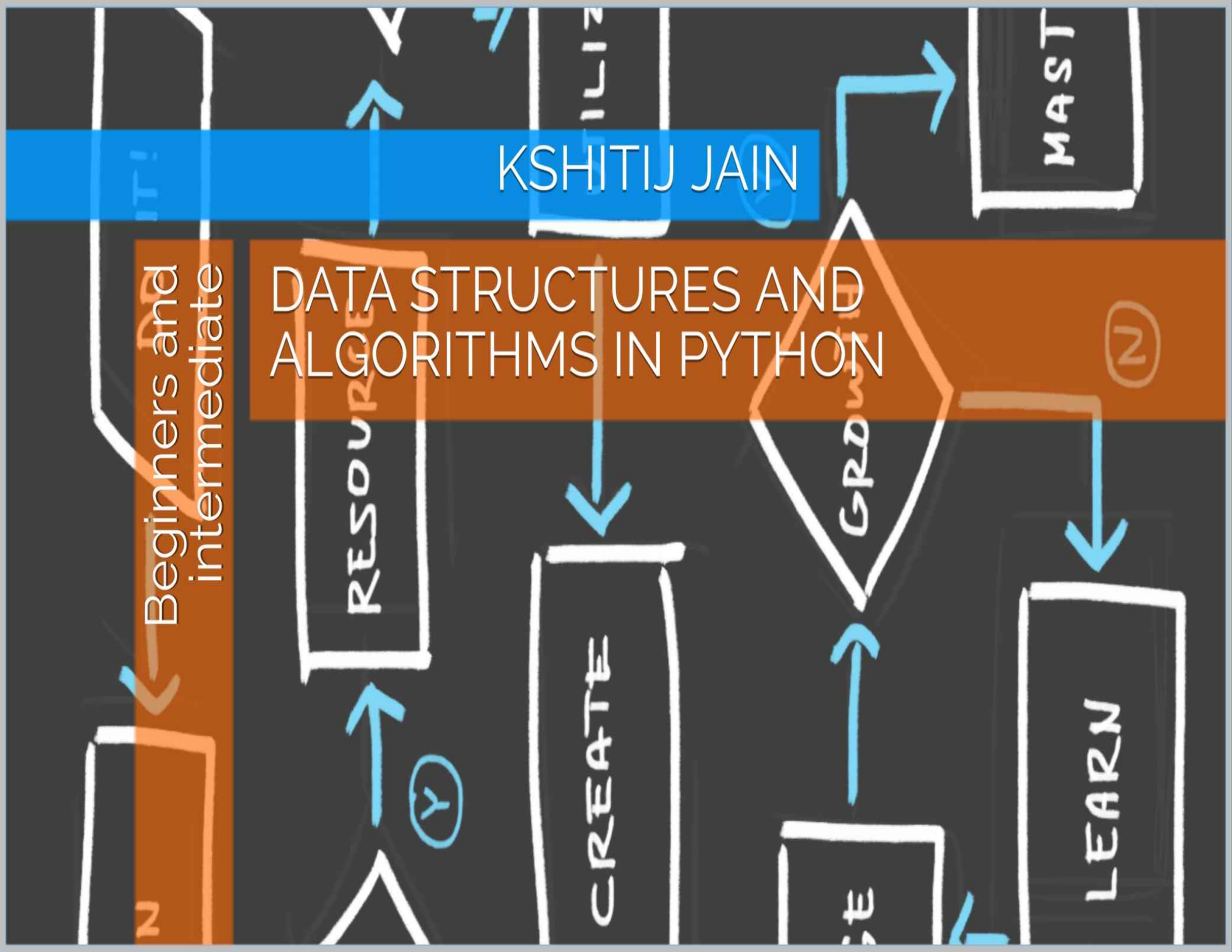

KSHITIJ JAIN
DATA STRUCTURES AND ALGORITHMS IN PYTHON
Beginners and intermediate
RESOURCE
CREATE
LEARN
MAST

DATA STRUCTURES AND ALGORITHMS IN PYTHON

Beginners and intermediate

Kshitij Jain

Kshitij Jain
Visit my Author Central Page at http://bit.ly/my_authorpage

CONTENTS

DATA STRUCTURES

Data structures

For all the data structure's functions, you can always refer to the Python documentation: https://docs.python.org/2/tutorial/index.html

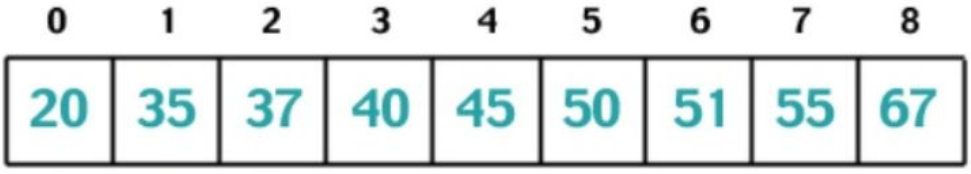
LIST

Explanation: A list represents a set of ordered elements, which means that each element can have another element before or after it. A linked list consist of **nodes** that contain **data**. Each node, or position **gets** a certain ordering number, depending on its ordering in the list: we talk about the first element, the second etc. Each node **contains** a value; it can be an integer, a string etc. When we refer to a certain position (for example when we talk about the first element), there is maximum one entry that may satisfy our request. This means that we cannot have two or more elements in a position.

The first node is called **head**. Internally, each node points to the **next** one, and the last one points to **null**. In Python we do not have to worry about pointers.

Below, we have a simple list. The black numbers represent the node number, the ordering of its cell in the list. Lists in python start from 0, so the first element is at position number 0.

Blue numbers represent the values of each cell. So, the first cell contains number 20, the second contains number 35 etc.



We can add and remove elements in a list and we can search for a certain element with reference to its position in the list.

When we want to add an element, this means that we are placing it in the end of an existing list. This needs only 1 step, therefore the complexity is O(1).

When we want to delete an element, we need first to search for it and then remove it. In the worst case, our element will be the last one, so we need to search the whole list of length n. The complexity will be O(n). Then, we can delete it in one step.

Implementation: We can name our list with a name of our liking, usually something that represents its functionality (be careful not to use a built-in python name!).

Lets name our list **a**.

In the brackets that follow the name of our list, we place the index, the position which we refer to:

a[0] refers to the element at the **first** position (**position 0**). We need to assign a value in this cell.

a[0] = 5, assigns 5 in the **first** node of our list.

So, we have:

list[index] = value

We can easily create a list in Python. In order to define a list (initially empty) we just do:

a = []

Else, we could initialize our list with some data:

b = [2,5,4]

Note that we do not have to specify the positions of our elements each time, as the first element will always be in position 0 (b[0] is 2), the second in position 1 (b[1] is 5) etc.

We can add add an element x in the end of the list:

list.append(x)

For example, b.append(7) will give us:
b = [2,5,4,7]

We can also add an element x in a certain position I of the list.

list.insert(i,x)

For example in our previous list, we can do b.insert(0,3), which will give us:
b = [3,2,5,4,7]

We can remove an element that contains a certain value:

list.remove(x)

Note that this will delete the first element that contains this certain value.

For example in our previous list, if we do b.remove(2), we will get:
b = [3,5,4,7]

TUPLE

Explanation: Tuples serve a similar functionality with lists, with the main difference being that tuples are immutable: as soon as you add your data in a tuple, they cannot be changed or deleted and it is not possible to insert new data. The only way to perform such operations is to create a new tuple containing the new or edited data etc.

In terms of syntax, tuples use parentheses to store the data in place of the lists' square brackets. There is also the possibility to not include parentheses at all, and just separate data with a comma.

When we need to refer to a certain position in the tuple, for example position 0 (first element), position 1 (second element) etc, square brackets are used.

Implementation: We can create an empty tuple named t:

```
t = ()
```

We can also create a tuple that contains some numbers:

```
t_2 = (9,4,6,8,2)
```

or:

```
t_2 = 9,4,6,8,2
```

Now, we can refer to a certain element of the tuple, like for example the first element:

```
t_2[0] = 9
```

DICTIONARY

Explanation: A dictionary is used to store data by using a unique key for each entry. Key is somehow serving the same functionality as indices in lists and tuples, when we want to refer to a certain element. The values where each key refers to are not necessarily unique. Dictionaries are mutable, so we can add, delete or modify the elements in it.

The entries of a dictionary are contained between curly braces {}. Each entry is in the form key: value.

Implementation: We have a dictionary d:

```
d = {}
```

and we can populate it with key-value pairs:

```
d = { key1:value1,
key2:value2,
key3:value3,
...
}
```

In Python, there is also the possibility to create a dictionary, by calling the built-in function dict() containing key-value pairs:

```
d = dict([(key1,value1),
(key2, value2),
(key3,value3),…])
```

Note that the arguments of dict() can be in the form of a list of tuples (like above) or a list of lists.

We can refer to any element of the dictionary by choosing a certain key enclosed in []:

dictionary[key] = value

The uniqueness of the key ensures that we are going to get back the right value.

The equivalent of the above is the method get, which takes a certain key as an argument and returns its corresponding value:

dictionary.get(key) = value

We can extract the key-value pairs in a form of a list of tuples using the function items():

dictionary.items() = [(key1,value1), (key2,value2), (key3,value3), …]

There is also the possibility to extract only the keys of the dictionary or only the values, by using keys() and values() functions respectively:

dictionary.keys() = [key1, key2, key3, …]

dictionary.values() = [value1, value2, value3, …]

SET

Explanation: A set is a collection of items that in contrast to lists/tuples, it is unordered. There can be no duplicate elements. Sets are efficient when we want to search if an item is contained in them or not. Later on, we are going to talk about hash functions, which implement efficient searching. We can add, delete or modify the items in a set. The set items are contained in {}.

Implementation: We can define an empty set s_empty:

s_empty = {}

or we can construct a set of certain items, like a set of fruits:

s_fruits = {‘orange’, ‘banana’, ‘apple’, ‘peach’}

We can add an item using the add() function:

s_fruits.add(‘cherry’)

‘cherry’ will be added in a random position in the initial set (remember, sets are unordered!). For example:

s_fruits = {‘orange’, ‘banana’, ‘cherry’,‘apple’, ‘peach’}

or we can add multiple items in the s_fruits set using the update() function:

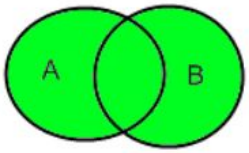
s_fruits.update([‘grapes’, ‘mango’])

s_fruits = {‘orange’, ‘grapes’, ‘banana’, ‘cherry’,‘apple’, ‘peach’, ‘mango’}

Mathematical set operations like intersection, union, difference etc are valid in Python as well. So, if we have set A and set B:

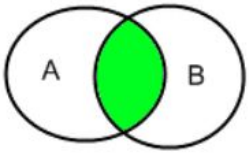
A.union(B)

Union of A and B



A.intersection(B)

Intersection of A and B



A.difference(B)

ARRAY

Explanation: An array consists of an ordered collection of elements. We can refer to each item by its index in the array. An array can have dimensions: the widely used 2-dimensional table that we encounter very often in our everyday life, but we can also have more dimensions. In **computer science theory**, the difference between an array and a list is the way data are stored in memory, which affects how we access an element. A list allocates non sequential cells in memory, therefore they need a reference for the next cell; this is what the index does. Lists support sequential access, where you need to search an element by examining the first element, then the next one, afterwards the third one etc until you find the element you are looking for. An array allocates only sequential cells in memory. Arrays support sequential access, as well as direct access.

Implementation: You may have noticed that Python lists have some array-like features, according to the list analysis above. Python in fact uses array functionalities in lists, like for example direct access to a certain position. However, there is an array structure in Python, but it should be imported:

import array

We need to specify the data type that this array contains and also the list of elements that the array is going to contain when we are creating it. For example:

a = array.array('i', [1,2,3])

Yes, array keyword is written twice, because we are calling the array module from the array library. It is easier to **rename** the imported library array though:

import array as arr

a = arr.array(‘i’, [1,2,3])

‘i’ refers to int Python type. Below is the list of all supported Python data types and the letter they use to indicate this data type when we construct an array:

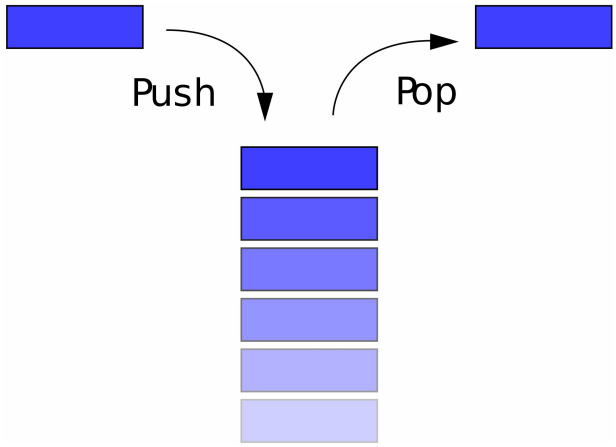
Code	C Type	Python Type	Min bytes
`'b'`	signed char	int	1
`'B'`	unsigned char	int	1
`'u'`	Py_UNICODE	Unicode	2
`'h'`	signed short	int	2
`'H'`	unsigned short	int	2
`'i'`	signed int	int	2
`'I'`	unsigned int	int	2
`'l'`	signed long	int	4
`'L'`	unsigned long	int	4
`'f'`	float	float	4
`'d'`	double	float	8

We can also use the more efficient array that the numpy library provides. In this case, we need to place the following line in our code, before constructing our array:

from numpy import array

STACK

Explanation: Stack is used to store items in the Last in – First out (LIFO) manner. Operations like insert element (push) and remove element (pop) can occur only from the ‘upper’ end of the stack. An overflow can occur if we exceed the maximum size of the stack, while an underflow can occur if we try to delete an element from an empty stack.

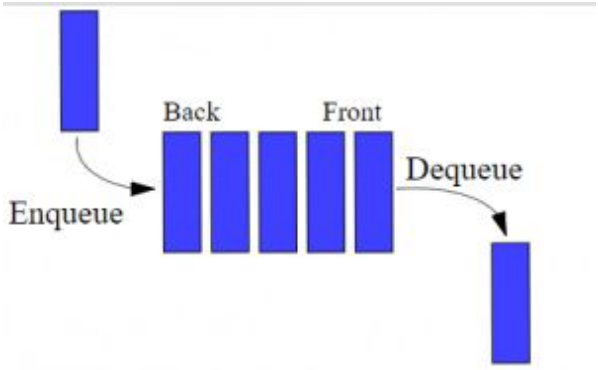


Implementation: In Python a stack can be implemented by using a list structure. We can create a simple stack:

```
stack = []
# Push elements in the stack
stack.append('1')
stack.append('2')
stack.append('3')

# stack = ['1', '2', '3']

# Pop elements from the stack
stack.pop()      # '3'
stack.pop()      # '2'
stack.pop()      # '1'
```

QUEUE

Explanation: A queue handles item in the first in – first out (FIFO) manner. We can add (enqueue) items from the ‘end’ of the queue and delete (dequeue) items from the ‘beginning’ of the queue. Overflow and underflow can occur as in the case of the stack.



Implementation: In Python we can import the queue module:

import queue

Then we can create a queue by calling the method Queue in that module. We can add (put) elements and we can delete (get) them in

the FIFO way:

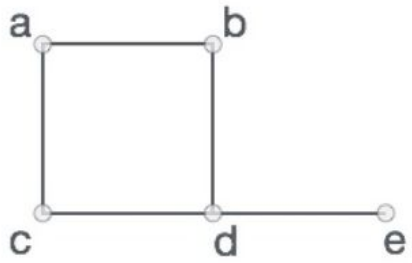
```
q = queue.Queue(maxsize = 100)
q.put('a')
q.put('b')
q.put('c')

# q = ['a','b','c']
q.get()          # 'a'
q.get()          # 'b'
q.get()          # 'c'
```

GRAPH

A graph is a representation of a set of objects (nodes or vertices **V**) where some of them are connected with links (edges **E**). Formally, a graph G is defined as pairs (V, E) . In the graph below we have:

V = {a,b,c,d,e}

E = {ab,ac,bd,cd,de}



We can dynamically add and remove vertices and edges.

Note that the above is an undirected graph: this means that the connections between the vertices have no direction, for example ab = ba etc.

In case of directed graphs there are directed arrows → as edges, and we can move only towards the direction of the arrow.

A sequence of edges that connect a vertex with another, either directly or indirectly, forms a path.

8.1 ADJACENT MATRIX FOR GRAPHS

When getting to coding, we need a data structure to store our graph. Most usually we use adjacent matrices or adjacent lists. In this case, vertices that are next to each other, for example a and b in the example above, are adjacent.

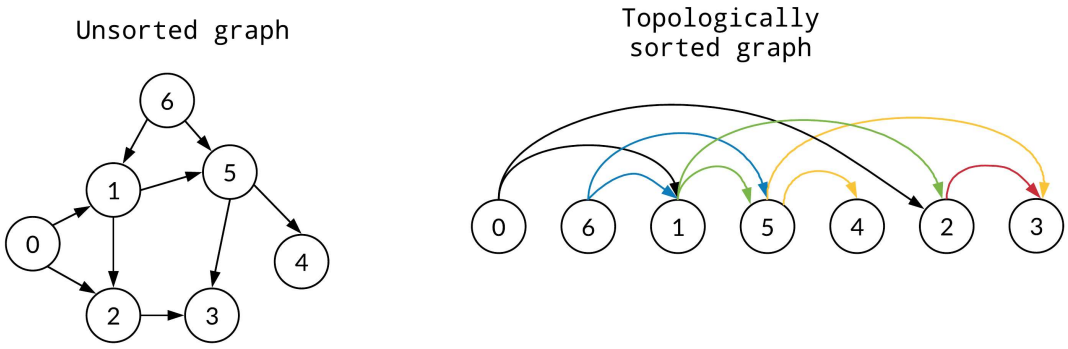
We can easily create an adjacent matrix for a simple graph by creating a 2-dimensional matrix where both rows and columns represent the graph's nodes. We mark with 1 where a connection between two nodes exists, and 0 when we have no connection (regarding an undirected graph). In this case we are considering edges with weight 1 (unweighted edges), which is the default scenario. In case we had different weights for any connection, we should add its weight to the cell that represents that connection instead of 1.

Here is the adjacent matrix of the above graph:

	a	**b**	**c**	**d**	**e**
a	0	1	1	0	0
b	1	0	0	1	0
c	1	0	0	1	0
d	0	1	1	0	1
e	0	0	0	1	0

8.2 TOPOLOGICAL SORT

A topological sort ‘reforms’ a (directed) graph, putting all the vertices on a line that goes from left to right. A graph that can be sorted this way should not contain any circles (a path where the beginning is the same node as the end), because in that case it is impossible to move only from left to right.



Topological sort is very helpful when we want to reveal an ordering among the nodes of a graph. Imagine if we have tasks with certain start and finish times and we wish to visualize their sequence.

An algorithm that helps in topological sorting is DFS, which is going to be explained in detail later. DFS starts from the first node in a graph and it is getting deeper and deeper, visiting first an adjacent to the root node, then a neighbor of that node etc. So, there is a sequence of events, which is reflected to the sequence of DFS visits of the nodes. After visiting a vertex, this vertex is stored in a linked list. In the end, the linked list will represent the sequence of nodes of the initial graph.

TREE

Trees are a subtype of graphs, which needs to satisfy the following requirements:

- It needs to be acyclic: to contain no cycles
- It needs to be connected: Any node should be reachable through some path, so there are not ‘stray’ nodes

Typically, the tree has a structure of a root node and children nodes. Root node can be only one and it can have any number of children nodes (directly connected to the root nodes). Any child can have any number of children etc. Nodes that do not have children are called leaves. Here we can see a tree structure:

BINARY TREE

A binary tree is the tree where any node has two children maximum. We can represent a binary tree with a class, which will contain the information for a root node, based on which we can construct the rest of the tree. This class should contain the value of this node, the right child and the left child. This class can be like the following:

```
class Tree:
def __init__(self,val):
self.data = val
self.left = None
self.right = None
```

Right and left child are initially set to None, as in the beginning we do not have more nodes. The value of each node is passed as a parameter.

Trees in general, as well as binary trees which are widely used in computer science use broadly recursion.

10.1. BINARY SEARCH TREES

Binary search tree (BST) is a binary tree which its elements positioned in special order. In each BST all values(i.e key) in left sub tree are less than values in right sub tree. In this case, when we insert or delete a node, the structure of the tree needs to change.

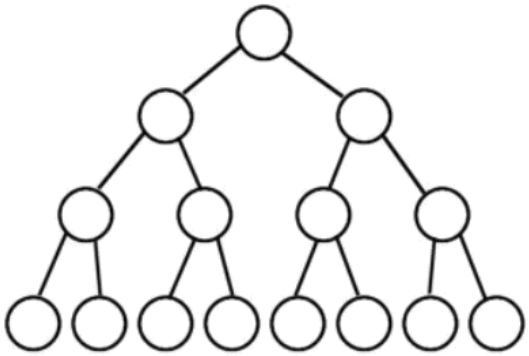
If we need to identify is a binary tree is BST, we need to check if it satisfies any of the conditions below:

1. It is empty
2. It has no subtrees

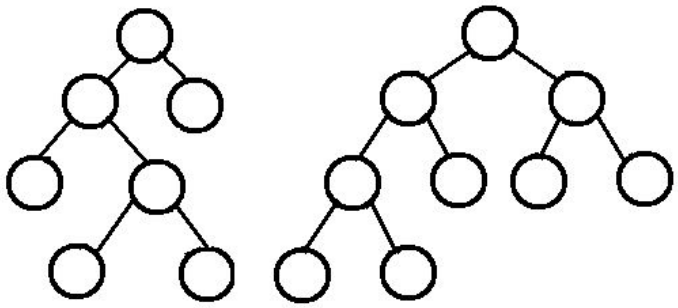
3. For every node x in the tree all the keys (if any) in the left sub tree must be less than key(x) and all the keys (if any) in the right sub tree must be greater than key(x).

10.2. FULL, COMPLETE AND PERFECT BINARY TREE

A full binary tree is the one in which every node (except from the leaves possibly) has 2 children.

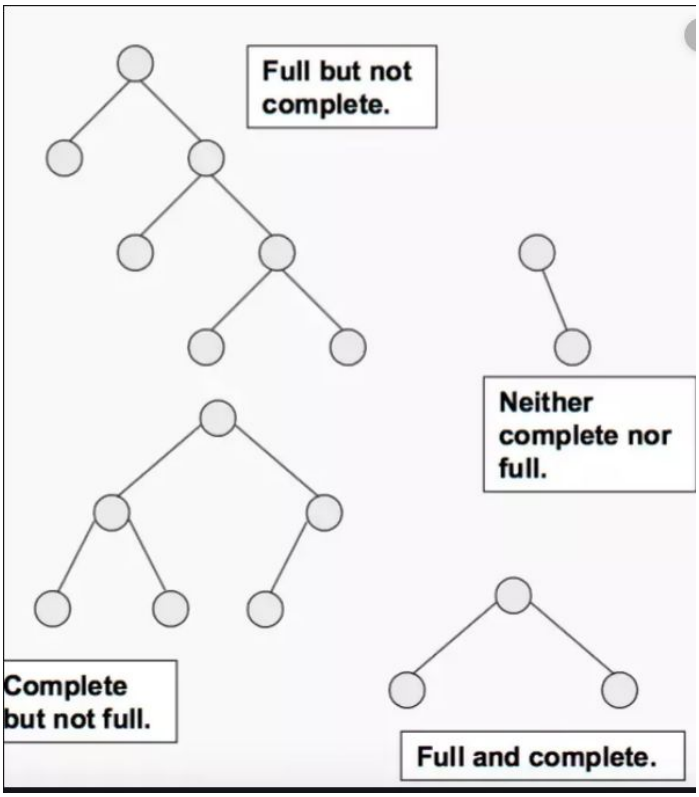


A complete binary tree is a binary tree in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible. We can see the different structure of a full and a complete binary tree:



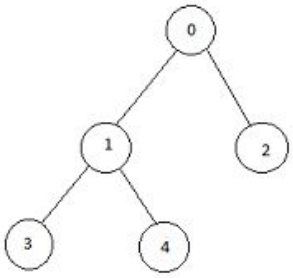
full tree complete tree

And in more detail when a tree is only full, only complete or both full and complete:

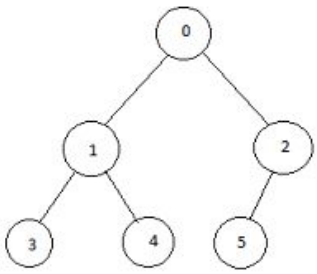


A Binary tree is Perfect Binary Tree in which all internal nodes have two children and all leaves are at the same level.

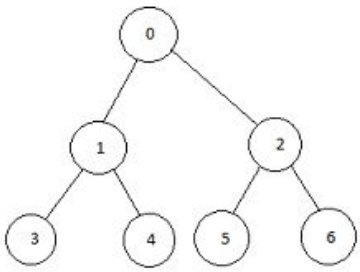
Here we can see a comparison of full, complete and perfect binary trees:



Full
Binary Tree



Complete
Binary Tree



Perfect
Binary Tree

10.3. BASIC OPERATIONS

Insertion: We use the class Tree that we defined above

```
def insert(root, node):
if root is None:                # create the root if there isn't one yet
root = node
else:
if root.data > node.data:               # value smaller than the root, go left
if root.left is None:           # if there is no left node, create one
root. left = node
else:                # if there already a left node, check the left subtree
insert(root.left , node)
else:                # value larger than the root, go right
if root.right is None:          # if there is no right node, create one
root. right = node
else:                # if there already a right node, check this subtree
insert(root.right , node)
```

Find minimum value: Due to the BST property, the minimum value will be the leftmost node.

```
def minimum_value(node):
current = node
while(current.left is not None):
current = current.left
return current
```

Delete: Search a certain value and delete that node, Pay attention as the root may change, and return it.

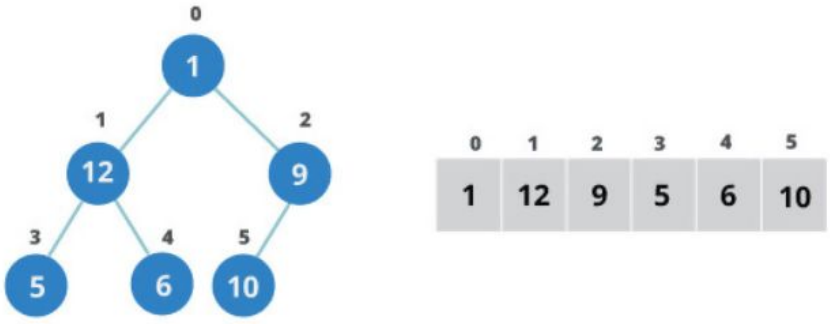
```
def delete(root, val):
if root is None:
return root

if val < root.val :             # If the val to be deleted is smaller than the root's
root.left = delete(root.left, val)        # then it lies in  left subtree

elif( val > root.val ):          # If the val to be delete is greater than the root's
root.right = delete(root.right, val)      # then it lies in right subtree

else:                # Else we need to delete the root
if root.left is None :           # if no left node
temp = root.right              # only the right node can be the new root
root = None
return temp

elif root.right is None :                # if no right node
temp = root.left             # only the left node can be the new root
root = None
return temp

else:        # if we have both children
# find the minimum of the right subtree, to become the new root. Remember              # that the right subtree has largest values from the root!
temp = minimum_value(root.right)
root. val = temp. val          # new root is the minimum value node
```

```
root.right = delete(root.right , temp.val)
return root
```

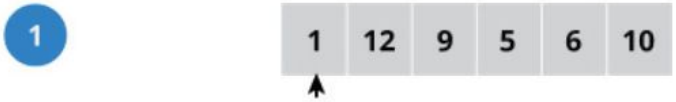
10.4. CREATE A COMPLETE BINARY TREE FROM AN UNSORTED LIST

This procedure can be very useful for sorting algorithms that make use of tree/heap structures.

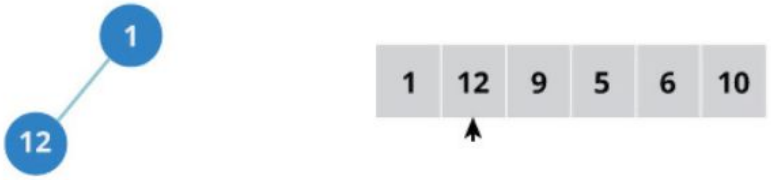
We need to transform the list on the right to the tree on the left side of the following image. The positions of the array are depicted on the tree:

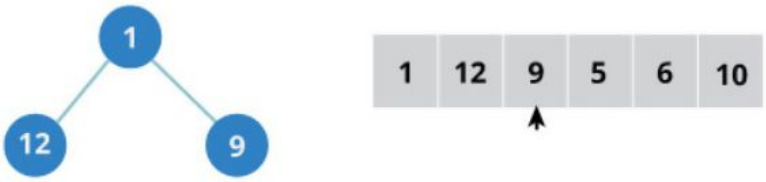


- Select first element of the list to be the root node. (First level - 1 element)

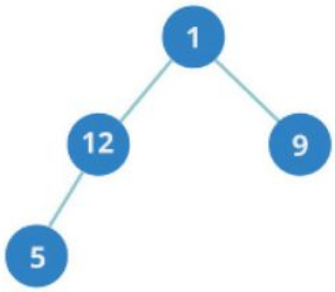


- Put the second element as a left child of the root node and the third element as a right child. (Second level - 2 elements)

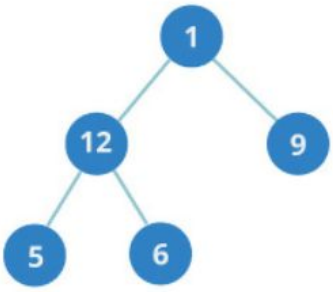




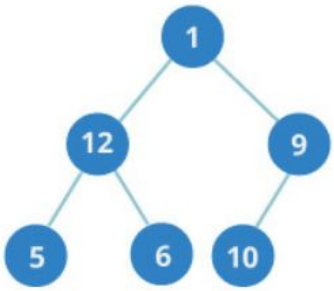
- Put next two elements as children of left node of second level. Again, put the next two elements as children of right node of second level (3rd level - 4 elements).


1
12
9
5


1 12 9 5 6 10


1
12
9
5
6


1 12 9 5 6 10


1
12
9
5
6
10


1 12 9 5 6 10

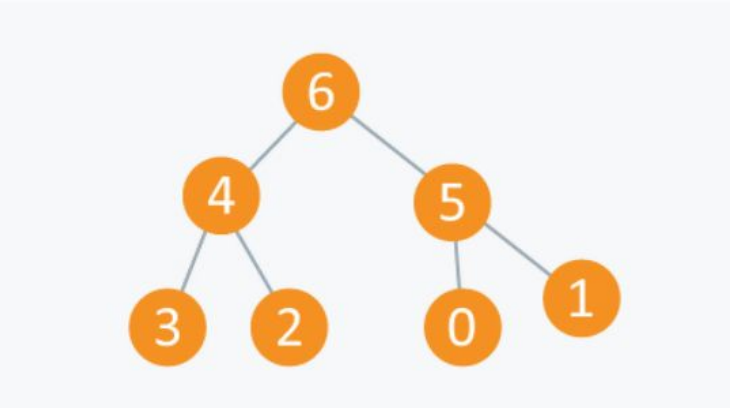
-

Keep repeating till you reach the last element.

HEAP

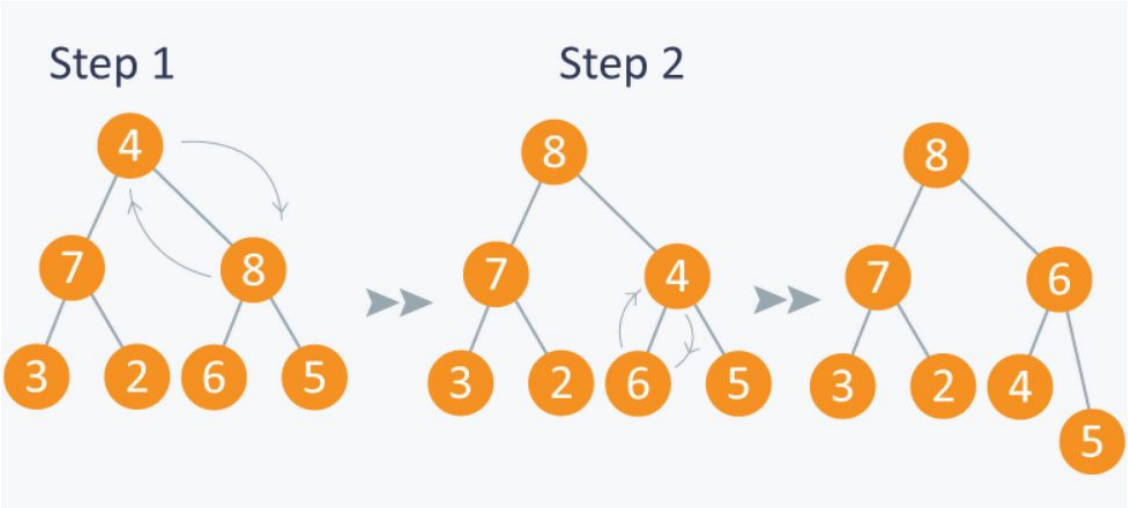
A heap is a tree-based data structure in which all the nodes of the tree are in a specific order.

For example, the order of the heap below is that every parent node has a greater value comparing to any of its children:

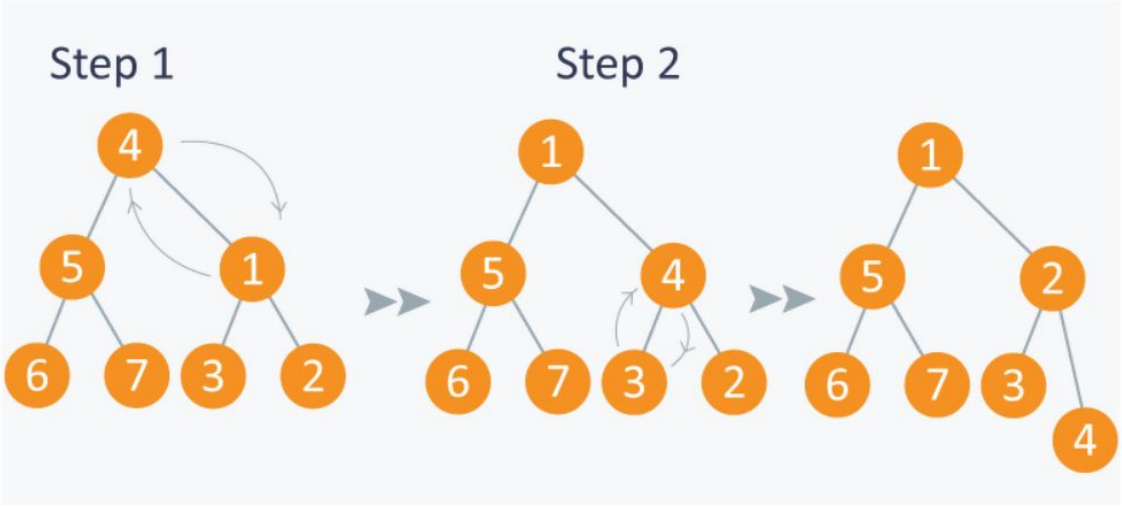


In general, there can be two types of heap:

Max Heap: In this type of heap, the value of parent node will always be greater than or equal to the value of child node across the tree and the node with highest value will be the root node of the tree. We can convert a heap to a max heap by swapping nodes according to the max heap property:

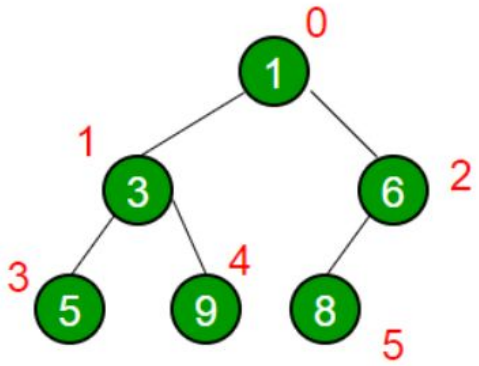


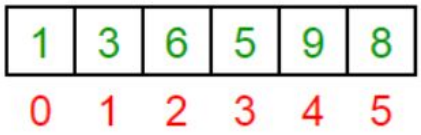
Min Heap: In this type of heap, the value of parent node will always be less than or equal to the value of child node across the tree and the node with lowest value will be the root node of tree. We can transform a heap into a min heap:



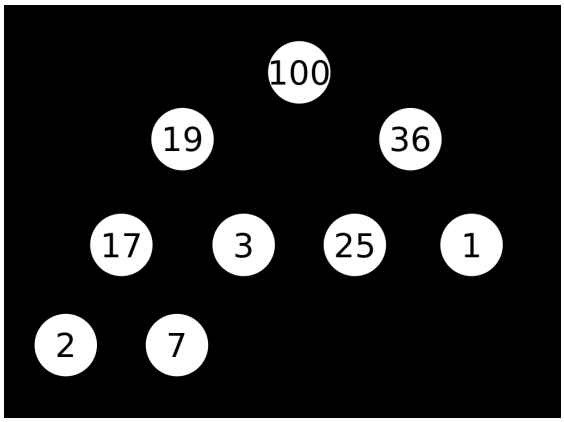
11.1. BINARY HEAPS

A Binary Heap is a Complete Binary Tree where items are stored in a special order such that value in a parent node is greater(or smaller) than the values in its two children nodes. The former is called as max heap and the latter is called min heap. The heap can be represented by binary tree or array:

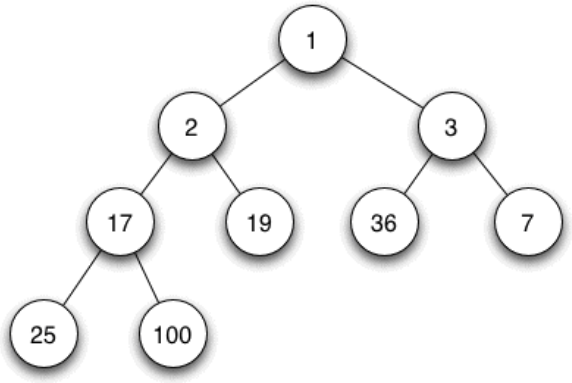




Here we can see a complete binary max heap:



And a complete binary min heap:



Binary heaps are going to be used in heap sort later on, achieving a complexity of O(nlogn). Binary heaps are also a common way of implementing priority queues.

ALGORITHMS

SORTING

Sorting algorithms have the task to put elements of a list or an array in a certain order. The most frequently sorting algorithms focus on numerical sort or lexicographical sort. The reason why there are many algorithms developed for sorting is because the computational complexity plays a major role in many real life applications, and the way we choose to sort a list affects significantly the complexity.

The most preferred and achievable complexity is O(nlogn), while an ideal complexity of O(n) is rarely achieved. Very simple algorithms (like Bubblesort-explained below) have a rather bad performance $O(n^2)$, so they are presented as introductory algorithms for teaching purposes because of their simplicity.

Comparison based algorithms are more intuitive, as they perform the obvious functionality of comparing the elements of the list between them and then rearrange them if required. There are also non-comparison based algorithms (counting sort, radix sort, bucket sort etc), which make assumptions for the data to be sorted and they achieve better complexity in general.

1.1. PYTHON SORTING SYNTAX

In Python sorting is a very simple task. Lets assume we have a list of integers:

a = [5,8,2,4,1]

We can sort this list in ascending order:

a.sort()

So, now our list will be:

a = [1,2,4,5,8]

If we want to sort this list in descending order:

a.sort(reverse = True)

We will get:

a = [8,5,4,2,1]

Note that the sort() method modifies the list itself and does not create a new one. It can be performed on lists only. If we want to sort any iterable (like dictionaries), then we need to use another method called sorted(). Regarding our initial list above, we can sort it with sorted():

```
sorted(a)
```

The result will be a new list, the sorted copy of the list a. So, we need to assign the result of this function call in order to use it later:

```
b = sorted(a)
b = [1,2,4,5,8]
```

Python offers the flexibility to customize any sorting task by modifying the parameters of sort() and sorted() as we wish. After you become familiar with basic sorting task, you can refer to the Python documentation https://docs.python.org/3.3/howto/sorting.html for complex customized sortings.

1.2. BUBBLE SORT

Bubble sort is the simplest in terms of understanding sorting algorithm. It compares each successive pair of elements in an unordered list and swaps them if they are not in order. Lets consider the list below:

[6,5,3,1,8,7,2,4]

It will start from the first 2 elements, 6 and 5, With ** we note the elements that are examined in this step:

[**5,6**,3,1,8,7,2,4] -- 5 < 6 -> swap
[5,**3,6**,1,8,7,2,4] -- 3 < 6 -> swap
[5,3,**1,6**,8,7,2,4] -- 1 < 6 -> swap
[5,3,1,**6,8**,7,2,4] -- 8 > 6 -> no swap
[5,3,1,6,**7,8**,2,4] -- 7 < 8 -> swap
[5,3,1,6,7,**2,8**,4] -- 2 < 8 -> swap
[5,3,1,6,7,2,**4,8**] -- 4 < 8 -> swap

When we reach the end of the list, the first parsing is finished. Our greatest element (8) has successfully reached the last position of the list. We need to repeat this process for the rest of the elements, therefore n-1 more times (considering the initial list is of length n). Bubblesort works as if all elements were unsorted, that is why it performs n times the same process of comparing n-1 pairs.

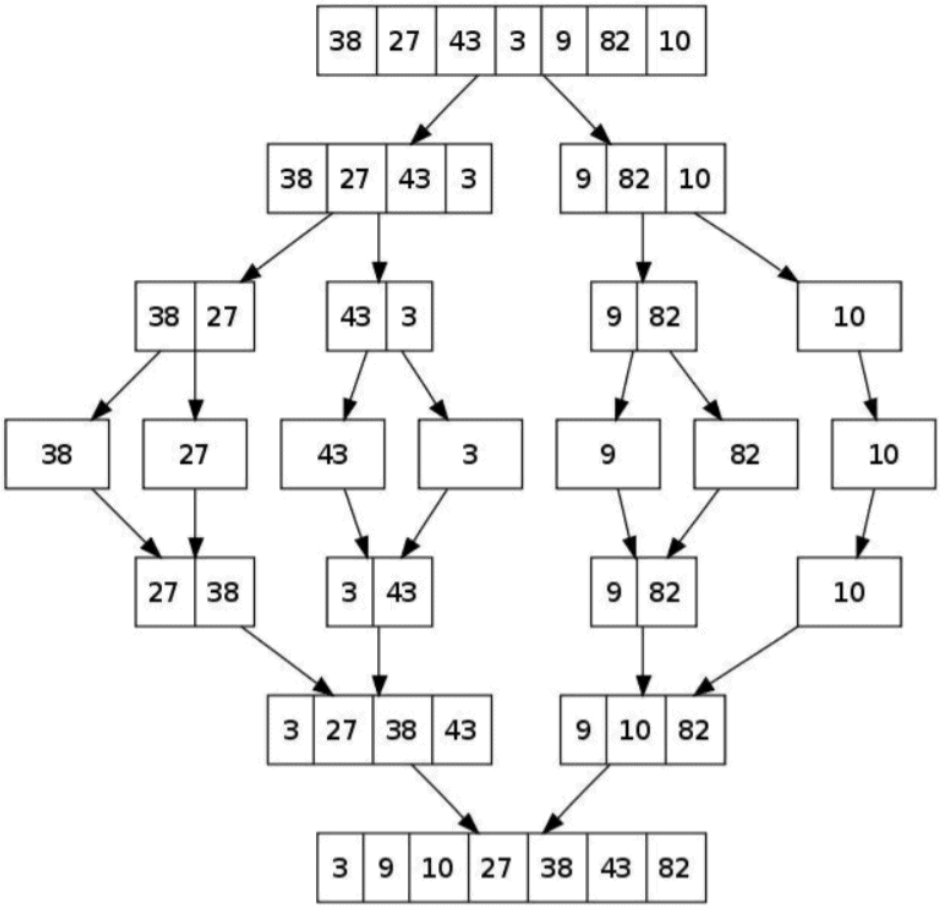
We can prune some unecessary repeats if we have no more swaps. This is not going to help much, as the asymptotic complexity will still be $O(n^2)$.

```
def bubblesort(input_list):
for i in range(len(input_list)):
```

```
for j in range(i):
if int(input_list[j]) > int(input_list[j+1]):
input_list[j],input_list[j+1] = input_list[j+1],input_list[j]
return input_list
```

1.3. MERGE SORT

Mergesort divides the given input and handles the problem in smaller parts. This is the divide and conquer technique that we are going to explain later on. Given a list of length n, Mergesort divides it in 2 equal parts until it gets n lists of length 1. Then, pairs of lists are merged together with the smaller first element among the pair of lists being added in each step. Through successive merging and through comparison of first elements, the sorted list is built. We can see graphically how the list [38,27,43,3,9,82,10] is sorted:




```
def merge(X, Y):
    p1 = p2 = 0
    out = []
    while p1 < len(X) and p2 < len(Y):
    if X[p1] < Y[p2]:
    out.append(X[p1])
    p1 += 1
    else:
    out.append(Y[p2])
    p2 += 1
    out += X[p1:] + Y[p2:]
    return out

    def mergeSort(A):
    iIf len(A) <= 1:
    return A
```

```
if len(A) == 2:
return sorted(A)
# divide in half
mid = len(A) / 2
# [:mid] contains the elements unil mid
# [mid:] contains elements after mid
# recursively divide the elements, and merge the sublists
return merge(mergeSort(A[:mid]), mergeSort(A[mid:]))
```

1.4. QUICKSORT

Quicksort is a sorting algorithm that picks an element (the pivot) and reorders the list forming two partitions such that all elements less than the pivot come before it and all elements greater come after. The algorithm is then applied recursively to the partitions until the list is sorted. There are different approaches regarding the choice of pivot, like selecting the last element or a random element of the list.

```
def quicksort(arr):
if len(arr) <= 1:              # Base case: if we have only one element
return arr
pivot = arr[len(arr) / 2]              # take middle element

# left partition
left = [x for x in arr if x < pivot]              # takes all elements x of value smaller than pivot
middle = [x for x in arr if x == pivot]   # takes elements x equal to pivot

# right partition
right = [x for x in arr if x > pivot]             # takes elements x greater than pivot

# recursively applies the same steps on the left and right partitions
return quicksort(left) + middle + quicksort(right)
```

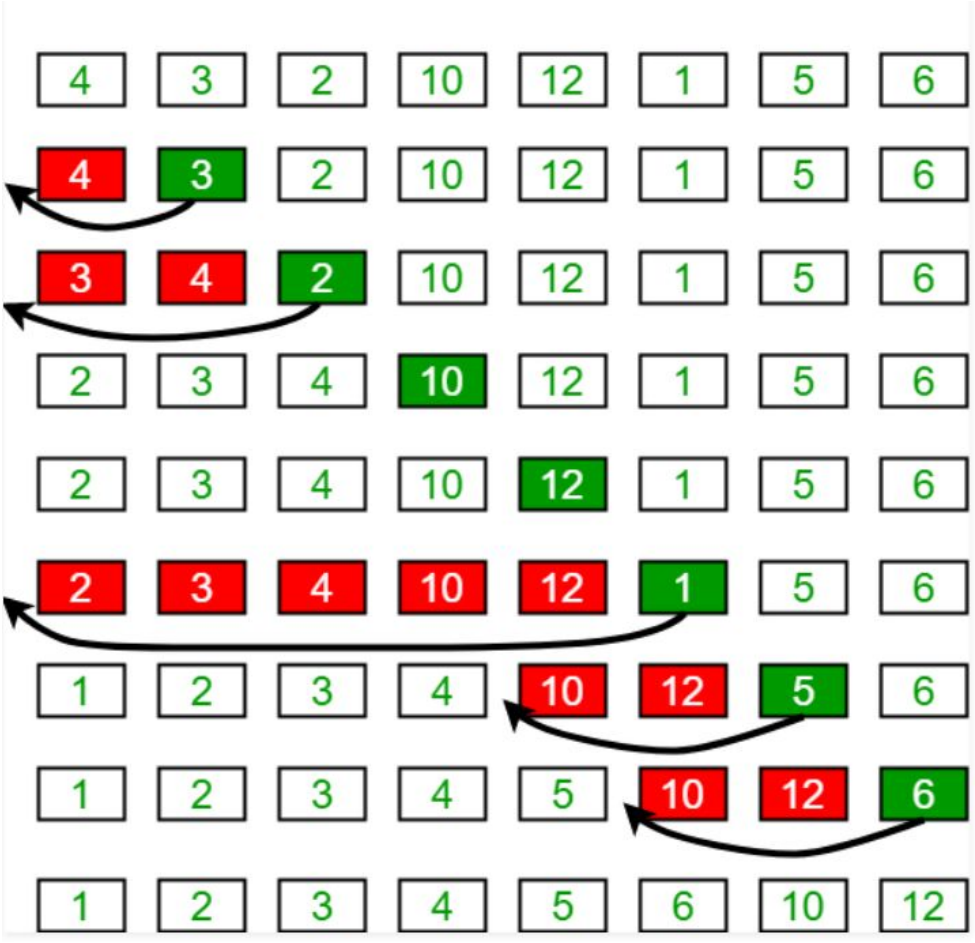
1.5. INSERTION SORT

Insertion sort implements the logic of considering a part of a list sorted and another part unsorted. We start taking the very first element of our list at position 0. We consider this item sorted and the rest n-1 items unsorted. We move to the 2nd element (position 1) and we compare with the 1st one. If it is smaller, we bring it before the 1st element, otherwise we

left it as it is. Now, we have these 2 sorted elements and n-2 unsorted. We continue in the same way by proceeding step by step to the end of our list.

Insertion sort is used when number of elements is small. It can also be useful when input array is almost sorted, only few elements are misplaced in complete big array.

Lets consider the following list:
[4,3,2,10,12,1,5,6]
We can see graphically how our elements move in the list in order to get sorted. Green is our current element, and the sublist on its left is sorted, while the other sublist on its right is unsorted. If an element of the unsorted part is smaller than the current one, it needs to be inserted on its right position by moving from where it is.



```
def insertionSort(arr):
for i in range(1, len(arr)):
# the ‘green’ element of above
key = arr[i]

# Move elements of arr[0..i-1], that are greater than key, one position to the right each time
```

```
    j = i-1
    # While the current element j is greater than the key (and we haven't reached the end of the list)
    while key < arr[j] and j >=0:
    arr[j+1] = arr[j]              # bring the 'next' j+1 element one step to the left
    j -= 1              # go one cell left
    arr[j+1] = key              # and finally bring the key one step right
    # and go for the next key
```

1.6. BUCKET SORT

Bucket sort is useful when we know the distribution of our input data (i.e. uniformly distributed into a certain range). This distribution highly affects the computational complexity of the algorithm, so it should be used only in cases that each bucket can contain almost the same number of elements. It requires more memory to temporarily store the elements. We consider a list of length n and k buckets. Each bucket can accept elements within a certain range. The elements are placed in the buckets and are sorted internally using a sorting method described above. In the end, the items of the buckets are combined to form the final sorted list.

Lets consider the list:

[11,9,21,8,17,19,13,1,24,12]

(It is obvious that our items are within the range (0,25))

and k = 5 buckets with the following ranges:

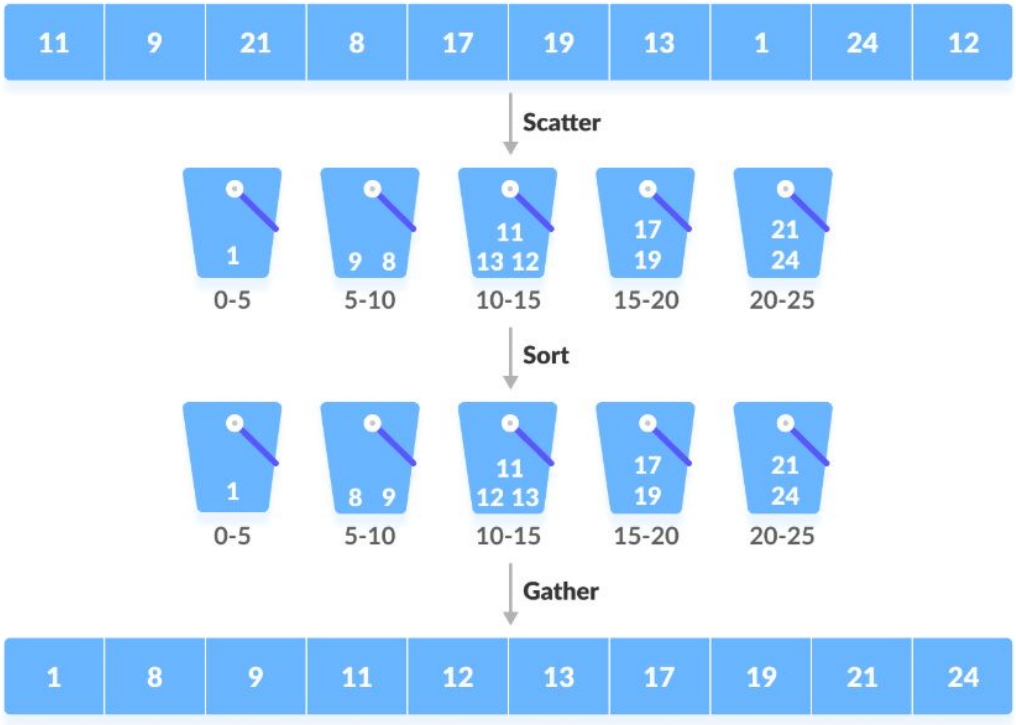
bucket 1 accepts elements from 0-5

bucket 2 accepts elements from 5-10

bucket 3 accepts elements from 10-15

bucket 4 accepts elements from 15-20

bucket 5 accepts elements from 20-25




```
def bucketSort(array):
bucket = []
for i in range(len(array)):
bucket.append([])

for j in array:
index_b = int(10 * j)
bucket[index_b].append(j)

# sort the bucket
for i in range(len(array)):
bucket[i] = sorted(bucket[i])

# put items from the buckets together in the final array
k = 0
for i in range(len(array)):
for j in range(len(bucket[i])):
array[k] = bucket[i][j]
```

```
    k += 1
    return array
```

1.7. SELECTION SORT

The algorithm divides the input list into two parts: the sublist of items already sorted, which is built up from left to right at the front (left) of the list, and the sublist of items remaining to be sorted that occupy the rest of the list. It reminds us of the insertion sort algorithm.

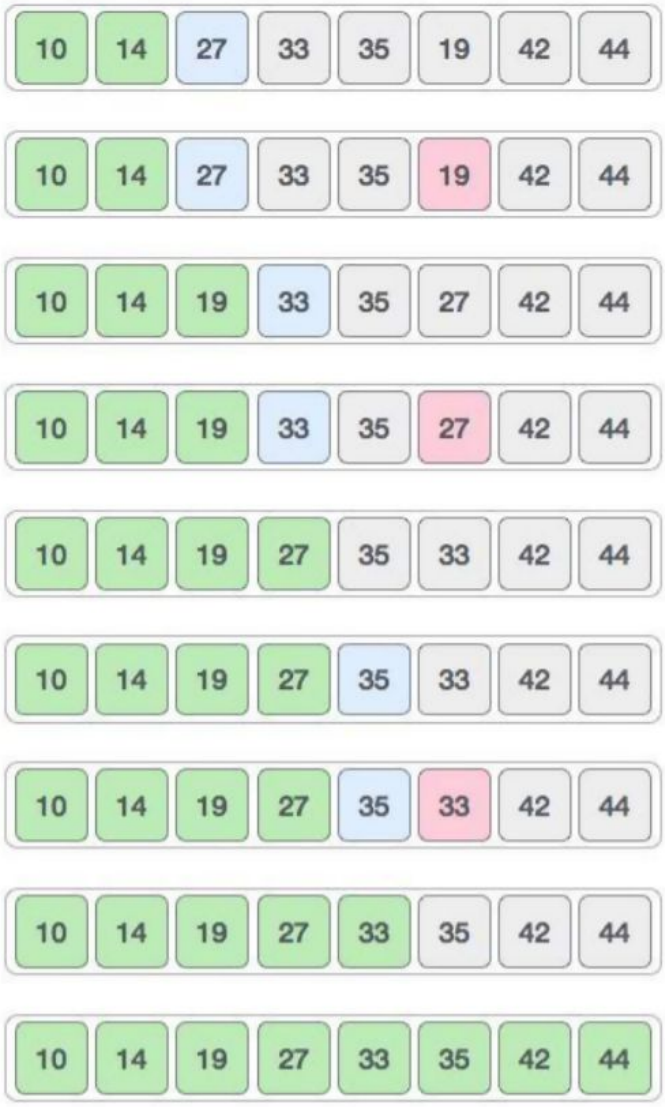
Initially, the sorted sublist is empty and the unsorted sublist is the entire input list. The algorithm proceeds by finding the smallest element in the unsorted sublist, exchanging (swapping) it with the leftmost unsorted element (putting it in sorted order), and moving the sublist boundaries one element to

the right. (Remember in insertion sort we were moving the elements within the array until their right position, here we are swapping a pair of elements to bring the smallest of the two in the left position of this pair).

It has $O(n^2)$ time complexity, making it inefficient on large lists, and generally performs worse than the similar insertion sort. Selection sort is noted for its simplicity, and it has performance advantages over more complicated algorithms in certain situations, particularly where auxiliary memory is limited.

We consider the following list:
[10, 14, 27, 33, 35, 42, 44]

With green we denote the sorted part, with gray the unsorted part of the list and with red the smallest element of the unsorted part. With blue we note the item that will be swapped with the red one.


10 14 27 33 35 19 42 44
10 14 27 33 35 19 42 44
10 14 19 33 35 27 42 44
10 14 19 33 35 27 42 44
10 14 19 27 35 33 42 44
10 14 19 27 35 33 42 44
10 14 19 27 35 33 42 44
10 14 19 27 33 35 42 44
10 14 19 27 33 35 42 44

1.8. HEAP SORT

Heap sort is a comparison based sorting technique on binary heap data structure. It is similar to selection sort in which we first find the maximum element and put it at the end of the data structure. Then repeat the same process for the remaining items.

This can be achieved by utilizing the max heap (to sort in increasing order). We know that the largest item is on the root of the heap.

Heapify procedure can be applied to a node only if its children nodes are heapified. So the

heapification must be performed in the bottom up order.

heapify subtree rooted at index i. n is size of heap

```
def heapify(arr, n, i):
largest = i          # Initialize largest as root
l = 2 * i + 1              # left = 2*i + 1
r = 2 * i + 2              # right = 2*i + 2

# See if left child of root exists and is greater than root
if l < n and arr[i] < arr[l]:
largest = l

# See if right child of root exists and is greater than root
if r < n and arr[largest] < arr[r]:
largest = r

# Change root, if needed
if largest != i:
arr[i],arr[largest] = arr[largest],arr[i] # swap

# Heapify the root.
heapify(arr, n, largest)
```

```
# The main function to sort an array of given size
def heapSort(arr):
n = len(arr)

# Build a maxheap.
for i in range(n, -1, -1):
heapify(arr, n, i)

# One by one extract elements
for i in range(n-1, 0, -1):
arr[i], arr[0] = arr[0], arr[i] # swap
heapify(arr, i, 0)
```

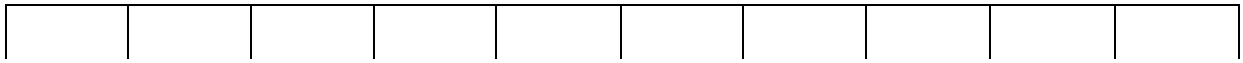
1.9. COUNTING SORT

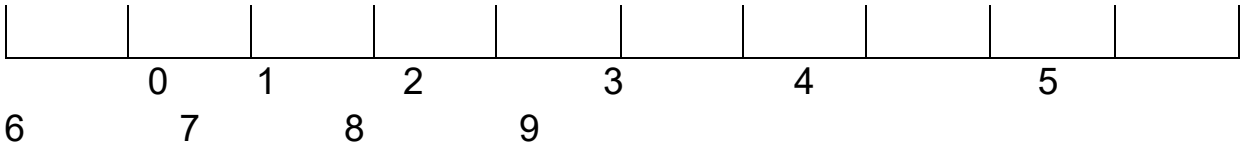
Counting sort is an integer sorting algorithm for a collection of objects that sorts according to the keys of the objects. It takes advantage of situations where we know the range of elements that must be sorted (for example numbers between 0 and 10), and those elements are integers. The algorithm counts the unique objects and uses some math to define their order. Lets assume that we have an input list or array:

a = [9,4,1,7,9,1,2,0]

Notice that our numbers belong in range [0,9]

1. Construct a working array C that has size equal to the range of the input array a. In our case the range is [0,9], so we need an array C of 10 positions (indices will go from 0 to 9).





2. Iterate through a assigning C[x] based on the number of times x appeared in A. In other words, count how many times each number appears in a: how many 0s, how many 1s, how many 2s etc. Populate the 1st cell of C (position 1), the 2nd cell (position 2), the third cell (position 3) etc respectively.

C = [1,2,1,0,1,0,0,1,0,2]

3. Transform C into an array where C[x] refers to the number of values ≤ x by iterating through the array, assigning to each C[x] the sum of its prior value and all values in C that come before it.

The first element of the new array will be the same as before:

C_new = [1,2,1,0,1,0,0,1,0,2]

Continuing with the first 2 elements, we get:

1 + 2 = 3

and we change the 2nd element:

C_new = [1,3,1,0,1,0,0,1,0,2]

Now we need to add the 2nd and the 3rd element:

1 + 3 = 4

4 is going to be the new 3rd element:

C_new = [1,3,4,0,1,0,0,1,0,2]

3rd and 4th element:

4 + 0 = 4

4 is going to be the new 4th element.

C_new = [1,3,4,4,1,0,0,1,0,2]

4th and 5th:

4 + 1 = 5

C_new = [1,3,4,4,5,0,0,1,0,2]

If we continue this we will end up with:
C_new = [1,3,4,4,5,5,5,6,6,8]

4. Shift our new array one step right. The first element will be 0 and 8 disappears:
C_new = [0,1,3,4,4,5,5,5,6,6]

The value at each index tells us where the first instance of that integer will appear in our sorted array. For example, the value at index 9 in our count array is 6. In other words, this tells us that the first instance of 9 will occur at index 6 of our sorted array. Another way to think about this is that there were be six elements that will appear before the number 9 in our sorted array.

5. Translate this count array in a sorted array: iterate through our original array, and find the index that corresponds to the value we're looking at. Then, we'll look at the tallied count at that index. Our original array was:
a = [9,4,1,7,9,1,2,0]

We start looking at our first element in the original array: 9. We'll go to our count array, find index 9, and the value at that that index. In this case, it's the number 6. C_new is:

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

We'll find the index 6 in our new array, and put a 9 there.
a_sorted = [_,_,_,_,_,_,9,_]
And we'll increment the value at the corresponding index in the count array. We do this increment step after we've sorted the first instance of the number 9:
C_new = [0,1,3,4,4,5,5,5,6,**7**]

We can see how this same step is repeated with the next couple elements in the array, and how each time, as we sort, we increment the number at the corresponding index in the count array. The increment step is crucial in order to be able to sort duplicate values, and in order to maintain the order of those elements as they appear in the unsorted array.

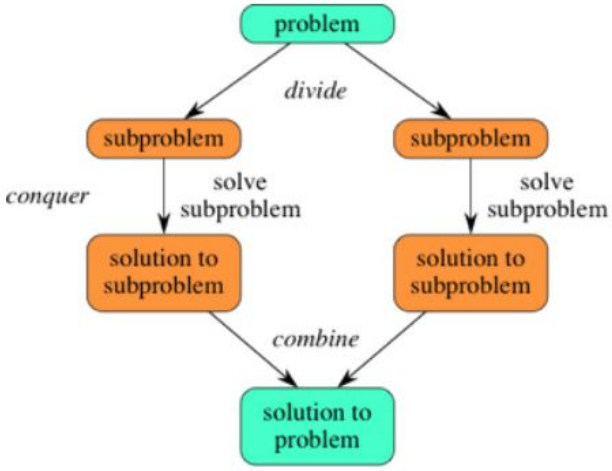
Eventually, if we continue doing this as we iterate through the entire array, we'll end up with a new, sorted array, that looks like this:

a_sorted = [0,1,1,2,4,7,9,9]

2. DIVIDE AND CONQUER

A typical Divide and Conquer algorithm solves a problem using following three steps.

1. Divide: Break the given problem into subproblems of same type.
2. Conquer: Recursively solve these subproblems
3. Combine: Appropriately combine the answers

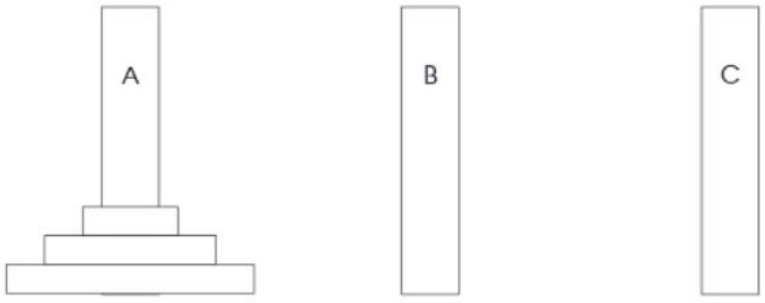


Because divide-and-conquer creates at least two subproblems, a divide-and-conquer algorithm makes multiple recursive calls.

As we saw before, both mergesort and quicksort use that technique of solving subproblems and combine their solutions.

2.1. TOWERS OF HANOI

The Towers of Hanoi is a mathematical problem which consists of 3 pegs and a number of discs. In this instance, 3 discs.



Each disc is a different size. We want to move all discs to peg C so that the largest is on the bottom, second largest on bottom — 1 etc. The requirements are:

- We can only move 1 disc at a time.
- A disc cannot be placed on top of other discs that are smaller than it.
- We want to use the smallest number of moves possible.

So, we can start building our solution:

If we have 1 disc, we only need to move it once.

If we have 2 discs, we need to move it 3 times.

We need to store the smallest disc in a buffer peg (1 move), move the largest disc to peg c (2 moves) and move the buffer disc to peg c (3 moves).

When we have 4 discs we need to make 15 moves. 5 discs is 31.

These move numbers are powers of 2 minus 1.

To find out how many moves a Tower of Hanoi solution takes you calculate (2^n)-1 where n is how many discs there are.

Notice how we need to have a buffer to store the discs.

We can generalise this problem.

If we have n discs: move n-1 from A to B recursively, move largest from A to C, move n-1 from B to C recursively.

If there is an even number of pieces the first move is always into the middle. If there are an odd number of pieces the first move is always to the other end.

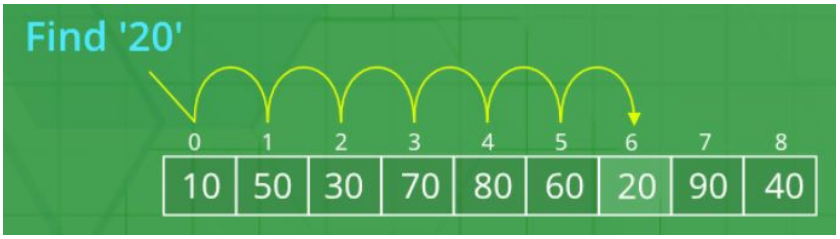
SEARCH

3.1. LINEAR SEARCHING

Given an array arr[] of n elements, we are looking for a given target element x in arr[].

Linear search is the simplest search algorithm.

We start from the leftmost cell of arr[] and we compare it to the target x. If arr[0]==x we found it. Otherwise, we continue with the next element arr[1]. We continue the same process until we find it. If we reach the end of the array at arr[n-1] and there was no element arr[i] at any position I where arr[i]==x, then the element does not exist in arr.



In the worst case, out element will be in the n-1 position or it will not exist at all, so we will need n steps before we can answer if it

exists or not. This results in O(n) complexity.

Code in python:

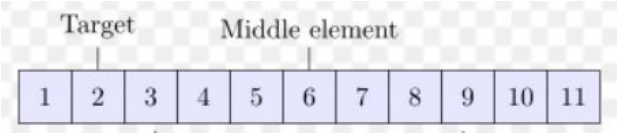
```
def lin_search(arr, n, x):
for i in range (0, n):
if (arr[i] == x):
return i;
return -1;              # if it doesn't exist
```

3.2. BINARY SEARCHING

Binary Search is a Divide and Conquer search algorithm. It uses O(log n) time to find the location of an element in a search space where n is the size of the search space. To use Binary Search, the search space must be ordered (sorted) in some way. Binary Search works by halving the search space at each iteration after comparing the target value to the middle value of the search space. If the target value is smaller than the mid value, then we need to examine only the lowest half of our search space. If the target value is larger than the middle value we need only to check the upper half. If target value is equal to the mid, then we can return what we were looking for.

Duplicate entries (ones that compare as equal according to the comparison function) cannot be distinguished, though they don't violate the Binary Search property.

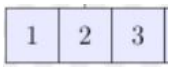
Lets have a look at the example below, where we are searching number 2. The mid element is number 6. So, if we compare the target with the mid, 2<6. So we need to continue to the left part, where all numbers are smaller than 6.



So, we continue with this sub-array:

Here we have an even length, so we need to predefine if we take the left or the right element to the middle (in this case if we take

3 or 4 as mid). Lets take 3 as the middle. 2<3, so we continue with the left sub-array:



Now the mid is 2, so 2=2 and we found our target!

Python code:

```
def binarySearch (arr, l, r, target):

# Check base case
if r >= l:
# find the mid
mid = l + (r - l)/2

# If the target equals to the value of the mid we found it
if arr[mid] == target:
return mid

# If element is smaller than mid, then it
# can only be present in left subarray
elif arr[mid] > target:
# Recursively call the function with the lower half of the array
return binarySearch(arr, l, mid-1, target)

# Else the element can only be present
# in right subarray
else:
return binarySearch(arr, mid + 1, r, target)

else:
# Element is not present in the array
return -1
```

GREEDY ALGORITHMS

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 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. The solution that this algorithm builds is the sum of all 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.

So, greedy algorithms work on problems for which in every step there is a choice that is optimal for the problem up to that step and after the last step the algorithm produces the optimal solution of the entire problem.

To make a greedy algorithm, identify an optimal substructure or subproblem in the problem. Then, determine what the solution will include (for example, the largest sum, the shortest path, etc.).

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.

4.1. JOB SEQUENCING

Lets assume that we have a set of things to do (activities). Each activity has a start time and an end time. You aren't allowed to

perform more than one activity at a time. Your task is to find a way to perform the maximum number of activities.

For example, suppose you have a selection of classes to choose from.

Activity No.	start time	end time
1	10.20 A.M	11.00AM
2	10.30 A.M	11.30AM
3	11.00 A.M	12.00AM
4	10.00 A.M	11.30AM
5	9.00 A.M	11.00AM

So what's the maximum number of classes we can take without any overlap?

Lets think for the solution by greedy approach.First of all we randomly chose some approach and check that will work or not.

1. Sort the activity by start time that means which activity start first we will take them first. then take first to last from sorted list and check it will intersect from previous taken activity or not. If the current activity is not intersect with the previously taken activity, we will perform the activity otherwise we will not perform. This approach will work for some cases like:

Activity No.	start time	end time
1	11.00 A.M	1.30P.M
2	11.30 A.M	12.00P.M
3	1.30 P.M	2.00P.M
4	10.00 A.M	11.00AM

the sorting order will be 4-->1-->2-->3 .The activity 4--> 1--> 3 will be performed and the activity 2 will be skipped. the maximum 3 activity will be performed. It works for this type of cases. but it will fail for some cases. Lets apply this approach for the case:

Activity No.	start time	end time
1	11.00 A.M	1.30P.M
2	11.30 A.M	12.00P.M
3	1.30 P.M	2.00P.M
4	10.00 A.M	3.00P.M

The sort order will be 4-->1-->2-->3 and only activity 4 will be performed but the answer can be activity 1-->3 or 2-->3 will be performed. So our approach will not work for the above case. Let's try another approach:

2. Sort the activity by time duration that means perform the shortest activity first. that can solve the previous problem.

Activity No.	start time	end time
1	6.00 A.M	11.40A.M
2	11.30 A.M	12.00P.M
3	11.40 P.M	2.00P.M

if we sort the activity by time duration the sort order will be 2--> 3 --->1 . and if we perform activity No. 2 first then no other activity can be performed. But the answer will be perform activity 1 then perform 3 . So we can perform maximum 2 activities. So this can not be a solution of this problem. We should try a different approach.

The solution

Sort the Activity by ending time that means the activity finishes first that come first. the algorithm is given below

1. Sort the activities by its ending times.
2. If the activity to be performed do not share a common time with the activities that previously

performed, perform the activity.

Activity No.	start time	end time

1	10.20 A.M	11.00AM
2	10.30 A.M	11.30AM
3	11.00 A.M	12.00AM
4	10.00 A.M	11.30AM
5	9.00 A.M	11.00AM

So sort order will be 1-->5-->2-->4-->3.. the answer is 1-->3 these two activities will be performed.

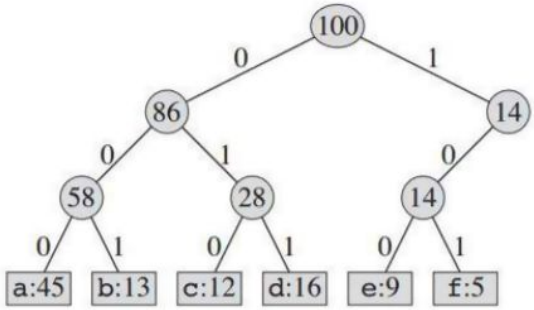
4.2. HUFFMAN CODING

Huffman code is a particular type of optimal prefix code that is commonly used for lossless data compression. It compresses data very effectively saving from 20% to 90% memory, depending on the characteristics of the data being compressed. We consider the data to be a sequence of characters. Huffman's greedy algorithm uses a table giving how often each character occurs (i.e., its frequency) to build up an optimal way of representing each character as a binary string. Huffman code was proposed by David A. Huffman in 1951.

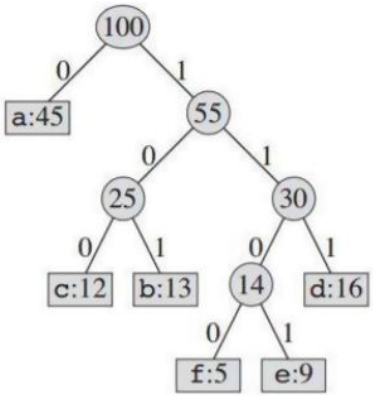
Suppose we have a 100,000-character data file that we wish to store compactly. We assume that there are only 6 different characters in that file. The frequency of the characters are given by:

Character	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5

We have many options for how to represent such a file of information. Here, we consider the problem of designing a Binary Character Code in which each character is represented by a unique binary string, which we call a codeword.



Fixed-length Codeword



Variable-length Codeword

The constructed tree will provide us with:

Character	a	b	c	d	e	f
Fixed-length Codeword	000	001	010	011	100	101
Variable-length Codeword	0	101	100	111	1101	1100

If we use a fixed-length code, we need three bits to represent 6 characters. This method requires 300,000 bits to code the entire file.

A variable-length code can do considerably better than a fixed-length code, by giving frequent characters short codewords and infrequent characters long codewords. This code requires: (45 X 1 + 13 X 3 + 12 X 3 + 16 X 3 + 9 X 4 + 5 X 4) X 1000 = 224000 bits to represent the file, which saves approximately 25% of memory.

One thing to remember, we consider here only codes in which no codeword is also a prefix of some other codeword. These are called prefix codes. For variable-length coding, we code the 3-

character file abc as 0.101.100 = 0101100, where "." denotes the concatenation.

Prefix codes are desirable because they simplify decoding. Since no codeword is a prefix of any other, the codeword that begins an encoded file is unambiguous. We can simply identify the initial codeword, translate it back to the original character, and repeat the decoding process on the remainder of the encoded file. For example, 001011101 parses uniquely as 0.0.101.1101, which decodes to aabe. In short, all the combinations of binary representations are unique. Say for example, if one letter is denoted by 110, no other letter will be denoted by 1101 or 1100. This is because you might face confusion on whether to select 110 or to continue on concatenating the next bit and select that one.

Compression Technique:

The technique works by creating a binary tree of nodes. These can stored in a regular array, the size of which depends on the number of symbols, n. A node can either be a leaf node or an internal node. Initially all nodes are leaf nodes, which contain the symbol itself, its frequency and optionally, a link to its child nodes. As a convention, bit '0' represents left child and bit '1' represents right child. Priority queue is used to store the nodes, which provides the node with lowest frequency when popped. The process is described below:

1. Create a leaf node for each symbol and add it to the priority queue.

2. While there is more than one node in the queue:

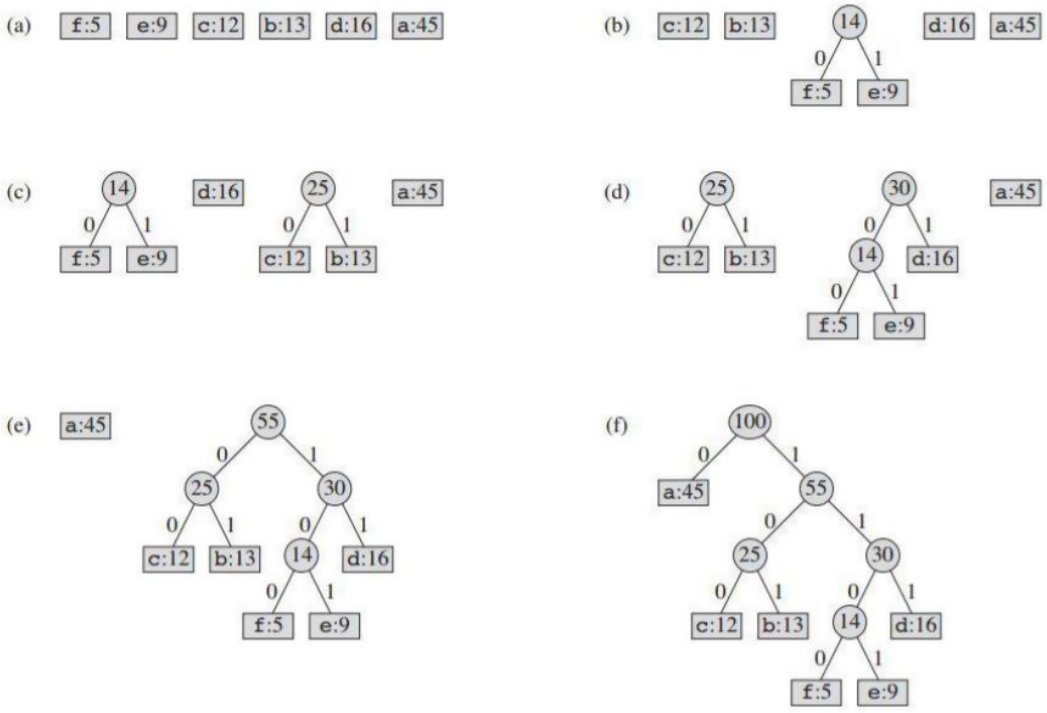
2. 1. Remove the two nodes of highest priority from the queue.

2. 2. Create a new internal node with these two nodes as children and with frequency equal to the sum of the two nodes' frequency.

2. 3. Add the new node to the queue.

3. The remaining node is the root node and the Huffman tree is complete.

For our example:



Decompression Technique:

The process of decompression is simply a matter of translating the stream of prefix codes to individual byte value, usually by traversing the Huffman tree node by node as each bit is read from the input stream. Reaching a leaf node necessarily terminates the search for that particular byte value. The leaf value represents the desired character. Usually the Huffman Tree is constructed using statistically adjusted data on each compression cycle, thus the reconstruction is fairly simple. Otherwise, the information to reconstruct the tree must be sent separately.

Greedy Explanation:

Huffman coding looks at the occurrence of each character and stores it as a binary string in an optimal way. The idea is to assign variable-length codes to input input characters, length of the assigned codes are based on the frequencies of corresponding characters. We create a binary tree and operate on it in bottom-up

manner so that the least two frequent characters are as far as possible from the root. In this way, the most frequent character gets the smallest code and the least frequent character gets the largest code.

4.3. OFFLINE CACHING

The caching problem arises from the limitation of finite space. Lets assume our cache C has k pages. Now we want to process a sequence of m item requests which must have been placed in the cache before they are processed. Of course if m<=k then we just put all elements in the cache and it will work, but usually is m>>k.

We say a request is a cache hit, when the item is already in cache, otherwise its called a cache miss. In that case we must bring the requested item into cache and evict another, assuming the cache is full. The goal is an eviction schedule that minimizes the number of evictions.

For the following examples we evict the page with the smallest index, if more than one page could be

evicted.

There are numerous greedy strategies for this problem, for example:

FIFO

Let the cache size be k=3 the initial cache a,b,c and the request a,a,d,e,b,b,a,c,f,d,e,a,f,b,e,c:

Request	a a d e b b a c f d e a f b e c
cache 1	a a d d d d a a a d d d f f f c
cache 2	b b b e e e e c c c e e e b b b
cache 3	c c c c b b b b f f f a a a e e
cache miss	x x x x x x x x x x x x x x x

Thirteen cache misses by sixteen requests does not sound very optimal

LFD

Let the cache size be k=3 the initial cache a,b,c and the request a,a,d,e,b,b,a,c,f,d,e,a,f,b,e,c:

Request	a a d e b b a c f d e a f b e c
cache 1	a a d e e e e e e e e e e e e c
cache 2	b b b b b b b a a a a a a f f f f
cache 3	c c c c c c c c c f d d d d b b b
cache miss	x x x x x x x x

Eight cache misses is a lot better.

The LFD strategy is optimal, but there is a big problem. Its an optimal offline solution. In praxis caching is usually an online problem, that means the strategy is useless because we cannot now the next time we need a particular item.

DYNAMIC PROGRAMMING

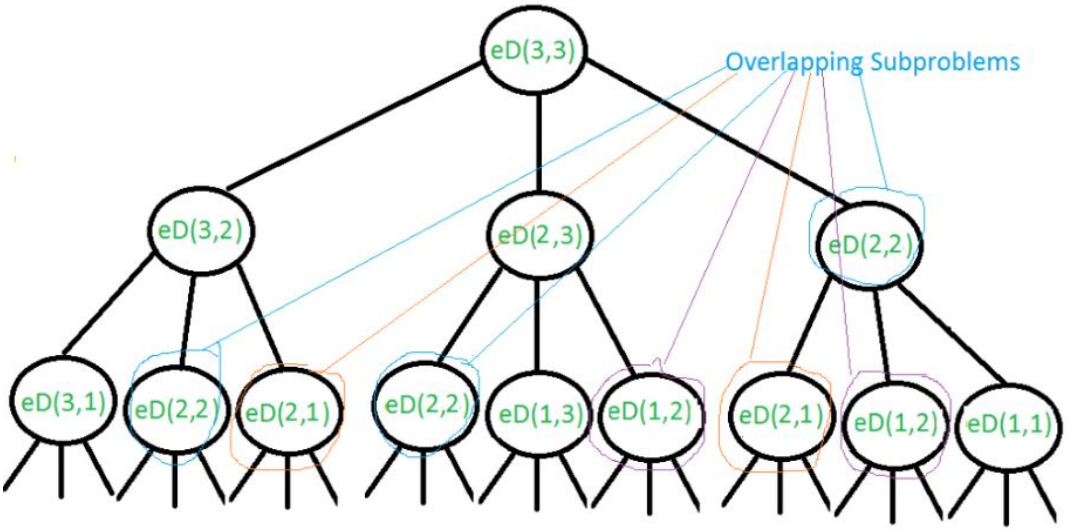
Dynamic programming is a widely used concept and its often used for optimization. It refers to simplifying a complicated problem by breaking it down into simpler sub-problems in a recursive manner usually a bottom-up approach. There are two key attributes that a problem must have in order for dynamic programming to be applicable "Optimal substructure" and "Overlapping sub-problems".

A problem is said to have **optimal substructure** if an optimal solution can be constructed efficiently from optimal solutions of its subproblems. Typically, a greedy algorithm is used to solve a problem with optimal substructure. Otherwise, provided the problem exhibits overlapping subproblems as well, dynamic programming is used.

Example:

Consider finding a shortest path for travelling between two cities by car. Such an example is likely to exhibit optimal substructure. That is, if the shortest route from Seattle to Los Angeles passes through Portland and then Sacramento, then the shortest route from Portland to Los Angeles must pass through Sacramento too. That is, the problem of how to get from Portland to Los Angeles is nested inside the problem of how to get from Seattle to Los Angeles.

A problem is said to have **overlapping subproblems** if the problem can be broken down into subproblems which are reused several times or a recursive algorithm for the problem solves the same subproblem over and over rather than always generating new subproblems. We can represent that with function calls that may recalculate the same values at some point (it happens very often with recursive algorithms):



In above figure, we can see that many subproblems are solved again and again, for example eD(2,2) is called three times. Since same suproblems are called again, this problem has Overlapping Subprolems property.

5.1. EDIT DISTANCE

The problem statement is like if we are given two string str1 and str2 then how many minimum number of operations can be performed on the str1 that it gets converted to str2. The valid operations are:

- Insert
- Delete
- Replace

and they all have equal cost.

a. Recursive solution

```
def editDistance(str1, str2, m , n):
```

```
# If first string is empty, the only option is to
# insert all characters of second string into first
if m==0:
return n

# If second string is empty, the only option is to
# remove all characters of first string
if n==0:
return m

# If last characters of two strings are same, nothing
# much to do. Ignore last characters and get count for
# remaining strings.
if str1[m-1]==str2[n-1]:
return editDistance(str1,str2,m-1,n-1)

# If last characters are not same, consider all three
# operations on last character of first string, recursively
# compute minimum cost for all three operations and take
# minimum of three values.
return 1 + min(editDistance(str1, str2, m, n-1),    # Insert
editDistance(str1, str2, m-1, n),    # Remove
editDistance(str1, str2, m-1, n-1)    # Replace
)
```

The time complexity of above solution is exponential. In worst case, we may end up doing $O(3^m)$ operations. The worst case happens when none of characters of two strings match.

Because there are many subproblems solved again and again, the overlapping subproblem property is valid and we can solve the edit distance problem by using dynamic programming.

b. Dynamic programming solution

```
# A Dynamic Programming based Python program for edit
```

```
# distance problem
def editDistDP(str1, str2, m, n):
# Create a table to store results of subproblems
dp = [[0 for x in range(n+1)] for x in range(m+1)]

# Fill d[][] in bottom up manner
for i in range(m+1):
for j in range(n+1):

# If first string is empty, only option is to
# insert all characters of second string
if i == 0:
dp[i][j] = j    # Min. operations = j

# If second string is empty, only option is to
# remove all characters of second string
elif j == 0:
dp[i][j] = i    # Min. operations = i

# If last characters are same, ignore last char
# and recur for remaining string
elif str1[i-1] == str2[j-1]:
dp[i][j] = dp[i-1][j-1]

# If last character are different, consider all
# possibilities and find minimum
else:
dp[i][j] = 1 + min(dp[i][j-1],        # Insert
dp[i-1][j],        # Remove
dp[i-1][j-1])    # Replace

return dp[m][n]
```

5.2. LONGEST COMMON SUBSEQUENCE

If we are given with the two strings we have to find the longest common sub-sequence (LCS) present in both of them.

Example:

LCS for input Sequences “ABCDGH” and “AEDFHR” is “ADH” of length 3.

LCS for input Sequences “AGGTAB” and “GXTXAYB” is “GTAB” of length 4.

The naive solution for this problem is to generate all subsequences of both given sequences and find the longest matching subsequence. This solution is exponential in term of time complexity.

Python code:

```
def lcs(X , Y):
```

```
# find the length of the strings
m = len(X)
n = len(Y)

# declaring the array for storing the dp values
L = [[None]*(n+1) for i in xrange(m+1)]

"""Following steps build L[m+1][n+1] in bottom up fashion
Note: L[i][j] contains length of LCS of X[0..i-1]
and Y[0..j-1]"""
for i in range(m+1):
for j in range(n+1):
if i == 0 or j == 0 :
L[i][j] = 0
elif 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])

# L[m][n] contains the length of LCS of X[0..n-1] & Y[0..m-1]
return L[m][n]
```

5.3. KNAPSACK

Given a set of items, each with a weight and a value, determine the number of each item to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible.

Suppose we have a knapsack which can hold int w = 10 weight units. We have a total of int n = 4 items to choose from, whose values are represented by an array val = {10, 40, 30, 50} and weights represented by an array wt = {5, 4, 6, 3}.

Since this is the 0–1 knapsack problem, we can either include an item in our knapsack or exclude it, but not include a fraction of it, or include it multiple times.

The value of the knapsack algorithm depends on two factors:

1. How many packages are being considered
2. The remaining weight which the knapsack can store.

Therefore, you have two variable quantities.

If K[i][j] is the maximum possible value by selecting in packages {1, 2, ..., i} with weight limit j.

- The maximum value when selected in n packages with the weight limit W is B[n][W]. In other words: When there are i packages to choose, K[i][j] is the optimal weight when the maximum weight of the knapsack is j.
- The optimal weight is always less than or equal to the maximum weight: K[i][j] ≤ j.

So, we have:

- wt[i], val[i] are in turn the weight and value of package i {1, …, n}.
- W is the maximum weight that the knapsack can carry.

The code in Python will be:

```
def knapSack(W, wt, val, n):
K = [[0 for x in range(W+1)] for x in range(n+1)]
```

```
# Build table K[][] in bottom up manner
for i in range(n+1):            # items
for w in range(W+1):          # weights
if i==0 or w==0:
K[i][w] = 0
elif wt[i-1] <= w:              # item weight fits remaining weight
# choose the best out of the two options
# either take it and increase your value but also your weight
# or don’t take it and keep your weight
K[i][w] = max(val[i-1] + K[i-1][w-wt[i-1]],  K[i-1][w])
else:                  # item doesn’t fit
K[i][w] = K[i-1][w]

return K[n][W]
```

5.4. FIBONACCI NUMBERS

The Fibonacci numbers are the numbers in the following integer sequence.

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ……..

the sequence Fn of Fibonacci numbers is defined by the recurrence relation:

$F_n = F_{n-1} + F_{n-2}$

so, every number is created by adding the 2 previous. The initial values are:

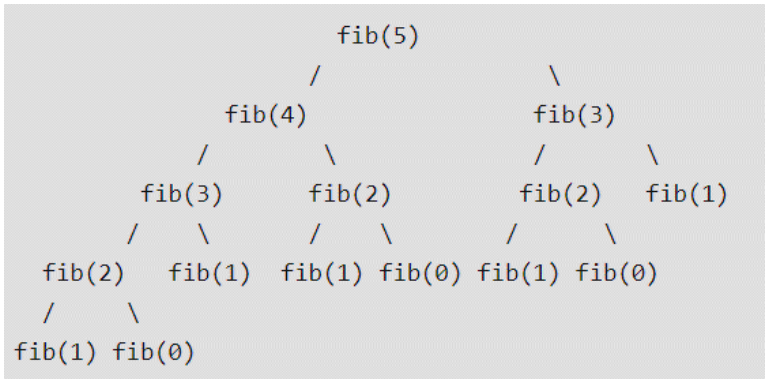
$F_0 = 0$ and $F_1 = 1$

Obviously, we can build recursively each fibonacci number.

The recursive code in Python is:

```
def Fibonacci(n):
if n<0:
return -1            # Invalid input
elif n==0:
return 0             # First Fibonacci number is 0
elif n==1:
return 1             # Second Fibonacci number is 1
else:
return Fibonacci(n-1)+Fibonacci(n-2)          # sum of the two previous
```

The recursion tree from the function calls:



We can see that some calls like Fib(0), Fib(1) etc are repeated. So, the overlapping subproblems requirement is satisfied and we can solve the Fibonacci numbers problem using dynamic programming:

```
def fibonacci(n):
FibArray = [0, 1]              # Taking 1st two fibonacci nubers as 0 and
1

while len(FibArray) < n + 1:
FibArray.append(0)

if n <= 1:
return n              # 0 or 1
else:
if FibArray[n - 1] == 0:        # if 1st factor it hasn't been examined
FibArray[n - 1] = fibonacci(n - 1)        # call and make it

if FibArray[n - 2] == 0:
FibArray[n - 2] = fibonacci(n - 2)

FibArray[n] = FibArray[n - 2] + FibArray[n - 1]  # add the 2 factors
return FibArray[n]
```

5.5. WEIGHTED JOB SCHEDULING

Weighted Job Scheduling Algorithm can also be denoted as Weighted Activity Selection Algorithm.

The problem is, given certain jobs with their start time and end time, and a profit you make when you finish the job, what is the maximum profit you can make given no two jobs can be executed in parallel?

Comparing to the previous greedy example of job scheduling, here we don't focus on adding as many jobs as possible, but our goal is to maximize our profit, by selecting the proper jobs.

Example:

Name	A	B	C	D	E	F
(Start Time, Finish Time)	(2,5)	(6,7)	(7,9)	(1,3)	(5,8)	(4,6)
Profit	6	4	2	5	11	5

The jobs are denoted with a name, their start and finishing time and profit. After a few iterations, we can find out if we perform Job-A and Job-E, we can get the maximum profit of 17. Now how to find this out using an algorithm? The first thing we do is sort the jobs by their finishing time in non-decreasing order. Why do we do this? It's because if we select a job that takes less time to finish, then we leave the most amount of time for choosing other jobs. We have:

Name	D	A	F	B	E	C
(Start Time, Finish Time)	(1,3)	(2,5)	(4,6)	(6,7)	(5,8)	(7,9)
Profit	5	6	5	4	11	2

We'll have an additional temporary array Acc_Prof of size n (Here, n denotes the total number of jobs). This will contain the maximum accumulated profit of performing the jobs. Don't get it? Wait and watch. We'll initialize the values of the array with the profit of each jobs. That means, Acc_Prof[i] will at first hold the profit of performing i-th job.

Acc_Prof	5	6	5	4	11	2

Now let's denote position 2 with i, and position 1 will be denoted with j. Our strategy will be to iterate j from 1 to i-1 and after each iteration, we will increment i by 1, until i becomes n+1.

j i

Name	D	A	F	B	E	C
(Start Time, Finish Time)	(1,3)	(2,5)	(4,6)	(6,7)	(5,8)	(7,9)
Profit	5	6	5	4	11	2
Acc_Prof	5	6	5	4	11	2

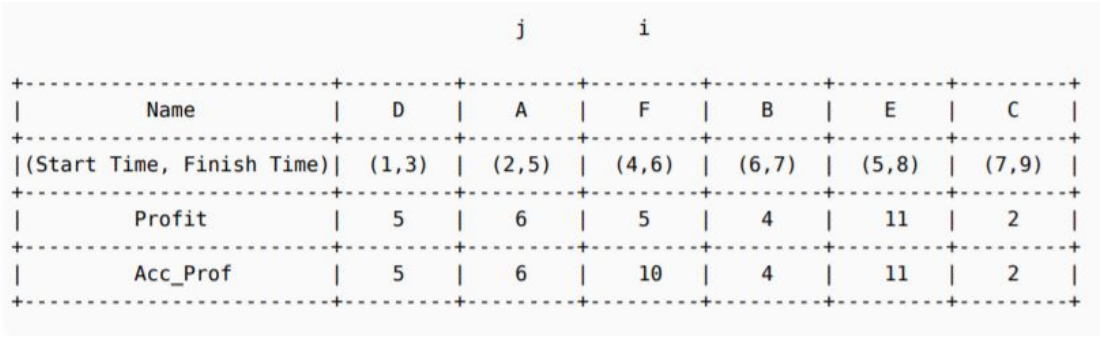
We check if Job[i] and Job[j] overlap, that is, if the finish time of Job[j] is greater than Job[i]'s start time, then these two jobs can't be done together. However, if they don't overlap, we'll check if Acc_Prof[j] + Profit[i] > Acc_Prof[i]. If this is the case, we will update Acc_Prof[i] = Acc_Prof[j] + Profit[i].

Acc_Prof[j] + Profit[i] represents the accumulated profit of doing these two jobs toegther. Let's check it for our example: Here Job[j] overlaps with Job[i]. So these to can't be done together. Since our j is equal to i-1, we increment the value of i to i+1 that is 3. And we make j = 1.

j i

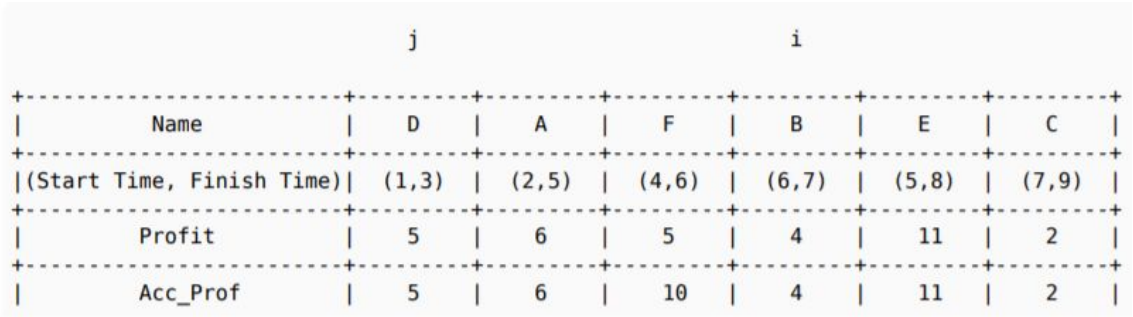
Name	D	A	F	B	E	C
(Start Time, Finish Time)	(1,3)	(2,5)	(4,6)	(6,7)	(5,8)	(7,9)
Profit	5	6	5	4	11	2
Acc_Prof	5	6	5	4	11	2

Now Job[j] and Job[i] don't overlap. The total amount of profit we can make by picking these two jobs is: Acc_Prof[j] + Profit[i] = 5 + 5 = 10 which is greater than Acc_Prof[i]. So we update Acc_Prof[i] = 10. We also increment j by 1. We get,



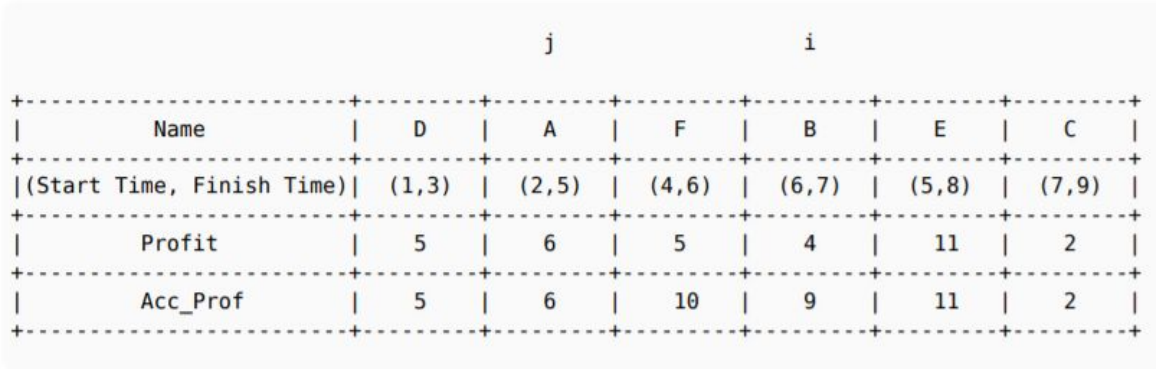
	j			i			
Name	D	A	F	B	E	C
(Start Time, Finish Time)	(1,3)	(2,5)	(4,6)	(6,7)	(5,8)	(7,9)
Profit	5	6	5	4	11	2
Acc_Prof	5	6	10	4	11	2

Here, Job[j] overlaps with Job[i] and j is also equal to i-1. So we increment i by 1, and make j = 1. We get,



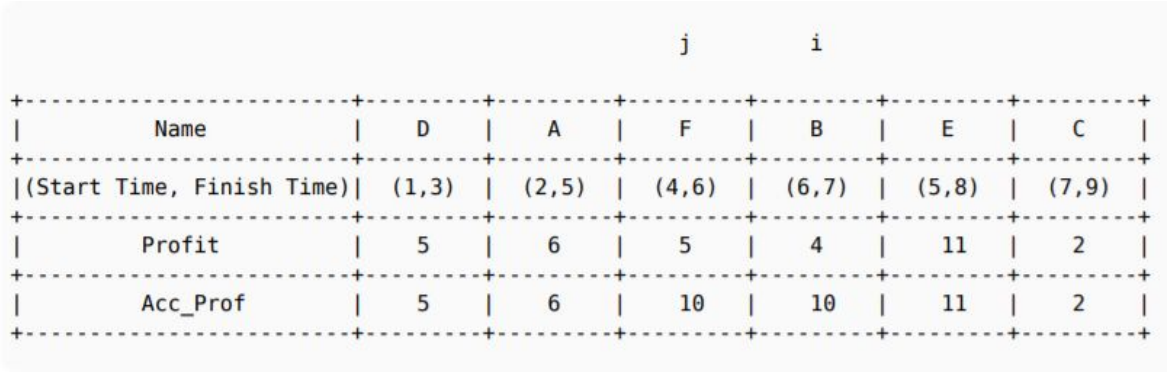
	j				i		
Name	D	A	F	B	E	C
(Start Time, Finish Time)	(1,3)	(2,5)	(4,6)	(6,7)	(5,8)	(7,9)
Profit	5	6	5	4	11	2
Acc_Prof	5	6	10	4	11	2

Now, Job[j] and Job[i] don't overlap, we get the accumulated profit 5 + 4 = 9, which is greater than Acc_Prof[i]. We update Acc_Prof[i] = 9 and increment j by 1.



			j		i		
Name	D	A	F	B	E	C
(Start Time, Finish Time)	(1,3)	(2,5)	(4,6)	(6,7)	(5,8)	(7,9)
Profit	5	6	5	4	11	2
Acc_Prof	5	6	10	9	11	2

Again Job[j] and Job[i] don't overlap. The accumulated profit is: 6 + 4 = 10, which is greater than Acc_Prof[i]. We again update Acc_Prof[i] = 10. We increment j by 1. We get:



				j	i		
Name	D	A	F	B	E	C
(Start Time, Finish Time)	(1,3)	(2,5)	(4,6)	(6,7)	(5,8)	(7,9)
Profit	5	6	5	4	11	2
Acc_Prof	5	6	10	10	11	2

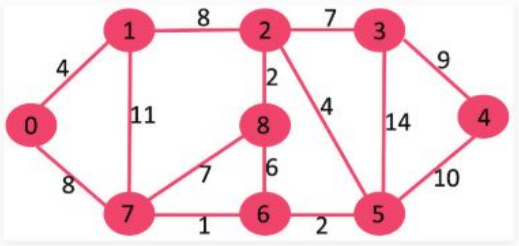
If we continue this process, after iterating through the whole table using i, our table will finally look like:

TREE AND GRAPH ALGORITHMS

6.1. DIJKSTRA

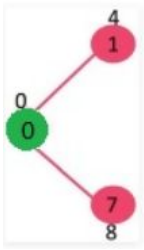
Given a graph and a source vertex in the graph, find shortest paths from source to all vertices in the given graph. In the following, the edges represent the distances between nodes.

Let’s see the following example:

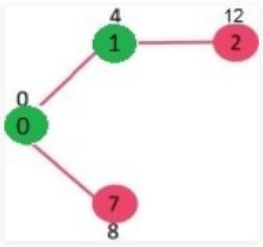


With green we will represent the nodes examined and with red the nodes that are being examined in this step.

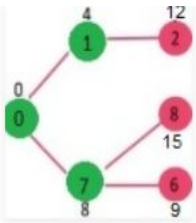
In each step, we are going to evaluate the calculated distance from the source (node 0), based on the edge weight. We are going to choose the best options in each step. This is obviously a greedy technique.



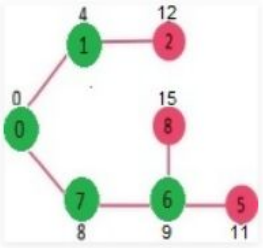
Pick the vertex with minimum distance value and not already included in our final result, in this case node 1 satisfies this condition (distance 4<8). We continue with the next level:



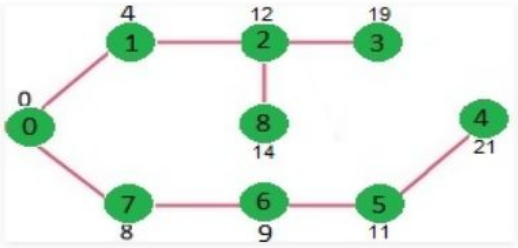
Again we are going to choose the minimum distance node, which is node 7 in total distance 8(<12).



We pick again the vertex of minimum distance, which is vertex 6 (distance 9<12):



By following the same logic, we finally get the result:



6.2. PRIM

First we need to explain what is a **minimum spanning tree**.

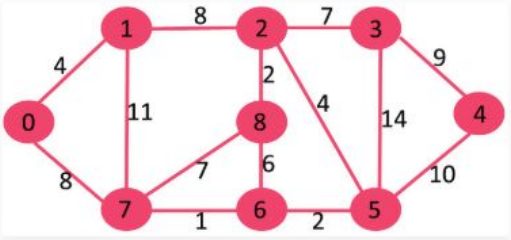
Given a connected and undirected graph, a spanning tree of that graph is a subgraph that is a tree and connects all the vertices together. A single graph can have many different spanning trees. A minimum spanning tree (MST) or minimum weight spanning tree for a weighted, connected and undirected graph is a spanning tree with weight less than or equal to the weight of every other spanning tree. The weight of a spanning tree is the sum of weights given to each edge of the spanning tree.

It starts with an empty spanning tree.The idea is to maintain two sets of vertices. The first set contains the vertices already included in the MST, the other set contains the vertices not yet included. At every step, it considers all the edges that connect the two sets, and picks the minimum weight edge from these edges. After picking the edge, it moves the other endpoint of the edge to the set containing MST.

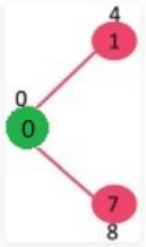
A group of edges that connects two set of vertices in a graph is called cut in graph theory.

So, at every step of Prim's algorithm, we find a cut (of two sets, one contains the vertices already included in MST and other contains rest of the vertices), pick the minimum weight edge from the cut and include this vertex to MST Set (the set that contains already included vertices).

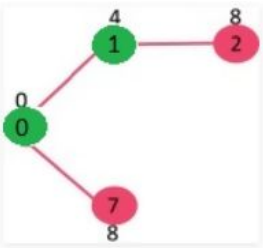
Here the numbers on the edges represent the weight of that edge. We can see the steps of the algorithm on the following graph:



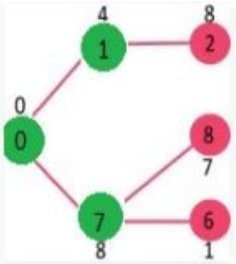
Adjacent vertices of 0 are 1 and 7. The key values of 1 and 7 are updated as 4 and 8. The vertices included in MST are shown in green color.



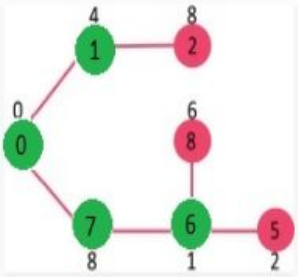
Pick the vertex with minimum key value and not already included in MST, so vertex 1 is included. Update now the key values of adjacent vertices of 1.



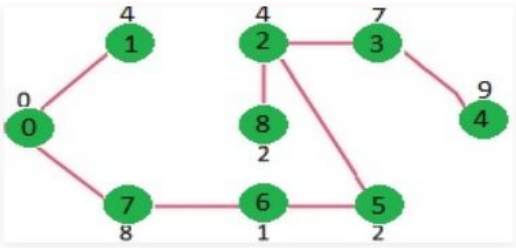
Pick the vertex with minimum key value and not already included in MST. We can either pick vertex 7 or vertex 2, as they have the same value, let vertex 7 is picked.



We keep working in the same way.



And finally we get our MST!



6.3. KRUSKAL

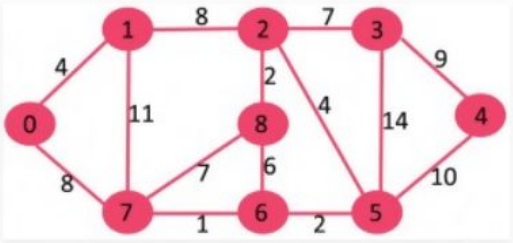
Kruskal algorithm follows the steps:

1. Sort all the edges in non-decreasing order of their weight.

2. Pick the smallest edge. Check if it forms a cycle with the spanning tree formed so far. If cycle is not formed, include this edge. Else, discard it.

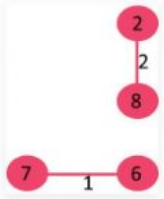
3. Repeat step#2 until there are (V-1) edges in the spanning tree.

The algorithm is a Greedy Algorithm. The Greedy Choice is to pick the smallest weight edge that does not cause a cycle in the MST constructed so far. Let us understand it with an example: Consider the below input graph.

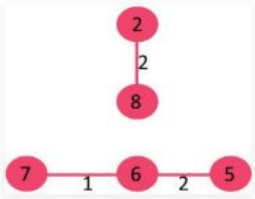


1. Pick edge 7-6, which is the lightest edge of all: No cycle is formed, include it.

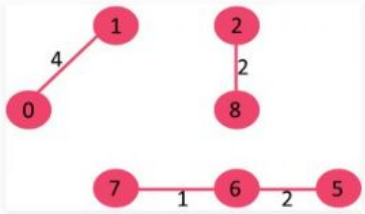
2. *Pick edge 8-2, 2nd lightest:* No cycle is formed, include it.



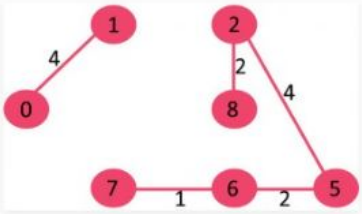
3. *Pick edge 6-5:* No cycle is formed, include it.



4. *Pick edge 0-1:* No cycle is formed, include it.

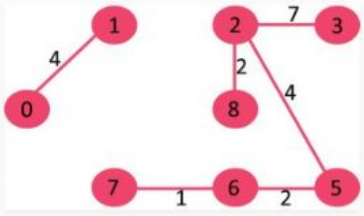


5. *Pick edge 2-5:* No cycle is formed, include it.



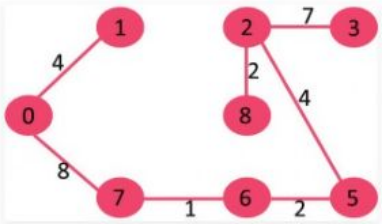
6. *Pick edge 8-6:* Since including this edge results in cycle, discard it.

7. *Pick edge 2-3:* No cycle is formed, include it.



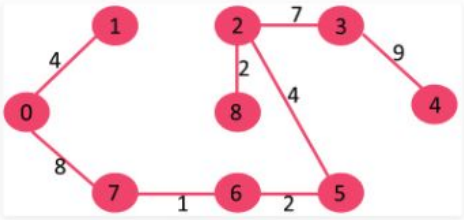
8. *Pick edge 7-8:* Since including this edge results in cycle, discard it.

9. *Pick edge 0-7:* No cycle is formed, include it.



10. *Pick edge 1-2:* Since including this edge results in cycle, discard it.

11. *Pick edge 3-4:* No cycle is formed, include it.



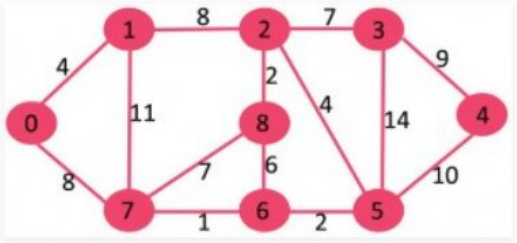
Since the number of edges included equals (V – 1), the algorithm stops here.

6.4. BORUVKA

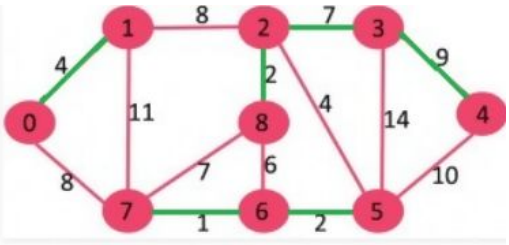
Boruvka is another greedy algorithm for MST. Below are the steps this algorithm follows:

1) Input is a connected, weighted and directed graph.
2) Initialize all vertices as individual components (or sets).
3) Initialize MST as empty.
4) While there are more than one components, do following for each component.
a) Find the closest weight edge that connects this component to any other component.
b) Add this closest edge to MST if not already added.
5) Return MST.

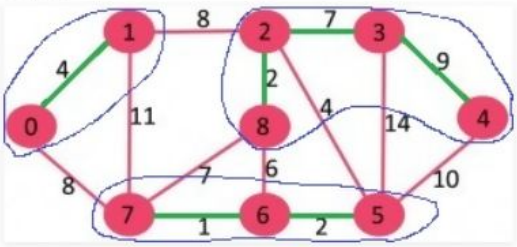
Let's consider the following graph. For every component (subtree), find the cheapest edge that connects it to some other subtree.



The cheapest edges are highlighted with green color. Now MST becomes {0-1, 2-8, 2-3, 3-4, 5-6, 6-7}.

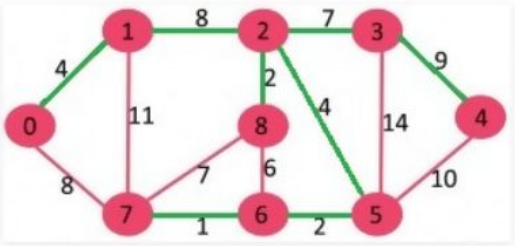


After the above step, components are {{0,1}, {2,3,4,8}, {5,6,7}}. The subtrees are encircled with blue color.



We again repeat the step, i.e., for every component (subtree), find the cheapest edge that connects it to some other component.

The cheapest edges are highlighted with green color. Now MST becomes {0-1, 2-8, 2-3, 3-4, 5-6, 6-7, 1-2, 2-5}.



At this stage, there is only one component {0, 1, 2, 3, 4, 5, 6, 7, 8} which has all edges. Since there is only one component left, we stop and return MST.

6.5. BFS

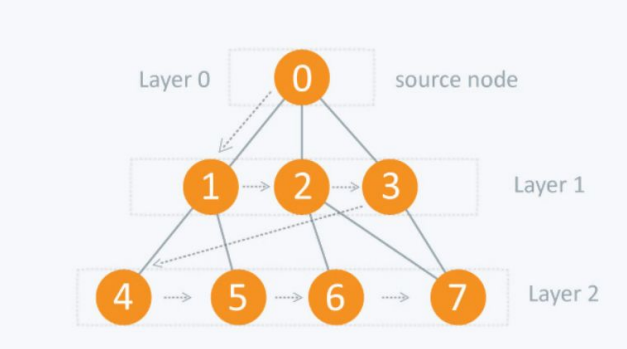
There are many ways to traverse graphs. BFS is the most commonly used approach.

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 neighbour 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

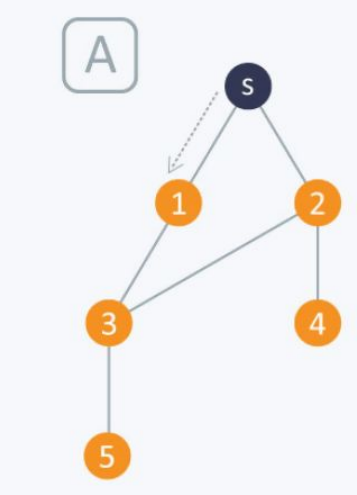


We need to be careful about cycles, as they will bring back to the previous layer if we follow them. The nodes are stored in a queue where we add them in order to visit them and from where we remove them every time we visit them.

Below we can see the steps of visiting the nodes:

Start with the root:

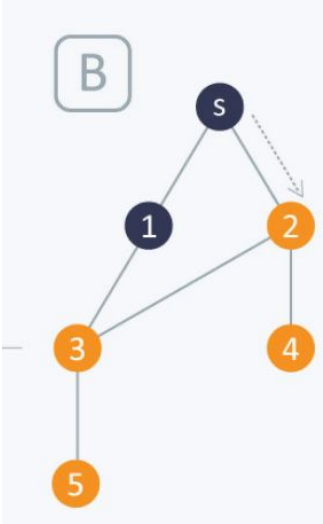
- s will be popped from the queue
- Neighbors of s i.e. 1 and 2 will be traversed
- 1 and 2, which have not been traversed earlier, are traversed. They will be:
    - Pushed in the queue
    - 1 and 2 will be marked as visited



Continue with layer 1:
1 is popped from the queue

- Neighbors of 1 i.e. s and 3 are traversed

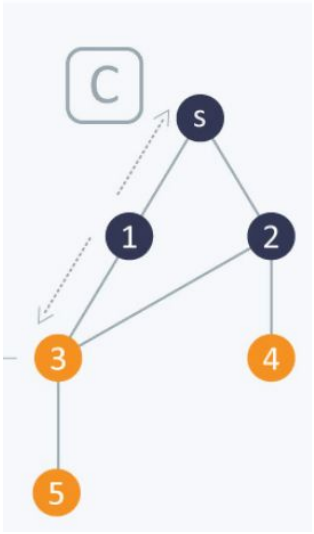
- s is ignored because it is marked as 'visited'
- 3, which has not been traversed earlier, is traversed. It is:
    - Pushed in the queue
    - Marked as visited



2 is popped from the queue

- Neighbors of 2 i.e. s, 3, and 4 are traversed
- 3 and s are ignored because they are marked as 'visited'
- 4, which has not been traversed earlier, is traversed. It is:

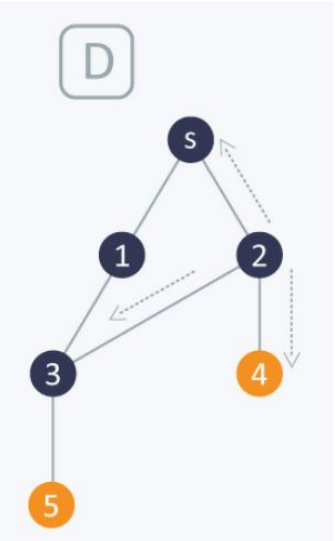
- Pushed in the queue
- Marked as visited



Let’s move to the next layer:
3 is popped from the queue

- Neighbors of 3 i.e. 1, 2, and 5 are traversed
- 1 and 2 are ignored because they are marked as 'visited'

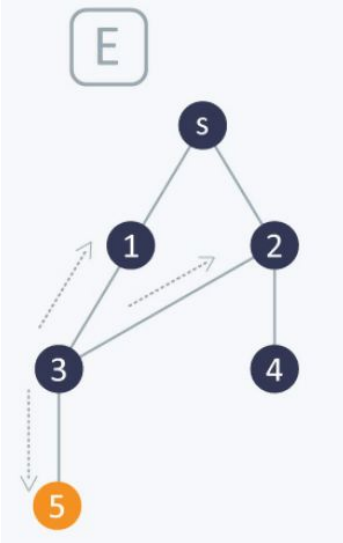
- 5, which has not been traversed earlier, is traversed. It is:
  - Pushed in the queue
  - Marked as visited



4 will be popped from the queue

- Neighbors of 4 i.e. 2 is traversed

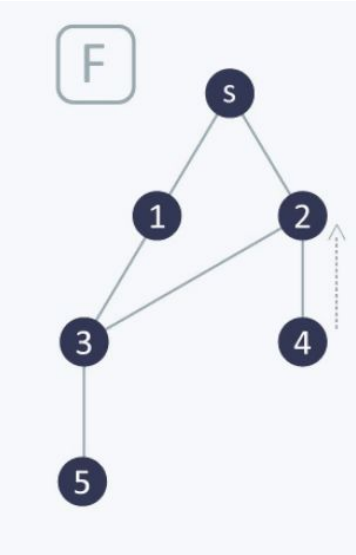
- 2 is ignored because it is already marked as 'visited'

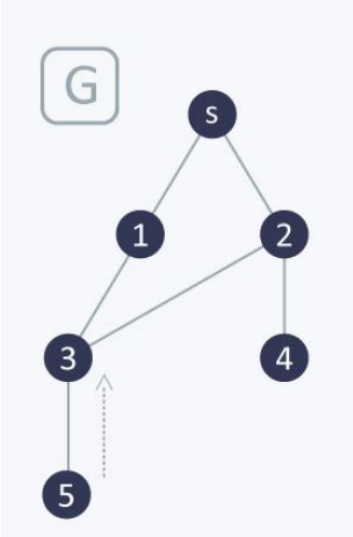


5 is popped from the queue

- Neighbors of 5 i.e. 3 is traversed
- 3 is ignored because it is already marked as 'visited'

Now we check that we have visited all the edges in the previous layer.




G
s
1
2
3
4
5

6.6. 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.

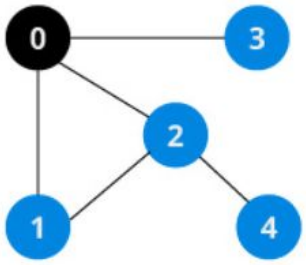
This recursive nature of DFS can be implemented using stacks. The basic idea is as follows:

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

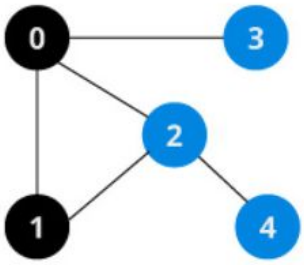
Pop a node from stack to select the next node to visit and push all its adjacent nodes into a stack.

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.

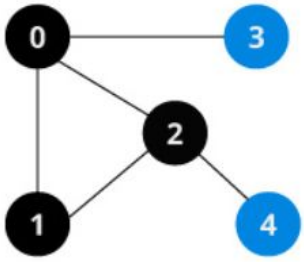
We can see the DFS steps below. With black we denote the visited part in the current step, with blue the unvisited. First, we visit the root:



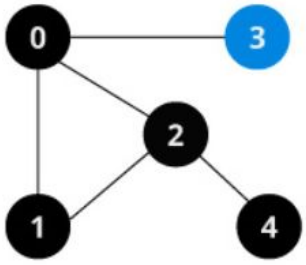
Then, we move to the first layer, at node 1:



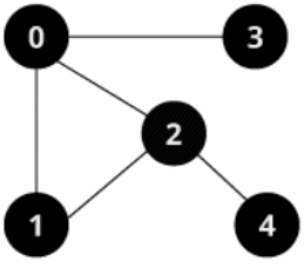
We move to node 2:



Node 4 is in the next layer:



We reached a leaf, and we backtrack to take another path depth-wise. Node 3 remains, so we visit is as well:



We visited all the nodes in the following sequence: {0,1,2,4,3}

6.7. FLOYD WARSALL

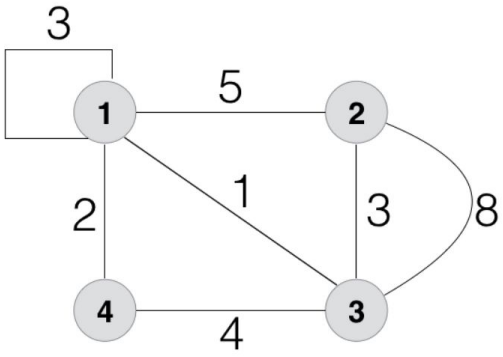
The problem is to find shortest distances between every pair of vertices in a given edge weighted directed Graph.

We initialize the solution matrix same as the input graph matrix as a first step. Then we update the solution matrix by considering all vertices as an intermediate vertex. The idea is to one by one pick all vertices and updates all shortest paths which include the picked vertex as an intermediate vertex in the shortest path. When we pick vertex number k as an intermediate vertex, we already have considered vertices {0, 1, 2, .. k-1} as intermediate vertices. For every pair (i, j) of the source and destination vertices respectively, there are two possible cases.

1) k is not an intermediate vertex in shortest path from i to j. We keep the value of dist[i][j] as it is.

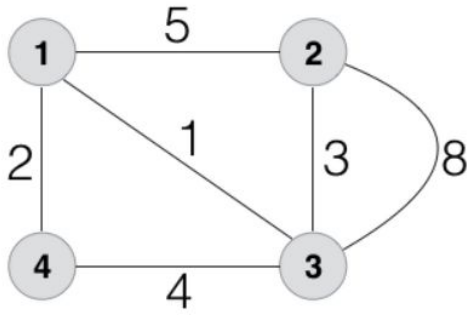
2) k is an intermediate vertex in shortest path from i to j. We update the value of dist[i][j] as dist[i][k] + dist[k][j] if dist[i][j] > dist[i][k] + dist[k][j]

Lets consider the following graph:



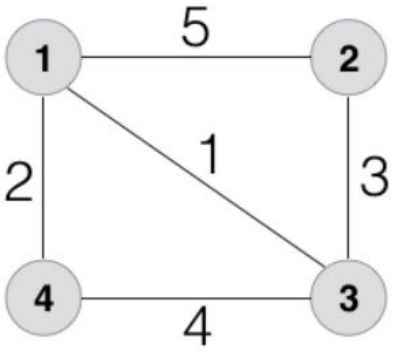
Our task is to find the all pair shortest path for the given weighted graph

Step 1: Remove all the loops.



Step 2: Remove all parallel edges between two vertices leaving only the edge with the smallest weight.

Step 3: Create a distance and sequence table.



After that:

- To find the shortest path between any two nodes we will draw two tables namely, Distance Table (D) and Sequence Table (S).
- We can also refer these tables as matrix.
- The Distance table (D) will hold distance between any two vertices.
- The Sequence table (S) will hold the name of the vertices that will help in finding the shortest path between any two vertices.
- If a graph has k vertices then our table D and S will have k rows and k columns.
- We will use the iterative method to solve the problem.

Notations we will be using

k = Iteration number
Dk= Distance table in kth iteration
Sk= Sequence table in kth iteration
dij= The distance between vertex i and j

There are 4 vertices in the graph so, our tables (Distance and Sequence) will have 4 rows and 4 columns.

Distance table D

This table holds the weight of the respective edges connecting vertices of the graph. So, if there in an edge u --> v connecting vertex u to vertex v and having weight w then we will fill the distance table D[u][v] = w. If there is no edge connecting any two vertex u to v then in that case we will fill D[u][v] = INFINITY.

D_0	1	2	3	4
1				
2				
3				
4				

From the graph above we will get the following distance table.

D_0	1	2	3	4
1	-	5	1	2
2	5	-	3	INF
3	1	3	-	4
4	2	INF	4	-

Sequence table S

This table holds the vertex that will be used to find the shortest path to reach from vertex u to vertex v.

S_0	1	2	3	4
1				
2				
3				
4				

From the graph above we will get the following sequence table.

S_0	1	2	3	4
1	-	2	3	4
2	1	-	3	4
3	1	2	-	4
4	1	2	3	-

Solution

After completing the 4 iterations we will get the following distance array.

D_4	1	2	3	4
1	-	4	1	2
2	4	-	3	6
3	1	3	-	3
4	2	6	3	-

And the following sequence table

S_4	1	2	3	4
1	-	3	3	4
2	3	-	3	3
3	1	2	-	1
4	1	3	1	-

And the required shortest paths.

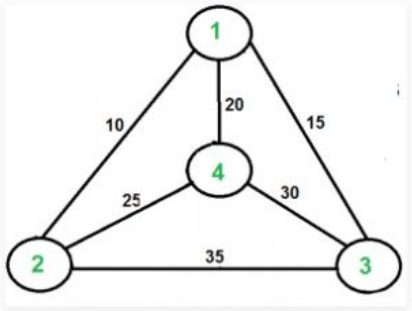
Source Vertex (i)	Destination Vertex (j)	Distance	Shortest Path
1	2	4	1 --> 3 --> 2
1	3	1	1 --> 3
1	4	2	1 --> 4
2	3	3	2 --> 3
2	4	6	2 --> 3 --> 1 --> 4
3	4	3	3 --> 1 --> 4

6.8. TSP

he travelling salesman problem (also called the travelling salesperson problem[1] or TSP) asks the following question: "Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city and returns to the origin city?"

TSP can be modelled as an undirected weighted graph, such that cities are the graph's vertices, paths are the graph's edges, and a path's distance is the edge's weight. It is a minimization problem starting and finishing at a specified vertex after having visited each other vertex exactly once. Often, the model is a complete graph (i.e. each pair of vertices is connected by an edge). If no path exists between two cities, adding an arbitrarily long edge will complete the graph without affecting the optimal tour.

For example, consider the graph shown in figure below. A TSP tour in the graph is 1-2-4-3-1. The cost of the tour is 10+25+30+15 which is 80.



Naive Solution:

1)Consider city 1 as the starting and ending point.

2)Generate all (n-1)! Permutations of cities.

3)Calculate cost of every permutation and keep track of minimum cost permutation.

4) Return the permutation with minimum cost.

Dynamic Programming solution:

Let the given set of vertices be {1, 2, 3, 4,….n}. Let us consider 1 as starting and ending point of output. For every other vertex i (other than 1), we find the minimum cost path with 1 as the starting point, i as the ending point and all vertices appearing exactly once. Let the cost of this path be cost(i), the cost of corresponding Cycle would be cost(i) + dist(i, 1) where dist(i, 1) is the distance from i to 1. Finally, we return the minimum of all [cost(i) + dist(i, 1)] values. This looks simple so far. Now the question is how to get cost(i)?

To calculate cost(i) using Dynamic Programming, we need to have some recursive relation in terms of sub-problems. Let us define a term *C(S, i) be the cost of the minimum cost path visiting each vertex in set S exactly once, starting at 1 and ending at i*.

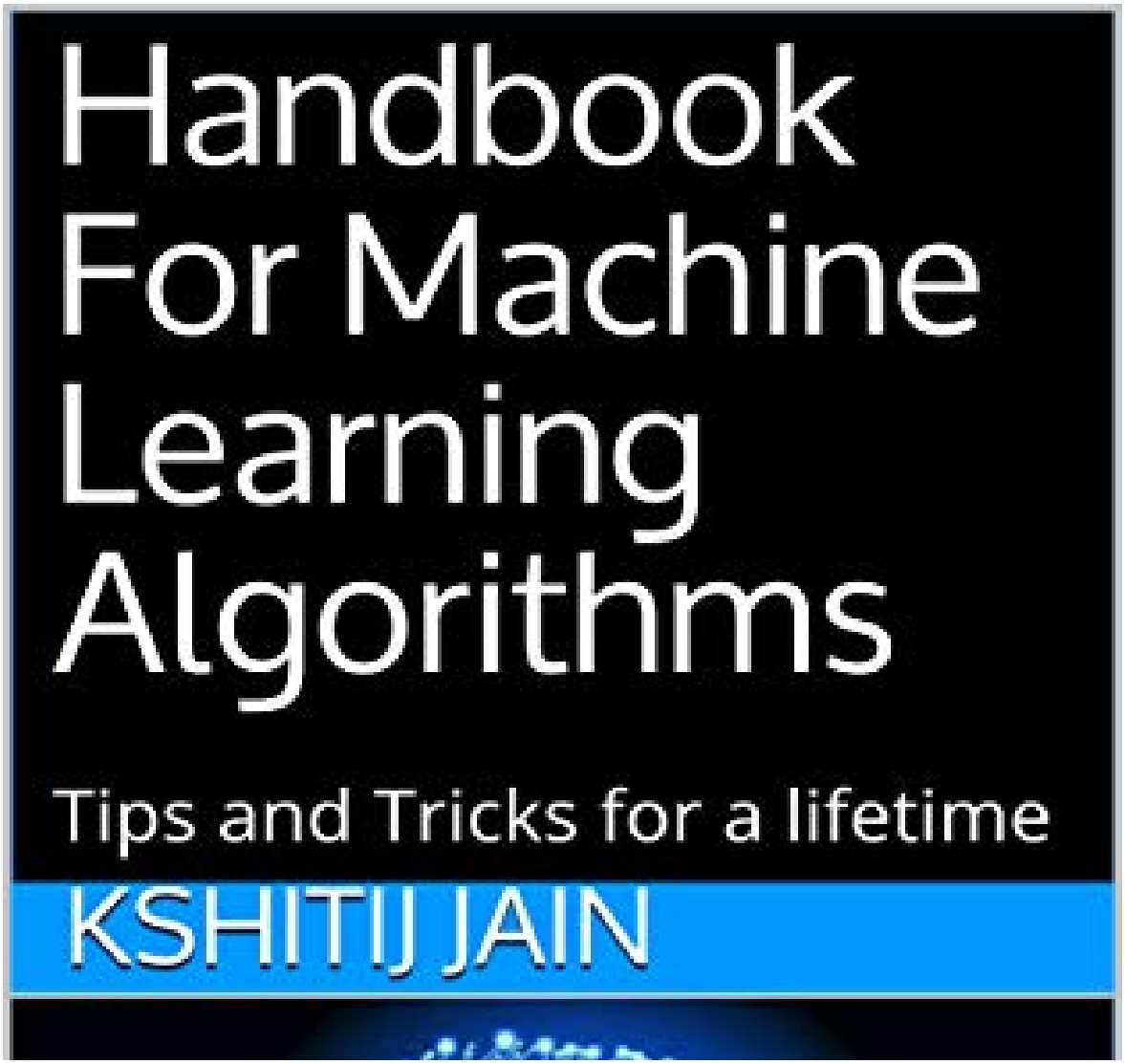
We start with all subsets of size 2 and calculate C(S, i) for all subsets where S is the subset, then we calculate C(S, i) for all subsets S of size 3 and so on. Note that 1 must be present in every subset.

If size of S is 2, then S must be {1, i},
C(S, i) = dist(1, i)
Else if size of S is greater than 2.
C(S, i) = min { C(S-{i}, j) + dis(j, i)} where j belongs to S, j != i and j != 1.

For a set of size n, we consider n-2 subsets each of size n-1 such that all subsets don’t have nth in them.

* * *

CHECK OUT OTHER AMAZING BOOKS BY AUTHOR:




Codes to get you started
with python
PYTHON CODING
KSHITIJ JAIN

again
continue;
} else if (sInput
again = true;
continue;
} while (++iN
if (isdigit(
continue;
if (iN
CODE BANK

* * *

For Amazing Offers on New Books to be released and FREE Goodies visit:

https://kshitijjainbuissne.wixsite.com/mysite

* * *

Thanks For Reading.