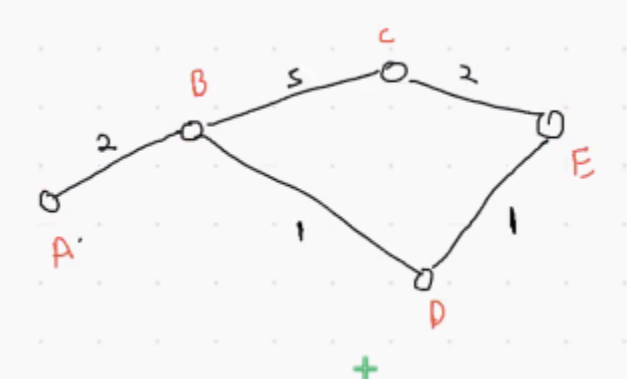
GRAPHS – 3

Single Source Shortest Path:

Suppose we have an undirected graph,



So, what is the min distance to reach E from A.

We have 2 paths;

A B C E = AB + BC + CE = 2 + 5 + 2 = 9

A B D E = AB + BD + DE = 2 + 1 + 1 = 4 ➔ this is the shortest path.

The question will be what is the min cost to travel from one point to another point. What is the min cost to travel from **source** to **destination**?

To solve this problem, Dijkstra, a Dutch guy came up with a solution.

In Google Search, pages are linked with another page or multiple pages. Google will try to find the max number of references in our search.

Dijkstra Algorithm:

In the first step we make the graph. The algorithm says, “assign all the vertices as infinity”. Distance means, distance from a particular node to the source vertex. So, the distance of C means, the distance of C node from A. As I target is E, distance of E will tell us the shortest distance from source to E.

A	B	C	D	E
INF	INF	INF	INF	INF

In Dijkstra, we create a distance array in which we assign everything to infinity because we do not know the min distance so we assign the max value. Each element in the array speaks about the distance from source to the point in the array.

The distance from source to source is ZERO. So,

A	B	C	D	E
[0	INF	INF	INF	INF]

Now, the second step in the algorithm is "**RELAXATION**".

Relaxation:

The neighbor of A is B. the current distance from A to B is infinity. The new distance will be from source to destination, so source A to A is zero and the edge value (2). Hence, distance from source to B = 0 + 2 = 2. Taking the min value between (infinity, 2) as the distance.

A	B	C	D	E
[0	2	INF	INF	INF]

Visited array = [A]

We will also have a visited array, and will mark A as visited in the visited array.

Now, for B, the neighbors are C, D & A. As A is already visited, we will not consider A now and go to C & D. At B, the distance of B = min distance from source to B is 2. So, 2 + 5 = 7 can be a solution for C. Among infinity & 7, 7 is min, so the distance at C would be 7.

A	B	C	D	E
[0	2	7	INF	INF]

Another neighbor of B, the distance of D is infinity and if from B to D 2 + 1 = 3. Since 3 < infinity, the distance of D would be 3.

A	B	C	D	E
[0	2	7	3	INF]

Visited array = [A, B]

Now among 3 & 7, the min cost is 3. So taking D, the neighbor of D is E. The distance of E is infinity and if we come via D to E by traveling the edge 1, the distance from D to E would be 3 + 1 = 4. Since 4 < infinity,

A	B	C	D	E

[0 2 7 3 4]

Visited array = [A, B, D]

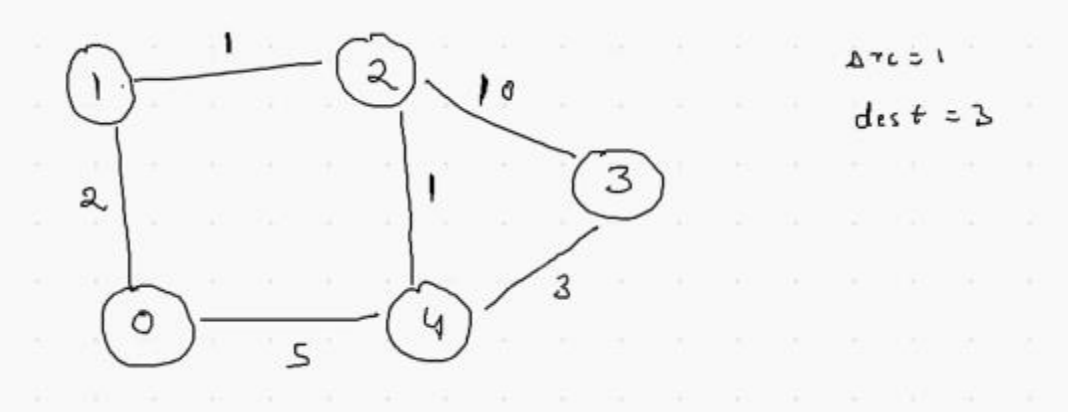
At E, the distance from C to E would be distance of C from source + edge value = 7 + 2 = 9. Since 4 < 9, the distance at E is 4. So,

A B C D E

[0 2 7 3 4]

Visited array = [A, B, D, E]

Q)



Source = 1

Dist = 3

0 1 2 3 4

[Inf inf inf inf inf]

Visited aray = []

The distance from (1) to (1) is zero.

0 1 2 3 4

[Inf 0 inf inf inf]

[F T F F F]

Relaxation:

In this algorithm, node (vertex) is pushed into a min heap and the distance from (1) to (1) = 0. Now, (1, 0) is the only entry which will be popped.

The neighbors of 1 are 0 and 2 will be relaxed. The distance of (2) is infinity and if we come from (1) to (2), the 0 + 1 = 1. Since min value is 1, so the inf value is replaced by 1 and the node value (2) along with its weight 1 is pushed into the min heap.

Now if we do the same for 0, the distance at 0 is infinity and the distance via 1 is 0 + 2 = 2. Since 2 is min, 2 is assigned as weight at node (0). So, the vertex 0 with its weight 2 is pushed into the heap. So, in the min heap now, we have

(2, 1) & (0, 2)

Which of these two has least distance? (2, 1) having the least distance will be popped from it. So now our vertex from (1, 0) is changed to (2, 1).

Since (2, 1) is the least, now we will check the neighbors of (2) which are (1), (3) and (4). Since (1) is already visited, lets check for (3).

The distance at (3) is infinity and via (2), it would be 1 + 10 = 11. 11 being min would be pushed into the heap along with (3, 11).

For (4), the distance till (2) is 1 and the min distance from (4) 1 +1 = 2. 2 being min, we will push, (4, 2) into the heap. In the heap,

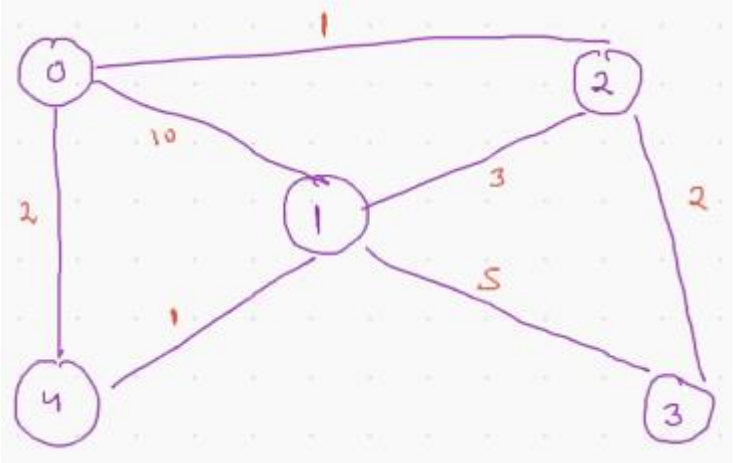
(4, 2) (0, 2) (3, 11)

(4, 2) is popped out so the vertex is 4 and distance is 2. The unvisited neighbors of 4 is 3. The current distance at 3 is 11 and if we come via (4) the distance would be 5. Since 5 < 11, (3, 5) is pushed into the heap. In the heap

(0, 2) (3, 11) (3, 5)

Source = 0

Destination = 3



The shortest path is

0 – 2 – 3 and the cost is 1 + 2 = 3.

0	1	2	3	4
[INF	INF	INF	INF	INF]
[F	F	F	F	F]

Since (0) is source, the distance array at (0) would be 0. So (0, 0) is pushed into the min heap. We will pop (0, 0) from the min heap and hence the first vertex would be (0, 0). We will also create a visited array,

0	1	2	3	4
[0	INF	INF	INF	INF]
[**T**	F	F	F	F]

The neighbors of (0) vertex are (1), (2) & (4). In relaxation of (1), the distance to (1) is infinity and if we come via (0), it would be 10. Since 10 is min, the value in the distance array is changed to 10. So, (1, 10) is pushed into the min heap.

The distance of 2 is infinity, and distance via (0) it would be 1. So (2, 1) is pushed into the min heap.

The distance of 4 is infinity, and distance via (0) it would be 2. So (4, 2) is pushed into the min heap. In the min heap, we have

(1, 10) (2, 1) (4, 2)

The min distance vertex (2, 1) is popped and so our vertex is 2 and distance is 1.

0	1	2	3	4
[0	10	INF	INF	INF]
[T	F	**T**	F	F]

The neighbors of 2 are (1) & (3). Lets relax (1). The distance of (1) is 10. From (2) the distance would be 1 + 3 = 4. Since 4 is min, we will relax it to 4. The distance of 3 is infinity, and distance via (0) it would be 1 + 2 = 3. Since 3 is min, we will relax the distance as 3 and will push (3, 3) into the min heap. In the min heap

(1, 10) (1, 4) (4,2) (3,3)

Since (4, 2) has min distance, we will continue from 4.

0	**1**	**2**	**3**	**4**
[0	10	INF	INF	INF]
[T	F	T	F	**T**]

The neighbors of 4 are (0) and (1). Since (0) is already visited, we will relax (1). The distance of (1) is 4 and via (4) it would be 3. So we will push (1, 3) into the heap.

(1, 10) (1, 4) (4,2) (3,3) (1, 3)

(4, 2) is the min value, but (4) is already visited and we are left with

(1, 10) (1, 4) (3,3) (1, 3)

The neighbors of (1) which is not visited is (3). The distance at (3) is 3 and if we come via (5), it would be 8. Since 3 < 8, the distance at (3) is 3. Since all the neighbors are visited the min heap would get empty and will get the value of min distance.

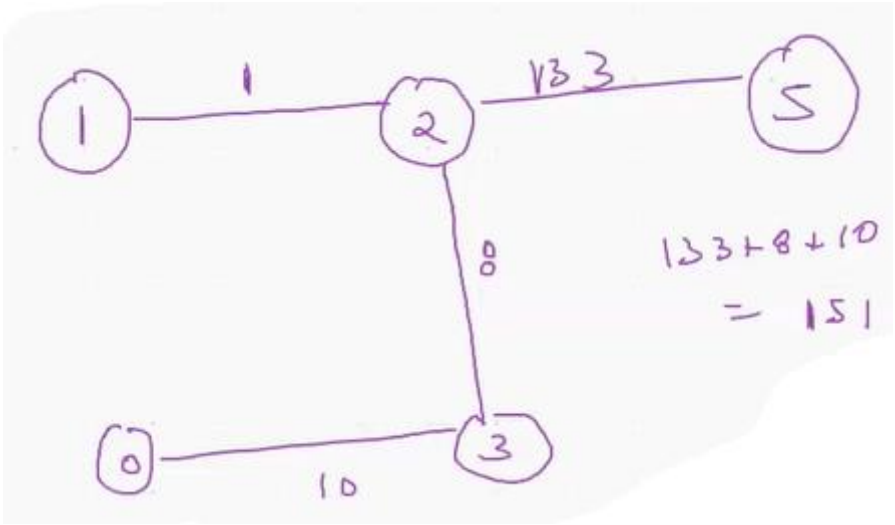
NOTE: this algorithm is called Greedy.

CODE:

```
import heapq
graph = dict()


def singleSourceShortestPath(src, dest):
    n = 1005
    dist = [float("inf")] * n
    dist[src] = 0

    visited = [False] * n

    heap = list()

    heapq.heappush(heap, (0, src))

    while len(heap) > 0:
        d, v = heapq.heappop(heap)
        visited[v] = True

        # relaxation
        for neighbor, wt in graph[v]:
            if not visited[neighbor] and dist[neighbor] < d + wt:
```

```
                dist[neighbor] = d + wt
                heapq.heappush(heap, (dist[neighbor, neighbor]))

    return dist[dest]

def addEdge(u, v, weight, directed):
    if u not in graph:
        graph[u] = list()

    graph[u].append((v, weight))

    if not directed:
        if v not in graph:
            graph[v] = list()
        graph[v].append((u, weight))

if __name__ == '__main__':
    addEdge(1, 2, 1, False)
    addEdge(0, 3, 10, False)
    addEdge(2, 3, 8, False)
    addEdge(3, 2, 11, False)
    addEdge(2, 5, 133, False)

    print(singleSourceShortestPath(0, 5))
```

OUTPUT: 151



Time Complexity: ***V log E***

Insertion Sort:

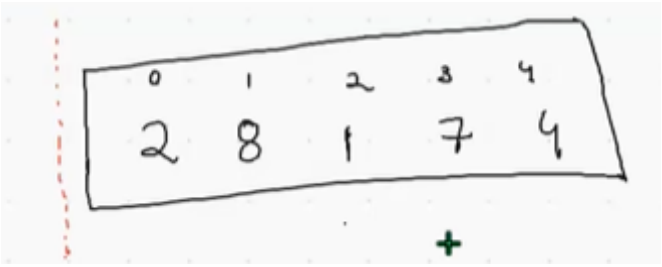
In,

1. Bubble Sort → O (n^2)
2. Selection Sort → O (n^2)
3. Merge Sort → O (nlogn)
4. Quick Sort → on an average time complexity is O(nlogn), but worst time complexity is O (n^2)

In insertion sort, as the name says it means to insert something.

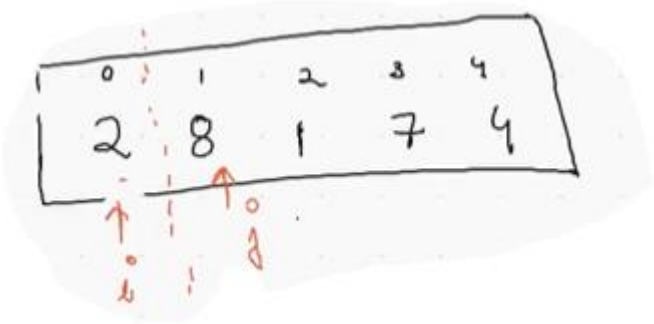
In an array [2, 8, 1, 7, 4], we pick on element and we try to put it in its correct position. There are two concepts:

1. Wall / Partition
2. Holes



The dotted red line is a wall and all the elements on the left side of the wall will be in sorted order and all the elements on the right side would be in unsorted order.

Let's say we are at j = 2 and in the sorted array we have a pointer 'i'. Can we directly put 2 to the left of the wall, as 2 in itself is directly sorted, the wall is shifted towards right and the j will move forward and our i value will be 2. (i = 2, j = 8)

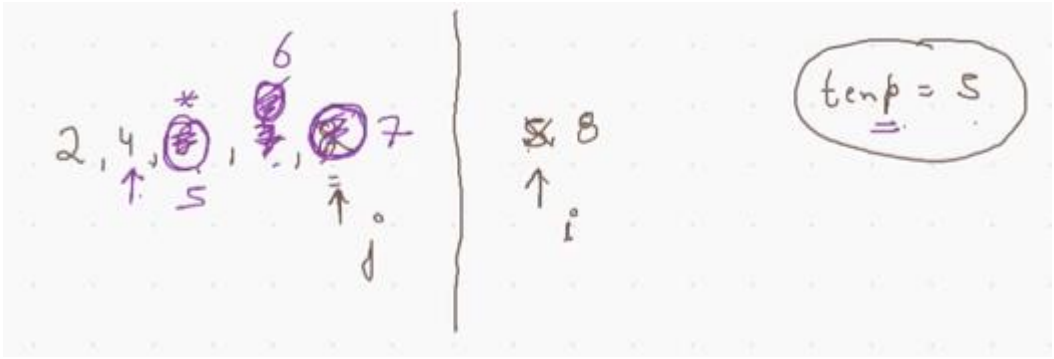


To insert 8 in the correct position in the sorted array, we will compare the j = 8 with i = 2. If we have more than one element, we have to search in the sorted array the correct position then we will hold 8 in a variable (k) and we will compare the value with every value in the sorted array. We will check for the 'i' which is less than k.

Now, we will check for the greatest value in the sorted array, if j > k, the i value is shifted to the right and there is a hole in the place of 8

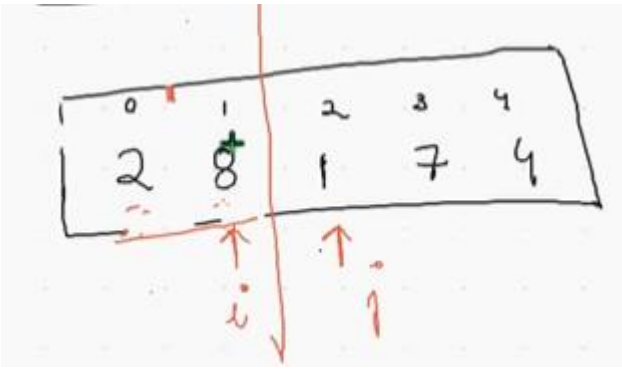
Now, the next value of I = 7 will be compared to k. Since i > k, i = 7 is shifted to the right the hole value is moved to the left. The same will happen with 6 as well.

At 4, since 4 < 5, the value of 5 is placed at hole which was previously holding 6.

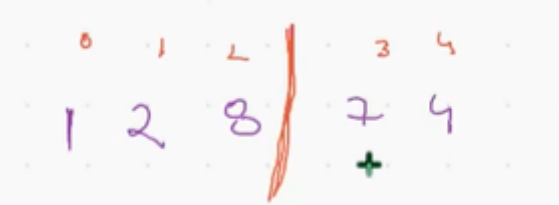


Coming back to the problem,

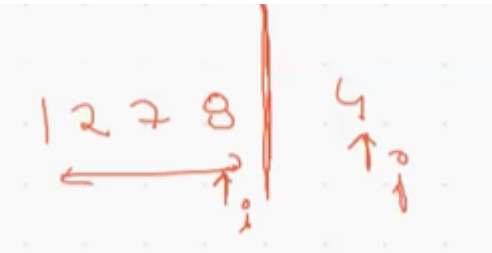
I = 2 and j = 8. J > I , so we can update the wall by moving it towards the left and the value of I = 8 and j = 1.



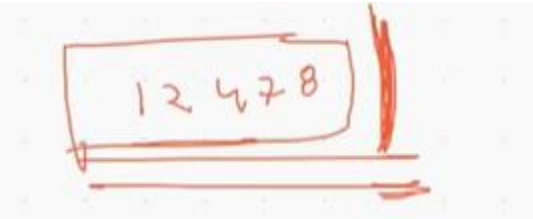
I = 8, j = 1, j < i. So, we will now check for the next I value, I = 2. Since I > j, the hole is at 2. 2 & 8 are moved towards right and the value of I = 8 and j =7.



I = 8, j = 7. Since j < I, the value of I is moved towards right and 7 is copied into the temp variable the hole is placed in 8. Now I = 2 and j = 7, since j > I, so the value of 7 is placed in the hole. So the wall after moving towards right, I = 8, j = 4



Temp = 4 and it would be placed between 2 & 7 because 2 < 4 < 7.



CODE:

```
def insertionSort(A):
    n = len(A)
    for j in range(1, n):
        i = j - 1
        temp = A[j]
        while i >= 0 and A[i] > temp:
            A[i + 1] = A[i]
            i -= 1
        A[i + 1] = temp
    return A


if __name__ == '__main__':
    a = [5, 6, 7, 1, 2, 3, 4, 56, 7]
    insertionSort(a)
    print(insertionSort(a))
```

OUTPUT: [1, 2, 3, 4, 5, 6, 7, 7, 56]