2, 3 and 4. So, we will select the edge with weight 2 and mark the vertex. Now again we have three options, edges with weight 3, 4 and 5. But we can't choose edge with weight 3 as it is creating a cycle. So we will select the edge with weight 4 and we end up with the minimum spanning tree of total cost 7 (= 1 + 2 +4).

Pseudocode for Prim's Algorithm:

```
T = ∅;
U = { 1 };
while (U ≠ V)
    let (u, v) be the lowest cost edge such that u ∈ U and v ∈ V - U;
    T = T ∪ {(u, v)}
    U = U ∪ {v}
```

Shortest Path Algorithm

The shortest path problem is about finding a path between 2 vertices in a graph such that the total sum of the edges weights is minimum.

Single Source Shortest Path Algorithm

Bellman Ford's Algorithm:

Bellman Ford's algorithm is used to find the shortest paths from the source vertex to all other vertices in a weighted graph. It depends on the following concept: Shortest path contains at most edges, because the shortest path couldn't have a cycle.

There is no need to pass a vertex again, because the shortest path to all other vertices could be found without the need for a second visit for any vertices.

Algorithm Steps:

- The outer loop traverses from 0 : n - 1.
- Loop over all edges, check if the next node distance > current node distance + edge weight, in this case update the next node distance to "current node distance + edge weight".

This algorithm depends on the relaxation principle where the shortest distance for all vertices is gradually replaced by more accurate values until eventually reaching the optimum solution. In the beginning all vertices have a distance of "Infinity", but only the distance of the source vertex = 0, then update all the connected vertices with the new distances (source vertex distance + edge weights), then apply the same concept for the new vertices with new distances and so on.

Dijkstra's Algorithm:

Dijkstra's algorithm has many variants but the most common one is to find the shortest paths from the source vertex to all other vertices in the graph.

Algorithm Steps:

- Set all vertices distances = infinity except for the source vertex, set the source distance = 0.
- Push the source vertex in a min-priority queue in the form (distance , vertex), as the comparison in the min-priority queue will be according to vertices distances.

Designed by *Faria Nawshin* and *Md. Manzurul Hasan*

- Pop the vertex with the minimum distance from the priority queue (at first the popped vertex = source).
- Update the distances of the connected vertices to the popped vertex in case of "current vertex distance + edge weight < next vertex distance", then push the vertex with the new distance to the priority queue.
- If the popped vertex is visited before, just continue without using it.
- Apply the same algorithm again until the priority queue is empty.

All-Pair Shortest Path Algorithm

Floyd–Warshall's Algorithm

Floyd–Warshall's Algorithm is used to find the shortest paths between between all pairs of vertices in a graph, where each edge in the graph has a weight which is positive or negative. The biggest advantage of using this algorithm is that all the shortest distances between any 2 vertices could be calculated in O (V^3), where V is the number of vertices in a graph.

The Algorithm Steps:

For a graph with N vertices:

- Initialize the shortest paths between any 2 vertices with Infinity.
- Find all pair shortest paths that use 0 intermediate vertices, then find the shortest paths that use 1 intermediate vertex and so on.. until using all N vertices as intermediate nodes.
- Minimize the shortest paths between any 2 pairs in the previous operation.
- For any 2 vertices *(i,j)* , one should actually minimize the distances between this pair using the first *K* nodes, so the shortest path will be:

  `min(dist [i] [k] + dist [k] [j] , dist [i] [j])`

- `dist [i][k]` represents the shortest path that only uses the first *K* vertices, `dist [k][j]` represents the shortest path between the `pair k,j` . As the shortest path will be a concatenation of the shortest path `from i to k`, then `from k to j` .