

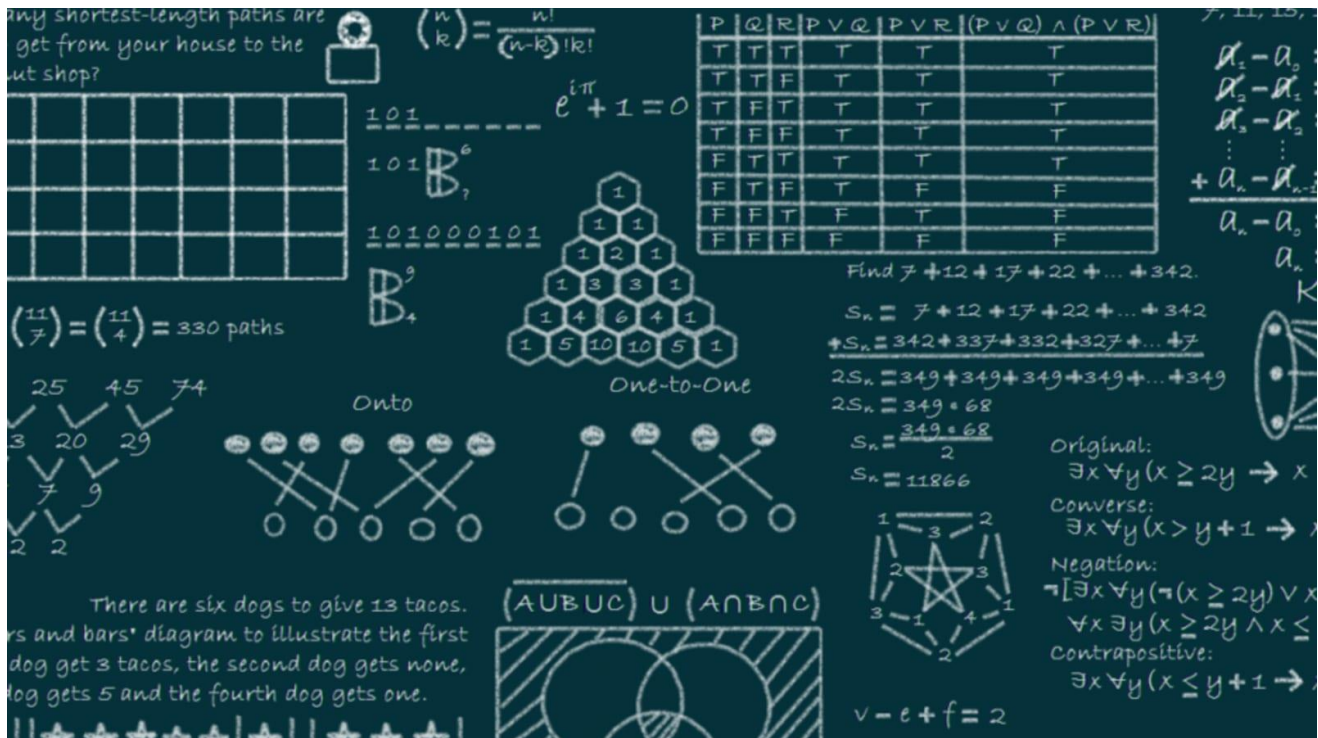
DATA STRUCTURES AND ALGORITHMS

(SOS- 2021)

Amruta Mahendra Parulekar

20D070009

Mentor:Krushnakant



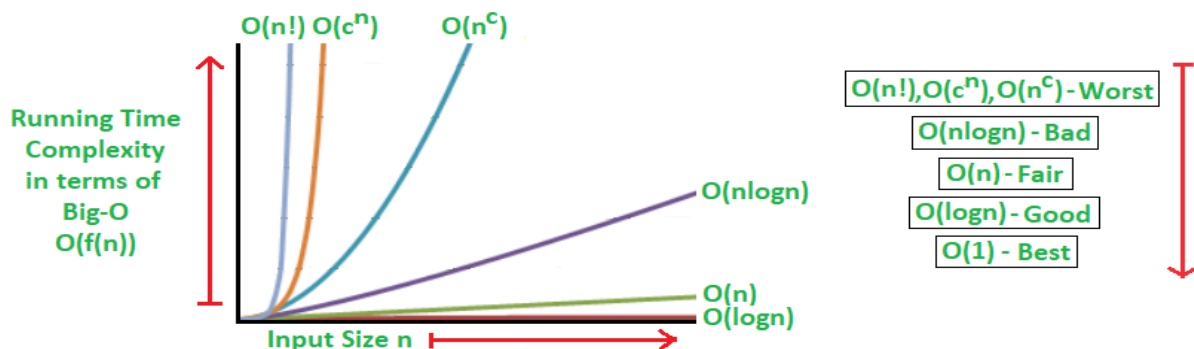
CONTENTS:

- 1. ARRAYS**
- 2. STRINGS**
- 3. MATRICES**
- 4. LISTS**
- 5. STACKS**
- 6. QUEUES**
- 7. TREES**
- 8. HEAPS**
- 9. HASH TABLES**
- 10. SORTING**
- 11. RECURSION**
- 12. GRAPHING ALGORITHMS**
- 13. NUMERICAL ALGORITHMS**
- 14. DYNAMIC PROGRAMMING**

NOTE:

THE BIG O NOTATION:

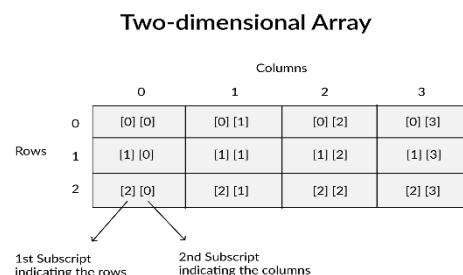
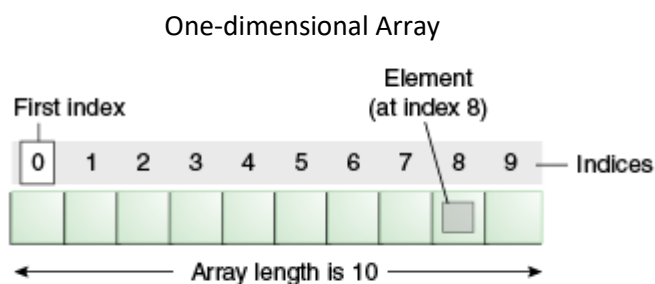
- The big O notation is used to classify algorithms according to how their run time or space requirements grow as the input size grows.
- Basically, this asymptotic notation is used to measure and compare the worst-case scenarios of algorithms theoretically.
- The general step wise procedure for Big-O runtime analysis is as follows:
 1. Figure out what the input is and what n represents.
 2. Express the maximum number of operations, the algorithm performs in terms of n.
 3. Eliminate all excluding the highest order terms.
 4. Remove all the constant factors.



1. ARRAYS

About Arrays

- An array is a collection of data of the same datatype referenced by a common name.
- It consists of adjacent memory locations
- The individual data elements are accessed using indices or subscripts, with the first element having index zero.
- Arrays can be one dimensional, two dimensional or multi-dimensional.



Using Arrays in programs

- **Declaration:** datatype arrayname [arraylength];
- **Initialization:** datatype arrayname []={data1,data2,...datan} ; or
datatype arrayname [arraylength]={data1,data2,...datan} ; where $n \leq \text{arraylength}$
- **Accessing data:** arrayname[index]; where $0 \leq \text{index} < \text{arraylength}$ (index out of range gives error)

Computer's interpretation of arrays

- When we declare an array, a block of memory of length $\text{elemtypeVariableSize} * \text{arraylength}$ is created.
- Arrayname gives the starting address of the zeroeth element which is also the address of the allocated block. (address of first byte)
- Type of arrayname=elemtype* and Type of arrayname[i]=elemtype.
- [] is a binary operator and arrayname, index are operands. Arrayname[index] gives variable of type arrayname stored at $\text{arrayname} + \text{elemtypeVariableSize} * \text{index}$.
- To pass an array to a function, pass the address of the first element and the arraylength so that operations occur on the original array.

Character arrays

- They store character strings, which are terminated with a null character(\0) and the remaining space has garbage values.
- The null character or sentinel has ASCII value 0.
- If we read in a string, it gets automatically terminated with a null byte
- If we print out a string, it will get printed till the null byte.

2D arrays

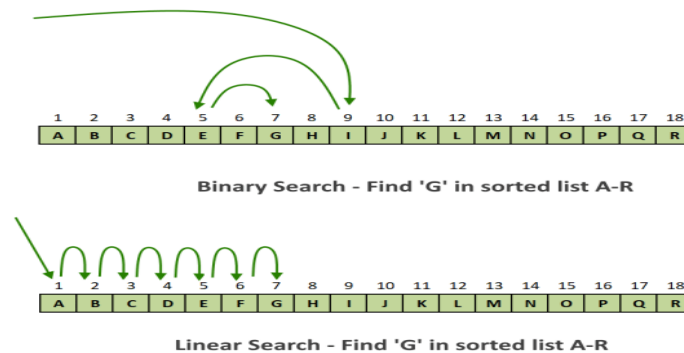
- Datatype arrayname [row number][column number(fixed)];
- Datatype arrayname[m][n]={ { n data } { n data } ... m groups};
- They may be stored in row major form or column major form

Static vs dynamic arrays

- **Static arrays** are allocated memory at compile time and the memory is allocated on the stack. They have a fixed size
- **dynamic arrays** are allocated memory at the runtime and the memory is allocated from heap. They can increase their size when an insertion occurs and the array is full. We use the **new** keyword for these arrays

Searching an array

- **Linear/sequential search:** In this search technique, we compare elements of the array one by one with the key element that we are looking for. It is simple, but time consuming so preferred for smaller arrays. Best case occurs when the key element is at the first position of the array.
- **Binary search:** In this search technique, we divide the array into two halves at each level and look for the key element in one of the two halves and the other half is rejected. This algorithm requires a sorted array. Best case occurs when the key element is in the middle of the array.



Formula for n- dimensional arrays: A[d1][d2][d3][d4]

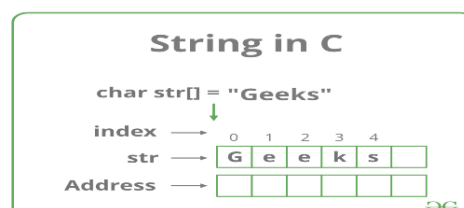
- **Row major form:**
$$\text{Address}(A[i_1][i_2][i_3][i_4]) = \text{Lo} + (i_1 * d_2 * d_3 * d_4 + i_2 * d_3 * d_4 + i_3 * d_4 + i_4) * \text{datatype size}$$
- **Column major form:**
$$\text{Address}(A[i_1][i_2][i_3][i_4]) = \text{Lo} + (i_4 * d_1 * d_2 * d_3 + i_3 * d_1 * d_2 + i_2 * d_1 + i_1) * \text{datatype size}$$

Operations on arrays

- We can code `get()`, `set()`, `max()`, `min()`, `insert()`, `delete()` etc functions for arrays.

2. STRINGS

- A string is a collection of characters which can include alphabets, both in upper and lower case, numbers, blanks and any special character.
- Strings are enclosed in double quotes and terminated with a null byte.
- Characters are represented by values called ASCII codes or unicodes in the computer memory. (a=97; A=65)



Some usage of strings

- Include the string class to use strings.

- **Declaration and initialization**

string v="abcdab"; // constructor

string w(v); // constructor

string x=v+w; //concatenation

V[2]=v[3]; //indexing

String functions

Use the dot operator to use these functions.

- **char* strcpy (char*dest, const char*src);**

char* strncpy (char*dest, const char* src, size_tn);

It copies the string pointed to by src, including the terminating null byte to the buffer pointed to by dest. It assumes that dest has sufficient memory. The second function copies atmost n bytes. If no null byte among these, dest might not be null terminated.

- **Unsigned int strlen (const char*s);**

Calculates length of the string pointed to by s, excluding the null terminated byte

- **Int strcmp (const char*s1, const char*s2);**

Int strncmp(const char* s1, const char* s2, size_tn);

It compares the two strings s1 and s2. It returns an integer <,>= 0 if s1 is found to be <,> or equal to s2. The second function compares the first n bytes of s1 and s2.

- **Int v.find(str);**

Int v.find(str,i);

It returns the position of occurrence of str in v. The second function checks from index i.

- **V.substr(i);**

V.substr(l,n);

It returns a substring of v starting from index i. The second function returns a substring of length n

3. Matrices

- A matrix is a two-dimensional data structure and all of its elements are of the same type.
- Matrix [i,j] can be mapped to array A[k] by certain formulae.

Types of matrices

- **Diagonal matrix:** $k=i-1$

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 3 \end{bmatrix}$$

- **Triangular matrix:** $k=i*(i-1)/2 + j-1$ (row major mapping)

$$\begin{bmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ 0 & & & \end{bmatrix}$$

Upper Triangular Matrix

$$\begin{bmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ 0 & & & \end{bmatrix}$$

Lower Triangular Matrix

$$k = (j-1)*n - (j-2)*(j-1)/2 + j-i \text{ (column major mapping)}$$

$$k = i*(i-1)/2 + j-1 \text{ (row major mapping)}$$

$$k = (j-1)*n - (j-2)*(j-1)/2 + j-i \text{ (column major mapping)}$$

- **Symmetric matrix:**

$$k = i*(i-1)/2 + j-1 \text{ (row major mapping)}$$

$$k = (j-1)*n - (j-2)*(j-1)/2 + j-i \text{ (column major mapping)}$$

$$k = i*(i-1)/2 + j-1 \text{ (row major mapping)}$$

$$k = (j-1)*n - (j-2)*(j-1)/2 + j-i \text{ (column major mapping)}$$

$$M[i,j] = M[j,i]$$

Symmetric matrix & Skew Symmetric Matrix	
• Symmetric: $A^T = A$.	
• Skew-symmetric: $A^T = -A$.	
• Examples:	
$\begin{bmatrix} 1 & 1 & -1 \\ 1 & 2 & 0 \\ -1 & 0 & 5 \end{bmatrix}$ symmetric	$\begin{bmatrix} 0 & 1 & -2 \\ -1 & 0 & 3 \\ 2 & -3 & 0 \end{bmatrix}$ skew-symmetric

- **Tridiagonal matrix:** case 1: $i \leq j$; $k = j-1$

$$f_n = \begin{bmatrix} a_1 & b_1 & & & \\ c_1 & a_2 & b_2 & & \\ & c_2 & \ddots & \ddots & \\ & & \ddots & \ddots & b_{n-1} \\ & & & c_{n-1} & a_n \end{bmatrix}$$

$$\text{case 2: } i > j; k = n + i - j - 1$$

(mapped as lowerdiag middlediag topdiag)

- **Toeplitz matrix:** case 1: $i-j=1$; $k = i-1$

$$\begin{bmatrix} a & b & c & d & e \\ f & a & b & c & d \\ g & f & a & b & c \\ h & g & f & a & b \\ i & h & g & f & a \end{bmatrix}$$

$$\text{case 2: } i-j=0; k = n-1+i-1$$

$$\text{case 3: } i-j=-1; k = 2n-1$$

(mapped as abcdefghi)

- **Sparse matrix:**

They have very few non zero elements.

They may be represented in:

a) 3 column representation

Rows	Columns	Values
5	6	6
0	4	9
1	1	8
2	0	4
2	3	2
3	5	5
4	2	2

b) Compressed sparse rows representation

We use 3 arrays, one consisting of the values of the elements, the second one consisting of the cumulative data of the number of elements per row and the third one consisting of column numbers of elements.

c) Array based representation

We provide the number of rows, columns and elements in the structure definition along with an array that gives us the indices and values of individual elements.

4. LISTS

A linked list is a sequence of data structures, which are connected together via links.

Types of Linked Lists

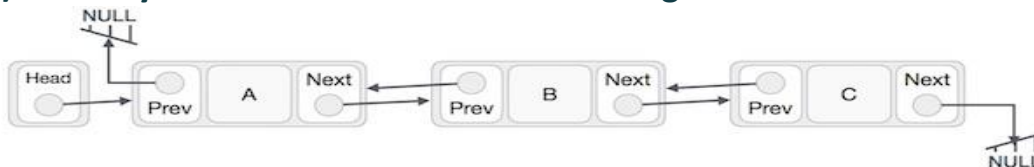
a) **Simple Linked List** – Item navigation is forward only.



○ Basic Operations

- **Insertion** – Adds an element at the beginning of the list.
- **Deletion** – Deletes an element at the beginning of the list.
- **Display** – Displays the complete list.
- **Search** – Searches an element using the given key.
- **Delete** – Deletes an element using the given key

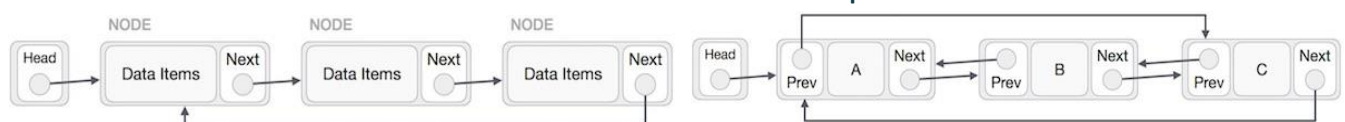
b) **Doubly Linked List** – Items can be navigated forward and backward.



○ Basic Operations

- **Insertion** – Adds an element at the beginning of the list.
- **Deletion** – Deletes an element at the beginning of the list.
- **Insert Last** – Adds an element at the end of the list.
- **Delete Last** – Deletes an element from the end of the list.
- **Insert After** – Adds an element after an item of the list.
- **Delete** – Deletes an element from the list using the key.
- **Display forward** – Displays the complete list in a forward manner.
- **Display backward** – Displays the complete list in a backward manner.

c) **Circular Linked List** – Last item contains link of the first element as next and the first element has a link to the last element as previous.



Simple linked list as circular linked list

Doubly linked list as circular linked list

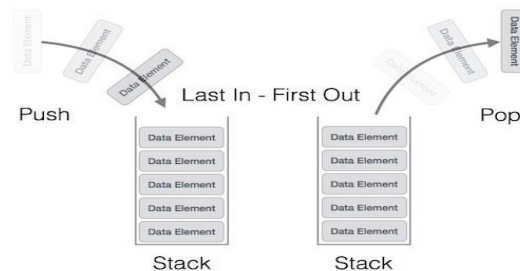
○ Basic Operations

- **insert** – Inserts an element at the start of the list.

- **delete** – Deletes an element from the start of the list.
- **display** – Displays the list.

5. STACKS

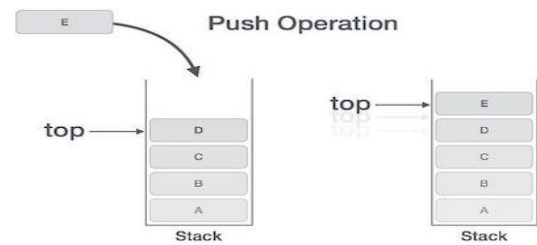
- A stack is an Abstract Data Type (ADT) that allows all data operations at one end only. At any given time, we can only access the top element of a stack
- This feature makes it Last-in-first-out data structure. Here, the element which is placed (inserted or added) last, is accessed first.
- A stack can be implemented by means of Array, Structure, Pointer, and Linked List.
- Stack can either be a fixed size one or it may have a sense of dynamic resizing.



Basic Operations

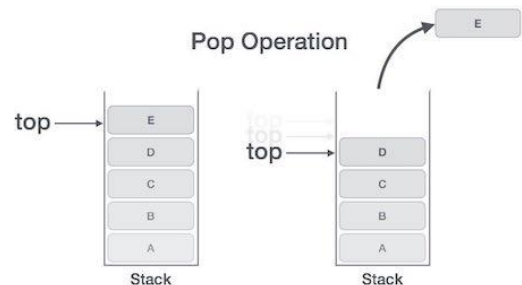
- Stack operations may involve initializing the stack, using it and de-initializing it.
- **push()** – Pushing (storing) an element on the stack.

```
void push(int data) {
    if(!isFull()) {
        top = top + 1;
        stack[top] = data;
    } else {
        printf("Could not insert data, Stack is full.\n");
    }
}
```



- **pop()** – Removing (accessing) an element from the stack.

```
int pop(int data) {
    if(!isEmpty()) {
        data = stack[top];
        top = top - 1;
        return data;
    } else {
        printf("Could not retrieve data, Stack is empty.\n");
    }
}
```



- **peek()** – get the top data element of the stack, without removing it.

```
int peek() {
```

```
    return stack[top];
```

```
}
```

- **isFull()** – check if stack is full.

```
bool isfull() {
```

```
    if(top == MAXSIZE)
```

```
        return true;
```

```
    else
```

```
        return false;
```

```
}
```

- **isEmpty()** – check if stack is empty.

```
bool isempty() {
```

```
    if(top == -1)
```

```
        return true;
```

```
    else
```

```
        return false;
```

```
}
```

- At all times, we maintain a pointer to the last pushed data on the stack. As this pointer always represents the top of the stack, hence named **top**, initialised at -1. The **top** pointer provides top value of the stack without actually removing it.

6. QUEUES

- Queue is an abstract data structure, open at both its ends.
- One end is always used to insert data (enqueue) and the other is used to remove data (dequeue).
- Queue follows First-In-First-Out methodology, i.e., the data item stored first will be accessed first.
- A queue can be implemented using Arrays, Linked-lists, Pointers and Structures.



Basic Operations

- Queue operations may involve initializing or defining the queue, utilizing it, and then completely erasing it from the memory
- **enqueue()** – add (store) an item to the queue.

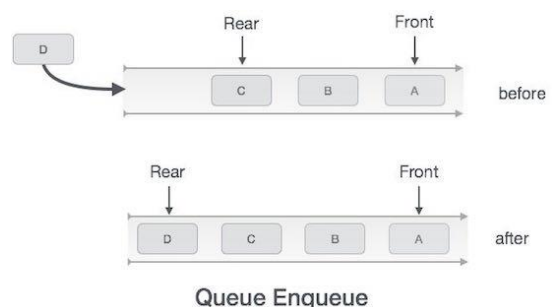
```
int enqueue(int data){
```

```
    if(isfull())
```

```
        return 0;
```

```
    rear = rear + 1;
```

```
    queue[rear] = data;
```

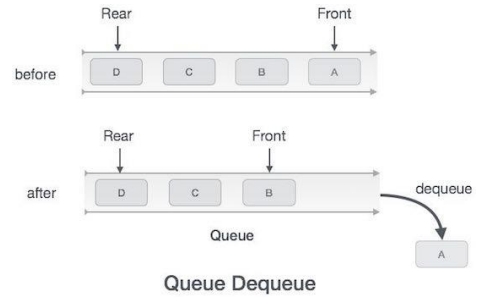


```
    return 1;
```

```
}
```

- **dequeue()** – remove (access) an item from the queue.

```
int dequeue() {  
    if(isempty())  
        return 0;  
    int data = queue[front];  
    front = front + 1;  
    return data;  
}
```



- **peek()** – Gets the element at the front of the queue without removing it.

```
int peek() {  
    return queue[front];  
}
```

- **isfull()** – Checks if the queue is full.

```
bool isfull() {  
    if(rear == MAXSIZE - 1)  
        return true;  
    else  
        return false;  
}
```

- **isempty()** – Checks if the queue is empty.

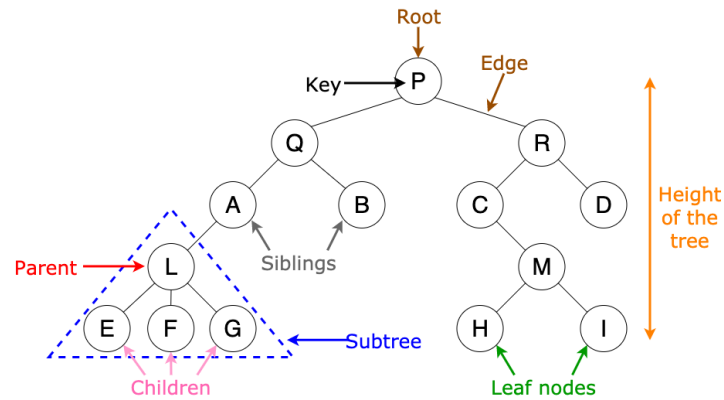
```
bool isempty() {  
    if(front < 0 || front > rear)  
        return true;  
    else  
        return false;  
}
```

- In queue, we always dequeue (or access) data, pointed by **front** pointer and while enqueueing (or storing) data in the queue we take help of **rear** pointer.

7. TREES

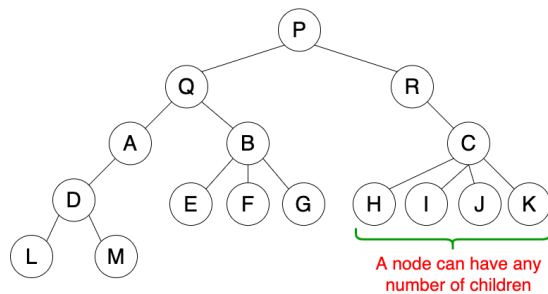
Properties of a Tree

- A tree can contain no nodes or it can contain one special node called the **root** with zero or more subtrees.
- Every edge of the tree is directly or indirectly originated from the root.
- Every child has only one parent, but one parent can have many children.



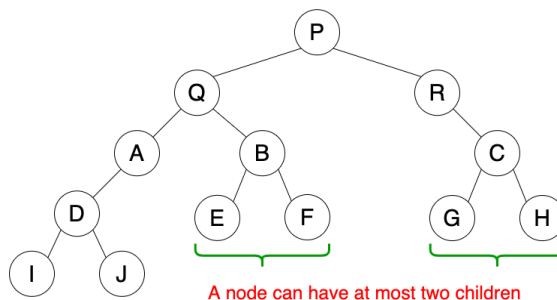
a) General Tree

1. Follow properties of a tree.
2. A node can have any number of children.



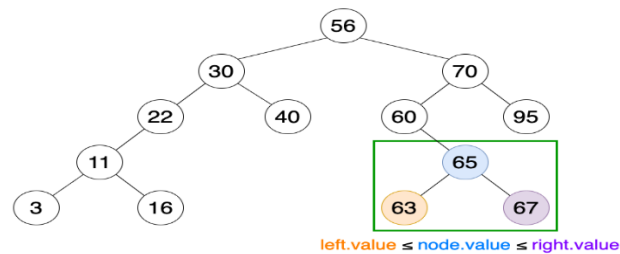
b) Binary Tree

1. Follow properties of a tree.
2. A node can have at most two child nodes (children).
3. These two child nodes are known as the **left child** and **right child**.



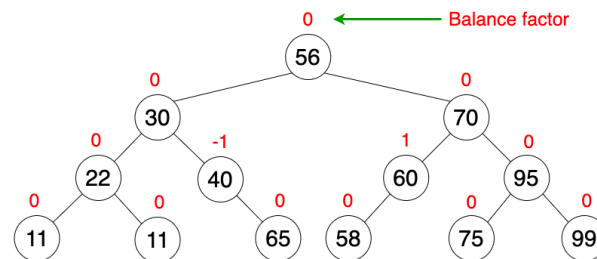
c) Binary Search Tree

1. Follow properties of a binary tree.
2. Has a unique property known as the **binary-search-tree property**. This property states that the value (or key) of the left child of a given node should be less than or equal to the parent value and the value of the right child should be greater than or equal to the parent value.



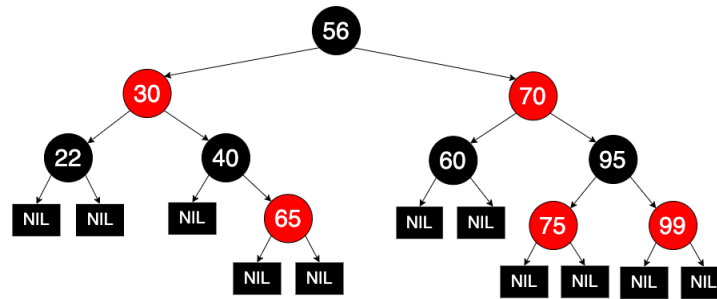
d) AVL tree

1. Follow properties of binary search trees.
2. Self-balancing.(Automatically balances its own height)
3. Each node stores a value called a **balance factor** which is the difference in height between its left subtree and right subtree.
4. All the nodes must have a balance factor of -1, 0 or 1.
5. After performing insertions or deletions, if there is at least one node that does not have a balance factor of -1, 0 or 1 then rotations should be performed to balance the tree (self-balancing).



e) Red-black tree.

1. Follow properties of binary search trees.
2. Self-balancing.
3. Each node is either red or black.
4. The colours of the nodes are used to make sure that the tree remains approximately balanced during insertions and deletions
5. The root is black (sometimes omitted).
6. All leaves (denoted as NIL) are black.
7. If a node is red, then both its children are black.
8. Every path from a given node to any of its leaf nodes must go through the same number of black nodes.



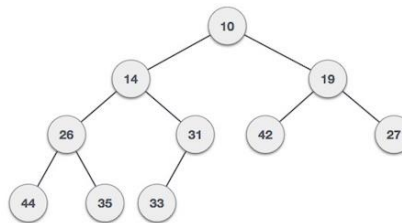
8. HEAPS

- Heap is a special case of balanced binary tree data structure where the root-node key is compared with its children and arranged accordingly.

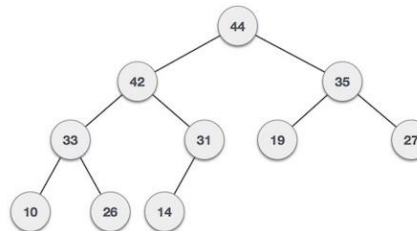
Types of heaps

For Input → 35 33 42 10 14 19 27 44 26 31

- A) Min-Heap** – Where the value of the root node is less than or equal to either of its children.



- B) Max-Heap** – Where the value of the root node is greater than or equal to either of its children.



Heap Construction Algorithm

Insert one element at a time. At any point of time, heap must maintain its property. While insertion, we also assume that we are inserting a node in an already heapified tree.

Step 1 – Create a new node at the end of heap.

Step 2 – Assign new value to the node.

Step 3 – Compare the value of this child node with its parent.

Step 4 – If value of parent is (less than for max heap) (greater than for min heap) child, then swap them.

Step 5 – Repeat step 3 & 4 until Heap property holds.

Heap Deletion Algorithm

Deletion in Max or Min Heap always happens at the root to remove the Maximum or minimum value.

Step 1 – Remove root node.

Step 2 – Move the last element of last level to root.

Step 3 – Compare the value of this child node with its parent.

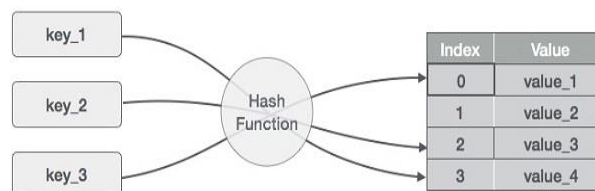
Step 4 – If value of parent is (less than for max heap) (greater than for min heap) . child, then swap them.

Step 5 – Repeat step 3 & 4 until Heap property holds.

9. HASH TABLES

- Hash Table is a data structure which stores data in an associative manner.
- In a hash table, data is stored in an array format, where each data value has its own unique index value. Access of data becomes very fast if we know the index of the desired data.
- Insertion and search operations are very fast irrespective of the size of the data.
- struct DataItem

```
{  
    int data;  
    int key;  
};
```



Hashing

- Hash Table uses hash technique to generate an index where an element is to be inserted or is to be located from.
- Hashing is a technique to convert a range of key values into a range of indexes of an array.
- We use modulo operator to get a range of key values.
- int hashCode(int key)
{
 return key % SIZE;
}
- It may happen that the hashing technique is used to create an already used index of the array. In such a case, we can search the next empty location in the array by looking into the next cell until we find an empty cell. This technique is called **linear probing**.

Basic Operations

- **Search** – Searches an element in a hash table. Compute the hash code of the key passed and locate the element using that hash code as index in the array. Use linear probing to get the element ahead if the element is not found at the computed hash code.

```
struct Dataltem *search(int key)
{
    int hashIndex = hashCode(key);           //get the hash
    while(hashArray[hashIndex] != NULL)      //move in array until an empty
    {
        if(hashArray[hashIndex]->key == key)
            return hashArray[hashIndex];
        ++hashIndex;                          //go to next cell
        hashIndex %= SIZE;                    //wrap around the table
    }
    return NULL;
}
```

- **Insert** – inserts an element in a hash table. Compute the hash code of the key passed and locate the index using that hash code as an index in the array. Use linear probing for empty location, if an element is found at the computed hash code.

```
void insert(int key,int data)
{
    struct Dataltem *item = (struct Dataltem*) malloc(sizeof(struct Dataltem));
    item->data = data;
    item->key = key;
    int hashIndex = hashCode(key);           //get the hash
    while(hashArray[hashIndex] != NULL && hashArray[hashIndex]->key != -1)
        //move in array until an empty or deleted cell
    {
        ++hashIndex;                          //go to next cell
        hashIndex %= SIZE;                    //wrap around the table
    }
    hashArray[hashIndex] = item;
}
```


- **Delete** – Deletes an element from a hash table. Compute the hash code of the key passed and locate the index using that hash code as an index in the array. Use linear probing to get the element ahead if an element is not found at the computed hash code. When found, store a dummy item there to keep the performance of the hash table intact.

```
struct Dataltem* delete(struct Dataltem* item)
{
    int key = item->key;
    int hashIndex = hashCode(key);           //get the hash
    while(hashArray[hashIndex] !=NULL)      //move in array until an empty
    {
        if(hashArray[hashIndex]->key == key)
        {
            struct Dataltem* temp = hashArray[hashIndex];
            hashArray[hashIndex] = dummyItem;
            //assign a dummy item at deleted position

            return temp;
        }
        ++hashIndex;                        //go to next cell
        hashIndex %= SIZE;                  //wrap around the table
    }
    return NULL;
}
```

10. **SORTING**

- It is the process of arranging data in ascending or descending order

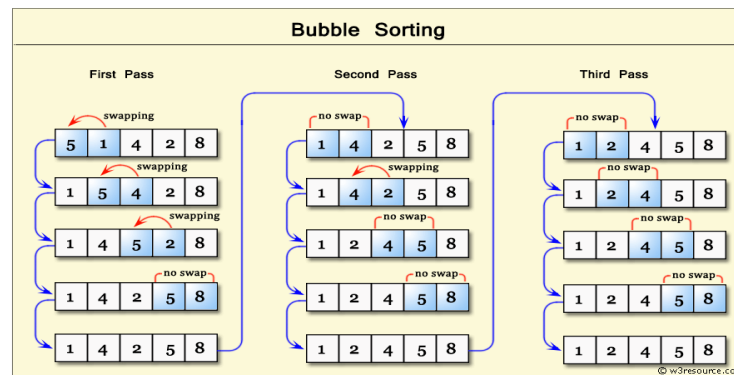
Time Complexities of Sorting Algorithms:

Algorithm	Best	Average	Worst
Quick Sort	$\Omega(n \log(n))$	$\Theta(n \log(n))$	$O(n^2)$
Bubble Sort	$\Omega(n)$	$\Theta(n^2)$	$O(n^2)$
Merge Sort	$\Omega(n \log(n))$	$\Theta(n \log(n))$	$O(n \log(n))$
Insertion Sort	$\Omega(n)$	$\Theta(n^2)$	$O(n^2)$

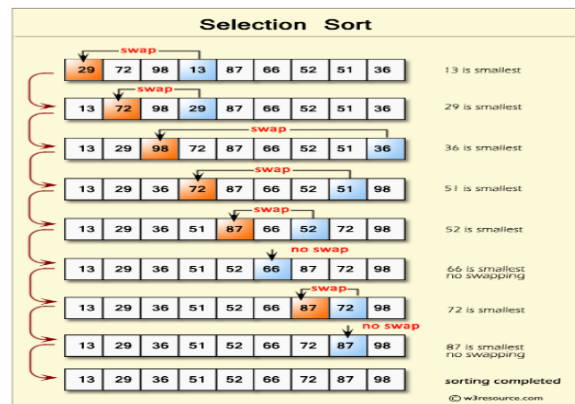
Algorithm	Best	Average	Worst
Selection Sort	$\Omega(n^2)$	$\Theta(n^2)$	$O(n^2)$

Types of sorting algorithms

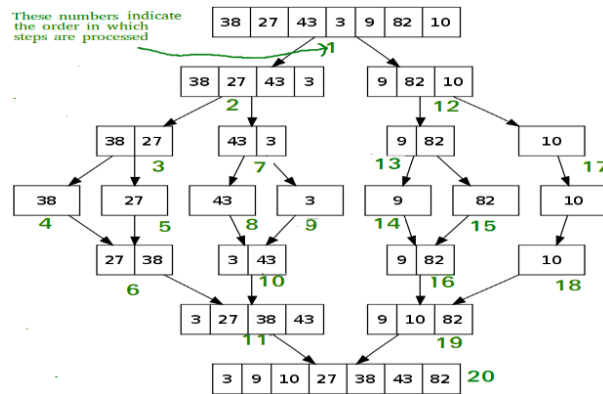
- Bubble sort:** In this technique, after one pass, the largest element is placed in the correct location. Every pass is reduced by one check because the largest element, once placed correctly, need not be included in the next pass. It requires $n-1$ passes to sort an array of n elements. In each pass, adjacent elements are compared and if they are not in order, they are swapped. In the k th pass we have $n-k$ comparisons.



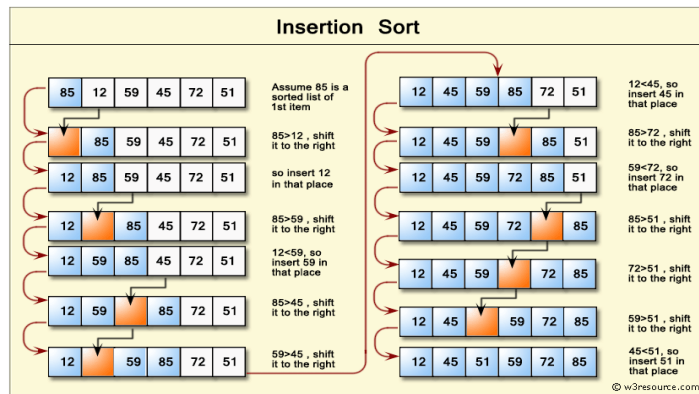
- Selection sort:** In this technique, we search for the smallest element in each pass and swap it with the appropriate position. It requires $n-1$ passes to sort an array of n elements. In the k th pass we have $n-k$ comparisons. Every pass is reduced by one check because the smallest element, once placed correctly, need not be included in the next pass.



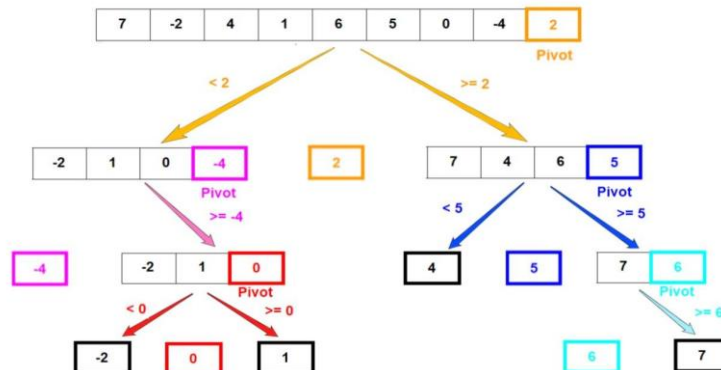
- Merge sort:** In this technique, we split the array into n sub-arrays which contain one element each and then we repeatedly merge the sub-arrays (using a merge function) into larger sub-arrays until only one sorted array remains. It involves recursion as the function calls itself.



- Insertion sort:** The first step involves the comparison of the element in question with its adjacent element. And if at every comparison reveals that the element in question can be inserted at a particular position, then space is created for it by shifting the other elements one position to the right and inserting the element at the suitable position. The above procedure is repeated until all the element in the array is at their apt position.



- Quick sort:** Select any splitting value, say L. The splitting value is also known as **Pivot**. Divide the array into two. A-L and M-Z. It is not necessary that the sub-arrays should be equal. The same approach was used for the sub-arrays by using the method of recursion. Ultimately, the sub-arrays can be combined to produce a fully sorted and ordered array.



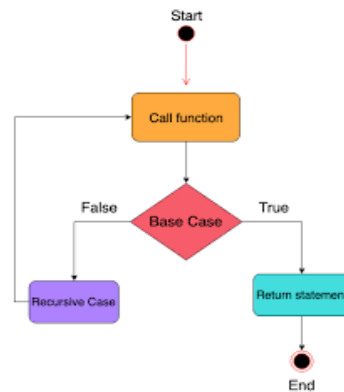
11. RECURSION

- Recursion is the phenomenon of a function calling itself.
- A recursive function can go infinite like a loop. To avoid infinite running of a recursive function, it has:

Base criteria – There must be at least one base criteria or condition, such that, when this condition is met the function stops calling itself recursively.

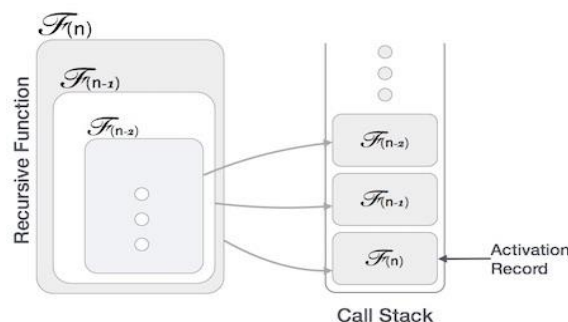
Progressive approach – The recursive calls should progress in such a way that each time a recursive call is made it comes closer to the base criteria.

- A call made to a function is $O(1)$, hence the (n) number of times a recursive call is made makes the recursive function $O(n)$.



Implementation

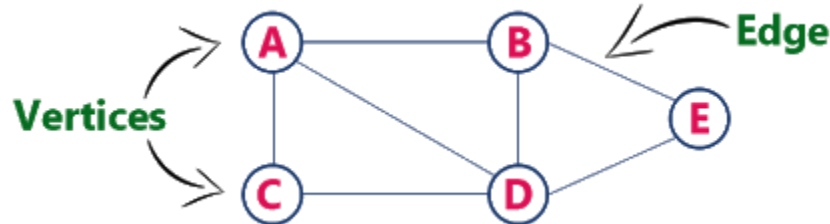
- Many programming languages implement recursion by means of **stacks**.
- Generally, whenever a function (**caller**) calls another function (**callee**) or itself as callee, the caller function transfers execution control to the callee. This transfer process may also involve some data to be passed from the caller to the callee.
- The caller function has to suspend its execution temporarily and resume later when the execution control returns from the callee function. Here, the caller function needs to start exactly from the point of execution where it puts itself on hold. It also needs the exact same data values it was working on.
- For this purpose, an activation record (or stack frame) is created for the caller function. This activation record keeps the information about local variables, formal parameters, return address and all information passed to the caller function.



12. GRAPHING ALGORITHMS

What is a Graph?

- A graph consists of a finite set of vertices or nodes and a set of edges connecting these vertices. $G=(v,e)$



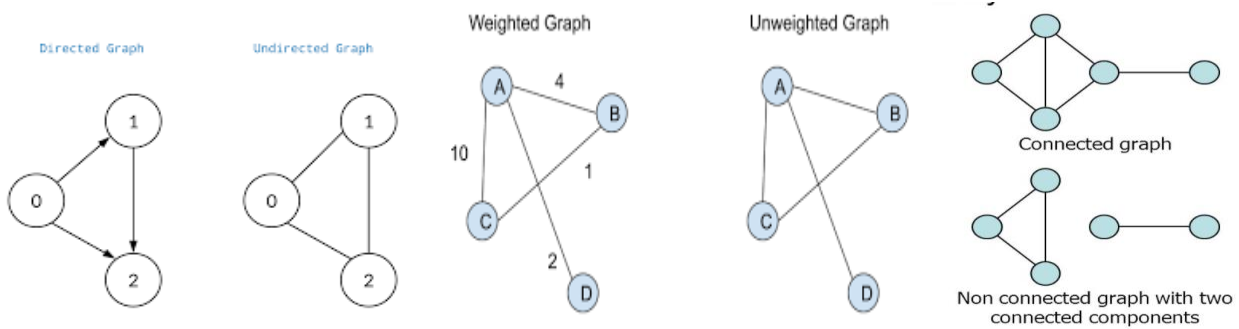
Elements of a graph

- **Adjacent vertices:** Two vertices connected by an edge
- **Parallel edges:** Two edges in opposite direction connecting the same pair of adjacent vertices.
- **Order:** The number of vertices in the graph
- **Size:** The number of edges in the graph
- **Vertex degree:** The number of edges incident to a vertex. Indegree is the number of edges entering a vertex and outdegree is the number of edges leaving the vertex
- **Isolated vertex:** A vertex that is not connected to any other vertices in the graph
- **Self-loop:** An edge from a vertex to itself in a directed graph.
- **Articulation point:** A vertex which when removed splits the graph into multiple parts
- **Path:** A set of vertices connecting any two vertices
- **Cycle:** A path that starts and ends at the same vertex.

Types of graphs

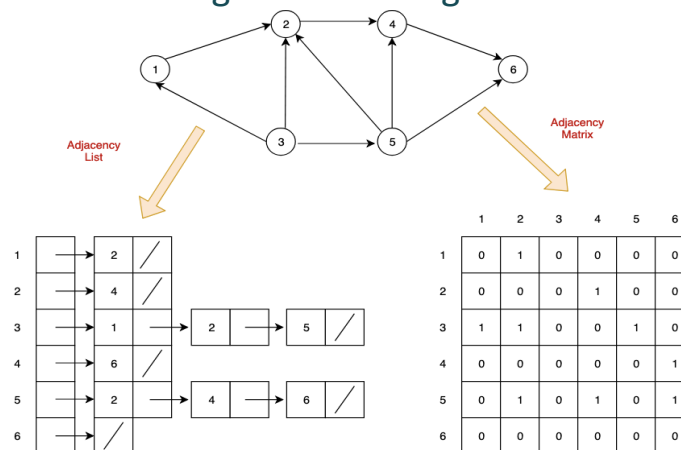
- **Directed graph:** A graph where all the edges have a direction indicating what is the start vertex and what is the end vertex. Also called a digraph.
- **Undirected graph:** A graph with edges that have no direction. Also called a graph. The edges are counted as going both ways.
- **Weighted graph:** Edges of the graph have weights
- **Unweighted graph:** Edges of the graph have no weights

- **Simple Digraph:** No self-loop or parallel edges
- **Non connected graph:** Has more than one unconnected components.
- **Connected graph:** Has more than one components which are all connected.
- **Strongly connected graph:** Directed graph in which from every vertex you can reach all other vertices.
- **Directed acyclic graph:** Directed graph without cycles.



Representation of graphs

- **Adjacency matrix:** Take a $v \times v$ matrix and between any two vertices, where there is an edge, fill 1 in the matrix else zero. The columns give the indegree and rows give the outdegree.
- **Cost adjacency matrix:** Instead of 1, fill with weights of edges.
- **Adjacency list:** It is an array of linked lists. It is of size v . Each linked list shows the edges connected to that vertex. For directed graphs only see edges going out. Normal adjacency list gives the outdegree and inverse adjacency list gives the indegree.
- **Cost adjacency list:** Store the weight of each edge too.

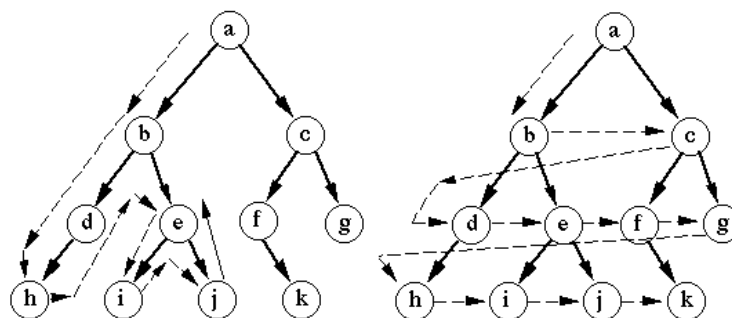


- **Compact list:** Take an array of $v+2e+2$ size. Leave the first spot blank and then fill the index number from which the edges of a particular vertex start. Then, starting from that index, fill the connected vertices of that particular vertex. Use a 2d array for weighted graphs. Normal compact list gives the outdegree and inverse compact list gives the indegree.

Graphing algorithms

a) Searching

1) Depth first search:	2) Breadth first search:
<ul style="list-style-type: none"> • Form a tree of the given graph by starting from one vertex and keep moving forward. When you reach the end, backtrack and reexplore. • Make a dotted edge (back edge) wherever a vertex is repeated. • We start from a particular vertex and explore as far as possible along each branch before retracing back (backtracking). • Unlike trees, graphs can contain cycles (a path where the first and last vertices are the same). Hence, we have to keep track of the visited vertices. • When implementing DFS, we use a stack data structure to support backtracking. • Analytical time: $O(n)$ 	<ul style="list-style-type: none"> • Form a tree of the given graph by starting from one vertex and putting all its adjacent vertices at the same level. Keep moving forward. • Make a dotted edge (cross edge) wherever a vertex is repeated. • We start at a particular vertex and explore all of its neighbours at the present depth before moving on to the vertices in the next level. • Unlike trees, graphs can contain cycles (a path where the first and last vertices are the same). Hence, we have to keep track of the visited vertices. • When implementing Breadth first search, we use a queue data structure. • Analytical time: $O(n)$



Depth-first search

Breadth-first search

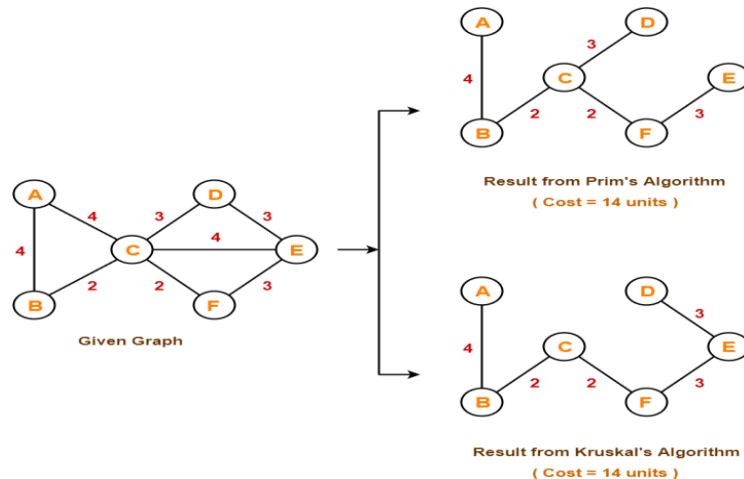
b) Minimum spanning tree

A spanning tree is a subgraph of a graph that has all vertices of a graph and $v-1$ edges so that there is no cycle and all vertices are connected.

A minimum spanning tree is a spanning tree with the minimum sum of edge weights.

Number of spanning trees is $e!/((v-1)!(e-v+1)!)$ – no. of cycles

1) Prim's algorithm:	2) Kruskal's algorithm:
<ul style="list-style-type: none">• Select the least cost edge. Select the next minimum cost edge that is connected to already selected vertices. Keep going till all vertices are covered.• Time: $O(n^2)$• More focus on finding a tree than minimizing it• It can find spanning tree for only one component of a non connected graph.	<ul style="list-style-type: none">• Keep selecting the edges in increasing order of their cost, but make sure they do not form a cycle.• Time: $O(n^2)$• More focus on minimizing than finding a tree• It can find spanning tree for each component of a non connected graph.



c) Maximum flow

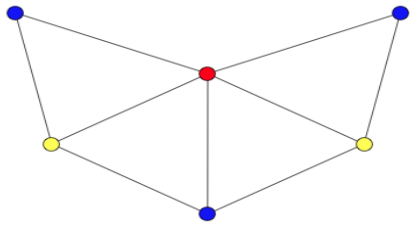
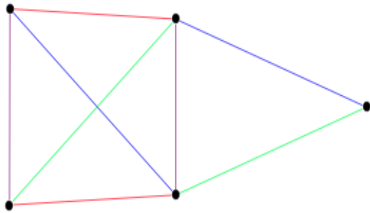
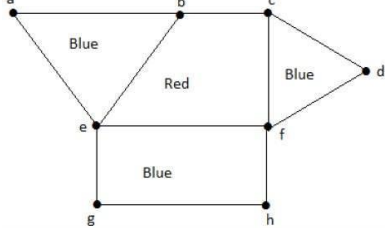
We can model a graph as a flow network with edge weights as flow capacities. In the maximum flow problem, we have to find a flow path that can obtain the maximum possible flow rate.

- 1) Ford-Fulkerson algorithm
- 2) Edmonds–Karp algorithm
- 3) Dinic's algorithm

d) Graph colouring

Graph colouring assigns colours to elements of a graph while ensuring certain conditions.

The minimum number of colors required for graph 'G' is called as the chromatic number of G, denoted by $X(G)$.

1) Vertex colouring	2) Edge colouring	2) Face colouring
<p>Assignment of labels, called colors, to the vertices of a graph such that no two adjacent vertices share the same color.</p> <p>Two vertices are said to be adjacent if they have a common edge</p> 	<p>Each edge of a graph has a color assigned to it in such a way that no two adjacent edges are the same color.</p> <p>Two edges are said to be adjacent if they have a common vertex</p> 	<p>Region coloring is an assignment of colors to the regions of a planar graph such that no two adjacent regions have the same color.</p> <p>Two regions are said to be adjacent if they have a common edge</p> 

e) Matching

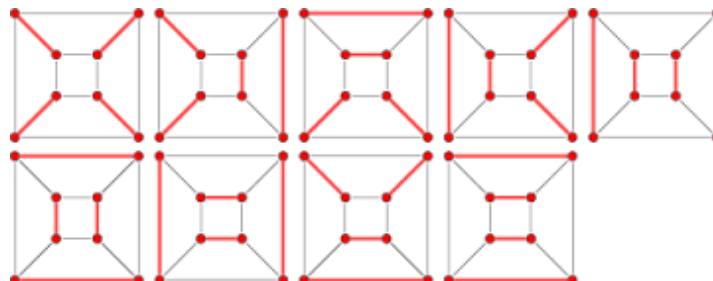
A matching in a graph is a set of edges that does not have common vertices (i.e., no two edges share a common vertex).

A matching is called a maximum matching if it contains the largest possible number of edges matching as many vertices as possible.

1) Hopcroft-Karp algorithm

2) Hungarian algorithm

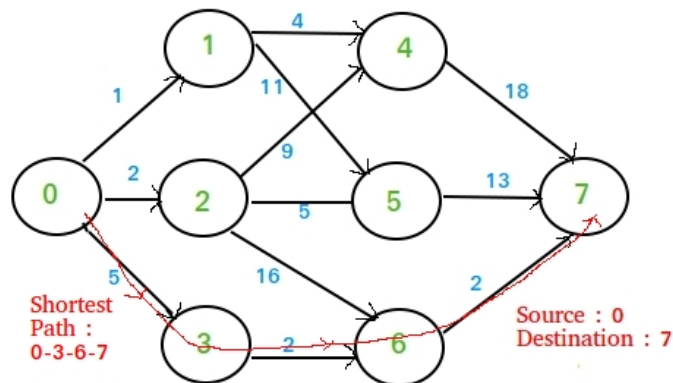
3) Blossom algorithm



f) Shortest path:

The *shortest path* from one vertex to another vertex is a path in the graph such that the sum of the weights of the edges that should be travelled is minimum.

1) Dijkstra's algorithm:	2) Bellman Ford's algorithm:
<ul style="list-style-type: none"> Choose a starting vertex. Mark the values of direct paths. Mark other vertices as infinity. Use relaxation, which means if distance of vertex u + cost of edge $(u, v) <$ the marked distance of vertex v, replace the marked distance by the distance of vertex u + cost of edge (u, v). Thus, keep moving ahead and relax all vertices. Thus you get the shortest distance to each vertex Worst case time depends on n^2 Drawback: may not work on negative edges 	<ul style="list-style-type: none"> Make a list of all edges. Use relaxation, which means if distance of vertex u + cost of edge $(u, v) <$ the marked distance of vertex v, replace the marked distance by the distance of vertex u + cost of edge (u, v). Keep moving ahead and relax all vertices. Relax each edge $v-1$ times. (repeat entire process) Thus you get the shortest distance to each vertex Worst case time depends on n^3 Drawback: may not work if the total weight of a cycle is negative

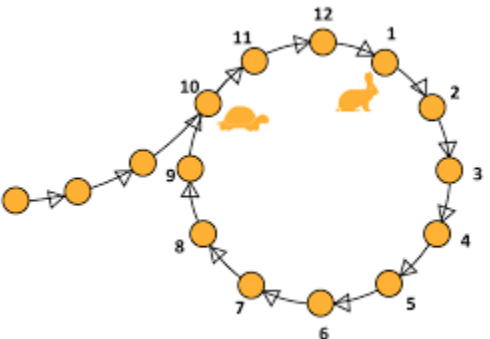


g) Cycle detection

A *cycle* is a path in a graph where the first and last vertices are the same.

If we start from one vertex, travel along a path and end up at the starting vertex, then this path is a cycle.

Cycle detection is the process of detecting these cycles

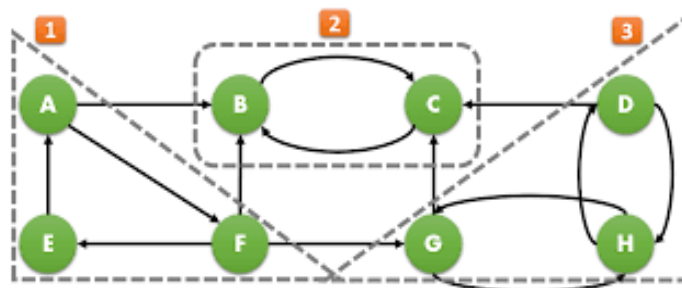
1) Floyd's cycle detection algorithm	2) Brent's algorithm
<ul style="list-style-type: none">• Also known as the tortoise and the hare problem, you take two pointers and in each iteration, make one move one step and the other move two steps.• In some iteration, they both will point to the same node if it is a cycle.• It does not give us the length of the cycle, we have to find that separately. 	<ul style="list-style-type: none">• Move fast pointer (or second_pointer) in powers of 2 until we find a loop.• After every power, we reset slow pointer (or first_pointer) to previous value of second pointer.• Reset length to 0 after every every power.• The condition for loop testing is first_pointer and second_pointer become same. And loop is not present if second_pointer becomes NULL.• When we come out of loop, we have length of loop.• We reset first_pointer to head and second_pointer to node at position head + length.• Now we move both pointers one by one to find beginning of loop.

h) Strongly connected components

A graph is said to be strongly connected if every vertex in the graph is reachable from every other vertex.

If you make every strongly connected component one vertex, you get a directed acyclic graph.

1) Kosaraju's algorithm	2) Tarjan's strongly connected components algorithm
<ul style="list-style-type: none">• Start from one vertex, as soon as u visit a vertex, add it in the set of visited vertices.• Explore its children.• Once all children of a vertex are explored, add it in a stack of finished vertices.• If all children are explored backtrack and reexplore.• Then move to new starting points.• After the stack is filled, reverse the graph.• Take out vertices from the stack and as u explore, add them to a set of strongly connected vertices until you cannot do so anymore.• Find all such strongly connected components.	<ul style="list-style-type: none">• The low link id of a node is the smallest(lowest) node id reachable from the node while doing a dfs (Including itself)• Mark the ids of each node as unvisited• Start DFS. Upon visiting a node, assign it an id and a low link value. Also mark current nodes as visited and add them to a seen stack.• On DFS callback, if the previous node is on the stack then min the current node's low-link value with the last node's low link value.• After visiting all neighbours, if the current node started a connected component then pop nodes off stack until current node is reached.

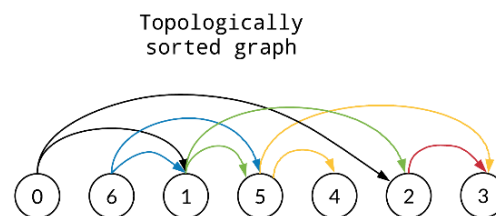
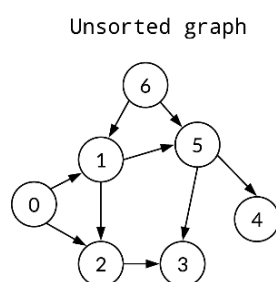


Strongly Connected Components

i) Topological sorting

Topological sorting of a directed acyclic graph is a linear ordering of its vertices so that for each directed edge (u, v) in the ordering, vertex u comes before v .

1) Kahn's algorithm	2) Algorithm based on DFS
<ul style="list-style-type: none"> Choose vertices in the same order as the eventual topological sort.^[2] First, find a list of "start nodes" which have no incoming edges and insert them into a set S; at least one such node must exist in a non-empty acyclic graph. $L \leftarrow$ Empty list that will contain the sorted elements $S \leftarrow$ Set of all nodes with no incoming edge while S is not empty do remove a node n from S add n to L for each node m with an edge e from n to m do remove edge e from the graph if m has no other incoming edges then insert m into S if graph has edges then return error (graph has at least one cycle) else return L (a topologically sorted order) 	<ul style="list-style-type: none"> The algorithm loops through each node of the graph, in an arbitrary order, initiating a depth-first search that terminates when it hits any node that has already been visited since the beginning of the topological sort or the node has no outgoing edges (i.e. a leaf node): $L \leftarrow$ Empty list that will contain the sorted nodes while exists nodes without a permanent mark do select an unmarked node n visit(n) function visit(node n) if n has a permanent mark then return if n has a temporary mark then stop (not a DAG) mark n with a temporary mark for each node m with an edge from n to m do visit(m) remove temporary mark from n mark n with a permanent mark add n to head of L



13. NUMERICAL ALGORITHMS

a) Kasturbha multiplication

Here's the naive multiplication algorithm to multiply two n -bit numbers, x and y that are in base b .

Divide each number into two halves, the high bits H and the low bits L :

$$x = x_H b^{\frac{n}{2}} + X_L, \quad y = y_H b^{\frac{n}{2}} + Y_L.$$

Multiply the two numbers together:

$$\begin{aligned} xy &= (x_H b^{\frac{n}{2}} + X_L) \times (y_H b^{\frac{n}{2}} + Y_L) \\ &= x_H y_H b^n + (x_H y_L + x_L y_H) b^{\frac{n}{2}} + x_L y_L. \end{aligned}$$

These formulas describe what is going on in the image above--the familiar grade-school way of multiplying numbers. This has four multiplications: $x_H y_H b^n$, $(x_H y_L) b^{\frac{n}{2}}$, $(x_L y_H) b^{\frac{n}{2}}$, and $x_L y_L$, which has a running time of $O(n^2)$.

The Karatsuba algorithm decreases the number of subproblems to three and ends up calculating the product of two n -bit numbers in $\Theta(n^{\log_2 3})$ time--a vast improvement over the naive algorithm.

To multiply two n -bit numbers, x and y , the Karatsuba algorithm performs three multiplications and a few additions, and shifts on smaller numbers that are roughly half the size of the original x and y .

Here's how the Karatsuba method works to multiply two n -bit numbers x and y which are in base b .

Create the following three subproblems where H represents the high bits of the number and L represents the lower bits:^[1]

- $a = x_H y_H$
- $d = x_L y_L$
- $e = (x_H + x_L)(y_H + y_L) - a - d$
- $xy = ab^n + eb^{\frac{n}{2}} + d.$

This only requires three multiplications and has the [recurrence](#)

$$3T\left(\frac{n}{2}\right) + O(n) = O(n^{\log 3}).$$

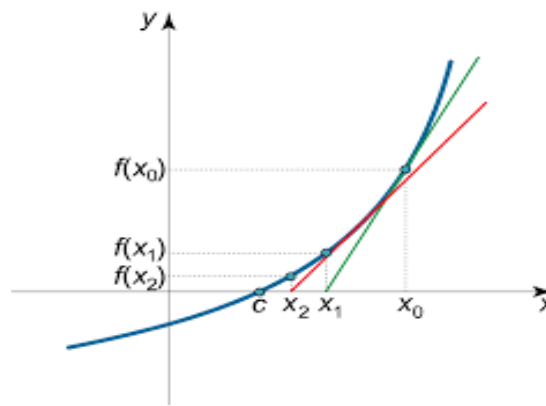
Karatsuba can be applied recursively to a number until the numbers being multiplied are only a single-digit long (the base case).

[Divide and conquer](#) techniques come in when Karatsuba uses [recursion](#) to solve subproblems--for example, if multiplication is needed to solve for a , d , or e before those variables can be used to solve the overall $x \times y$ multiplication.

b) Euclid's algorithm for gcd

- If d divides both m and n , d divides $m-n$ too. ($m > n$).
- $m = ad$ $n = bd$ $m-n = (a-b)d$
- So $\gcd(m, n) = \gcd(m-n, n)$
- Let $m, n > 0$ be positive integers. If n divides m , then $\gcd(m, n) = n$. Otherwise $\gcd(m, n) = \gcd(m \% n, n)$.

c) Newton's method

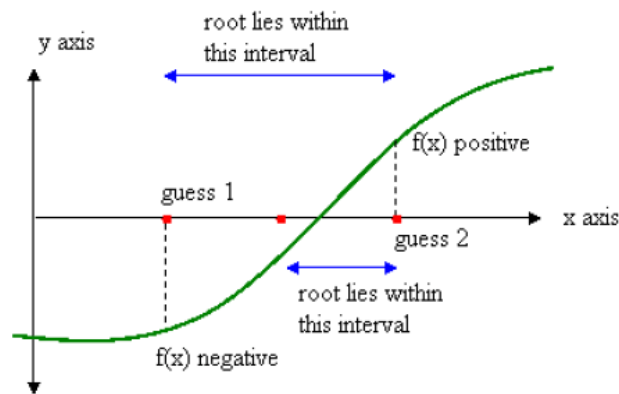


- It is a method to find the root of $f(x)$, i.e. x s.t. $f(x)=0$
- Method works if $f(x)$ and derivative $f'(x)$ can be easily calculated and a good initial guess x_0 for the root is available
- Point A ($x_i, 0$) is known. Calculate $f(x_i)$. Point B = ($x_i, f(x_i)$). Draw the tangent to $f(x)$. Point C = intercept on x axis. Point C = ($x_{i+1}, 0$)
- $f'(x_i) = \text{derivative} = (d f(x))/dx$ at $x_i = AB/AC$
- $x_{i+1} = x_i - AC = x_i - AB/(AB/AC) = x_i - f(x_i) / f'(x_i)$
- Starting with x_0 assumed, we compute x_1 , then x_2 , ... We can get as close to $f(y)$ as required

d) Taylor/Mclaurin series

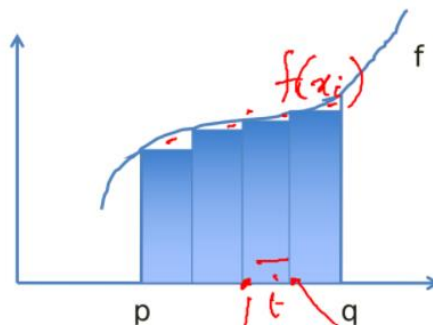
- When x is close to x_1 :
$$f(x) = f(x_1) + f'(x_1)(x-x_1) + f''(x_1)(x-x_1)^2 / 2! + f'''(x_1)(x-x_1)^3 / 3! + \dots$$
- When x is close to 0:
$$f(x) = f(0) + f'(0)x + f''(0)x^2 / 2! + f'''(0)x^3 / 3! + \dots$$
- Now we can write a program for different functions using this formula.
- Loop control variable will be k
- In each iteration we calculate t_k from t_{k-1}
- The term t_k is added to sum
- A variable term will keep track of t_k .
- At the beginning of k th iteration, term will have the value t_{k-1} , and at the end of k th iteration it will have the value t_k
- After k th iteration, sum will have the value = sum of the first k terms of the Taylor series
- Initialize sum = x , term = x
- In the first iteration of the loop we calculate the sum of 2 terms. So initialize $k=2$
- We stop the loop when term becomes small enough

e) Bisection method



- It is a method to find root of function f : Value x such that $f(x)=0$
- Requirement: Need to be able to evaluate f , f must be continuous, we must be given points x_L and x_R such that $f(x_L)$ and $f(x_R)$ are not both positive or both Negative
- Because of continuity, there must be a root between x_L and x_R (both inclusive)
- Let $x_M = (x_L + x_R)/2$ = midpoint of interval (x_L, x_R)
- If $f(x_M)$ has same sign as $f(x_L)$, then $f(x_M), f(x_R)$ have different signs. So we can set $x_L = x_M$ and repeat. Similarly if $f(x_M)$ has same sign as $f(x_R)$
- In each iteration, x_L, x_R are coming closer.
- When they come closer than certain epsilon, we can declare x_L as the root.

f) Integration



- Integral from p to q = area under curve
- Approximate area by rectangles
- Read in p, q (assume $p < q$)
- Read in n = number of rectangles
- Calculate w = width of rectangle = $(q-p)/n$
- i th rectangle, $i=0,1,\dots,n-1$ begins at $p+iw$
- Height of i th rectangle = $f(p+iw)$
- Given the code for f , we can calculate height and width of each rectangle and so we can add up the areas

14. DYNAMIC PROGRAMMING

- Dynamic Programming (DP) is an algorithmic technique for solving an optimization problem by breaking it down into simpler subproblems and utilizing the fact that the optimal solution to the overall problem depends upon the optimal solution to its subproblems.

Characteristics of Dynamic Programming

1. **Overlapping Subproblems**

Subproblems are smaller versions of the original problem. Any problem has overlapping sub-problems if finding its solution involves solving the same subproblem multiple times.

2. **Optimal Substructure Property**

Any problem has optimal substructure property if its overall optimal solution can be constructed from the optimal solutions of its subproblems.

Dynamic Programming Methods

1. **Top-down with Memoization**

In this approach, we try to solve the bigger problem by recursively finding the solution to smaller sub-problems.

Whenever we solve a sub-problem, we cache its result so that we don't end up solving it repeatedly if it's called multiple times. Instead, we can just return the saved result.

This technique of storing the results of already solved subproblems is called Memoization

2. **Bottom-up with Tabulation**

Tabulation is the opposite of the top-down approach and avoids recursion.

In this approach, we solve the problem "bottom-up" (i.e. by solving all the related sub-problems first). This is typically done by filling up an n-dimensional table.

Based on the results in the table, the solution to the top/original problem is then computed.

Types of dynamic programming problems

1. **Optimization problems.**

The optimization problems expect you to select a feasible solution, so that the value of the required function is minimized or maximized.

2. **Combinatorial problems.**

Combinatorial problems expect you to figure out the number of ways to do something, or the probability of some event happening.

Every Dynamic Programming problem has a schema to be followed:

- Show that the problem can be broken down into optimal sub-problems.
- Recursively define the value of the solution by expressing it in terms of optimal solutions for smaller sub-problems.
- Compute the value of the optimal solution in bottom-up fashion.
- Construct an optimal solution from the computed information.

Eg. Fibonacci numbers

Fibonacci (n) = 1; if n = 0

Fibonacci (n) = 1; if n = 1

Fibonacci (n) = Fibonacci(n-1) + Fibonacci(n-2)

So, the first few numbers in this series will be: 1, 1, 2, 3, 5, 8, 13, 21... and so on!

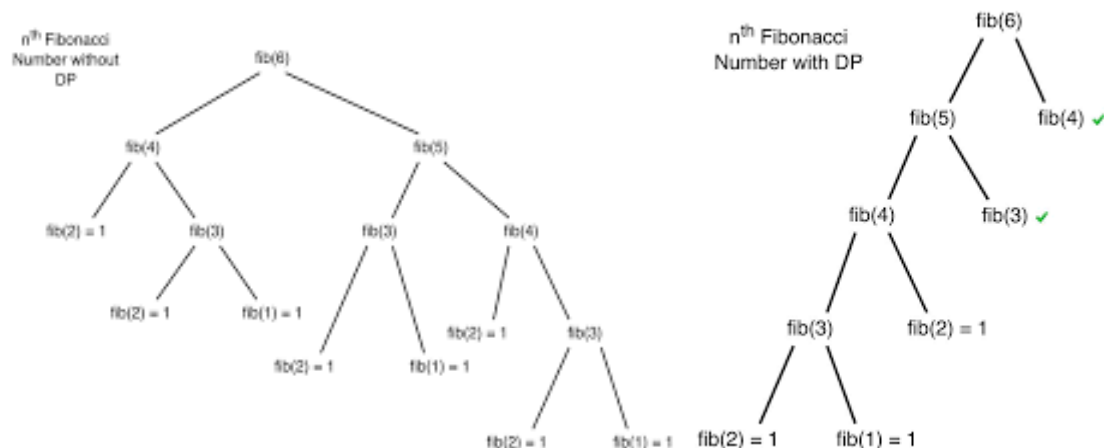
A code for it using pure recursion:

```
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];  
}
```

In the recursive code, a lot of values are being recalculated multiple times. We could do good with calculating each unique quantity only once.



Eg. Tower of Hanoi

The Tower of Hanoi is a mathematical game or puzzle. It consists of three rods, and a number of disks of different sizes which can slide onto any rod. The puzzle starts with the disks in a neat stack in ascending order of size on one rod, the smallest at the top, thus making a conical shape.

The objective of the puzzle is to move the entire stack to another rod, obeying the following rules:

- Only one disk may be moved at a time.
- Each move consists of taking the upper disk from one of the rods and sliding it onto another rod, on top of the other disks that may already be present on that rod.
- No disk may be placed on top of a smaller disk.

The dynamic programming solution consists of solving the functional equation
 $S(n, h, t) = S(n-1, h, \text{not}(h, t)) ; S(1, h, t) ; S(n-1, \text{not}(h, t), t)$

Where n denotes the number of disks to be moved, h denotes the home rod, t denotes the target rod, $\text{not}(h, t)$ denotes the third rod (neither h nor t), ";" denotes concatenation, and $S(n, h, t) :=$ solution to a problem consisting of n disks that are to be moved from rod h to rod t .

For $n=1$ the problem is trivial, namely $S(1, h, t) =$ "move a disk from rod h to rod t " (there is only one disk left).

The number of moves required by this solution is $2^n - 1$.

