Question 1

In this question, we are asked the corresponding value n for several algorithms for different runtimes. We know that the algorithm takes $f(n)$ microseconds.

First, we replace the provided runtimes with their equivalent in microseconds: 1000000, 60 * 1000000, 60 * 60 * 1000000, and 24 * 60 * 60 * 1000000.

We also know that the runtime of each of the provided algorithms is monotonically increasing. The larger the input size, the longer the algorithm will take.

Now, we want for each function to find the value n for which $f(n)$ is equal to the specified runtime. This is equivalent of finding the inverse of a function: $f^{-1}(n)$. For simple functions, this would be easy (for example the inverse of n^2 is $\sqrt{n}$)
However, the remaining functions are harder to solve.

We can use binary search to find the value n for each of the below functions. The idea is that we start with a range: 1, infinity. Here infinity means the largest n such that $f(n)$ is <= target runtime. We iteratively split the range moving lower or higher over the function line until we find the value n for which $f(n)$is very close to our target runtime.

```python
import math

MICROSECONDS_IN_SECOND = 1_00_00_00

def n_log_n(n):
    return n * math.log2(n)

def quadratic_function(n):
    return n ** 2

def cubic_function(n):
    return n ** 3

def find_inverse(target_val, function):
    low = 1.0
    high = 1.0
    while high < target_val:
        high = high * 10.0

    while low <= high:
        mid = (low + high) / 2
        val = function(mid)

        if abs(val - target_val) <= 1:
            return mid

        if val < target_val:
            low = mid
        else:
            high = mid

for f_n in [n_log_n, quadratic_function, cubic_function]:
    for target_val in [MICROSECONDS_IN_SECOND,  # one second
                       60 * MICROSECONDS_IN_SECOND,  # one minute
                       60 * 60 * MICROSECONDS_IN_SECOND,  # one hour
                       24 * 60 * 60 * MICROSECONDS_IN_SECOND]:  # one day
        res = find_inverse(target_val, f_n)
        print(f"{f_n.__name__}, {target_val} = {res}")
```

For the factorial and exponential functions it's very expensive to calculate $f(n)$ at each iteration. Instead, we generate the possible runtime for these two algorithms for problem size between 1 and 50. Then, we find the corresponding value for the problem size using linear search.

```python
import math

MICROSECONDS_IN_SECOND = 1_00_00_00

powers_of_two = [(i, 2 ** i) for i in range(1, 50)]
factorials = [(i, math.factorial(i)) for i in range(1, 50)]

def exponential_function(n):
    return 2 ** n

def factorial_function(n):
    return math.factorial(int(n))

def fin_n_from_list(target_val, list_of_values):
    for x, y in list_of_values:
        if y >= target_val:
            return x

for f_n, list_of_values in zip(
        [exponential_function, factorial_function],
        [powers_of_two, factorials]
):
    for target_val in [MICROSECONDS_IN_SECOND,  # one second
                       60 * MICROSECONDS_IN_SECOND,  # one minute
                       60 * 60 * MICROSECONDS_IN_SECOND,  # one hour
                       24 * 60 * 60 * MICROSECONDS_IN_SECOND]:  # one day
        res = fin_n_from_list(target_val, list_of_values)
        print(f"{f_n.__name__}, {target_val} = {res}")
```

Running the above code we get the following results for each algorithm and for each different runtime.

	1 second	1 minute	1 hour	1 day

$n * log_2 n$	62746.126469676638	2801417.8832414914	133378058.8644645	2755147513.215762
n^2	1000.0000000000773	7745.966692414822	59999.99999999999	293938.7691339814
n^3	100.00001455046004	391.48676576981285	1532.6188649161998	4420.837798356199
2^n	20	26	32	37
$n!$	10	12	13	14

Question 2

A. True. Let $f(n)$ be $10 * n^2 + 9$ and $g(n) = n^2$. We need to find c and k such that f(n) < c.g(n) for all n > k. For c = 11 and k = 4, we have:
$10 * n^2 + 9 < 11 n^2$

B. True. We have $f(n) = 5n^4 - 4n^2 + 3$ and $g(n) = n^4$. For c = 1 and k = 1, we have $f(n) \geq c.g(n)$

C. True. For c = 10 and k = 1, we satisfy the condition.

D. False. Let $f(n) = 3n^4 + 2n^2 + n$ and $g(n) = n^2$. This is false because we can't find values c1 and c2 for which f(n) is bounded by g(n).

E. True. For c = 1 and k = 1, the criteria $f(n) \geq c.g(n)$

Question 3

A.

B. Yes. We have a = 16, b = 4, and $log_4(16)$ = 2, and $f(n) = 16n^2$. $n^{log_b a} = n^2$. We see that $f(n) = 16n^2$ and n^2 are of the same order. The case 1 of master theorem is c < $log_b a$, then the the complexity is $T(n) = \Theta(n^2)$

C.

D.

E.

Question 4

A. We can notice that the list is nearly sorted in ascending order with the exception of elements 5 and 6. Naive bubble sort will require $O(n^2)$ passes to sort the array. For the

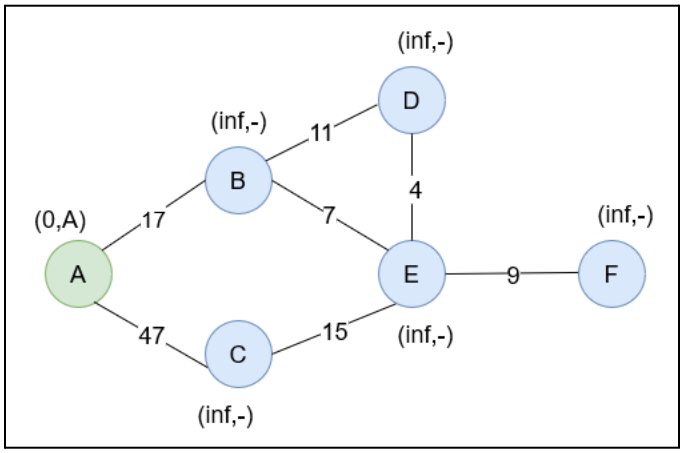
adaptive bubble sort, it will require two passes to sort the array so $O(n)$ operations. Likewise, the insertion sort will require one pass to sort the array, so $O(n)$, however, it will require having an additional array for storing the output result. Therefore, adaptive bubble sort is the best solution for this array.

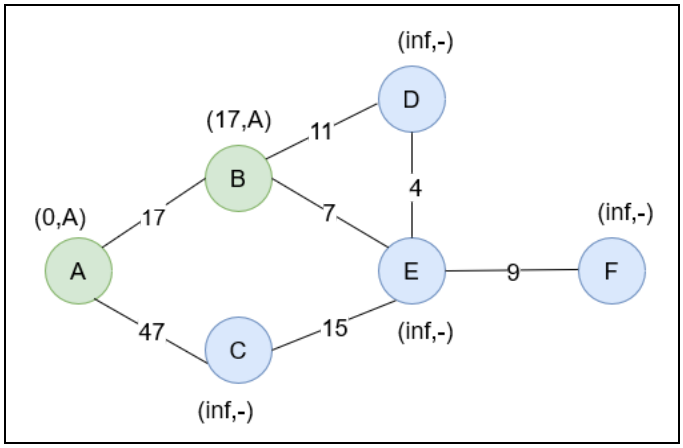
B. This list represents the worst case for sorting algorithms, where elements are sorted in descending order and we want to sort them in ascending order. The best algorithm to use in this case is merge sort because its complexity in worst case is $O(n.log(n))$. If memory complexity is of concern, quick sort might consume less memory.
C. The best algorithm for sorting this list is quick sort. If we select the middle number (3) as the pivot, the algorithm will gradually sort each sublist resulting in $O(n.log(n))$ complexity.
D. The requirement that two objects with the same values need to preserve their order after sorting make only stable sorting algorithms a valid option. Among insertion sort and merge sort, merge sort has better performance.
E. Like the previous answer, stability requires algorithms like merge sort. For handling list of tuples, we might implement a linked list where each element of the list contains two attributes: key and value.

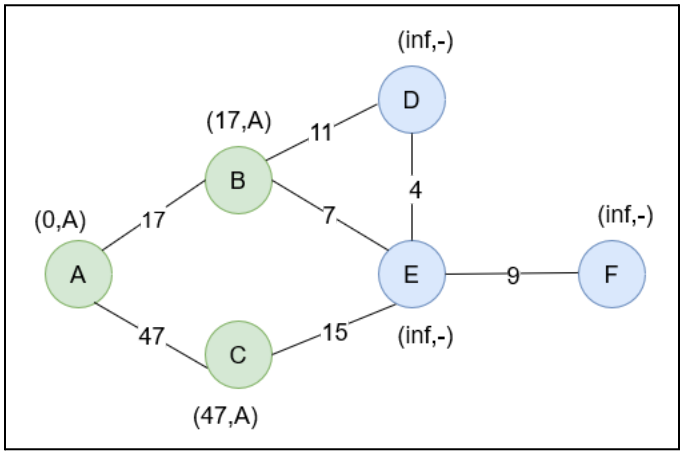
Question 5

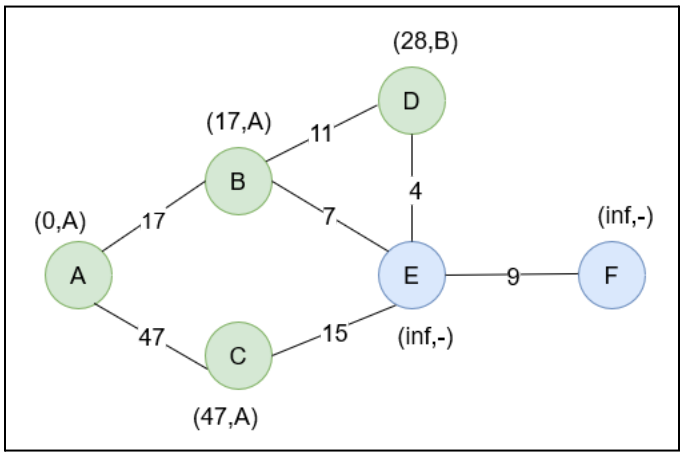
We start with a distance vector initialized with infinity (very large value) to all nodes. We assign the source node (A) distance = 0. Then, we explore the graph and update the distance vector accordingly. In addition, we maintain another vector path to tell what is the shortest path from node A to a particular node u.

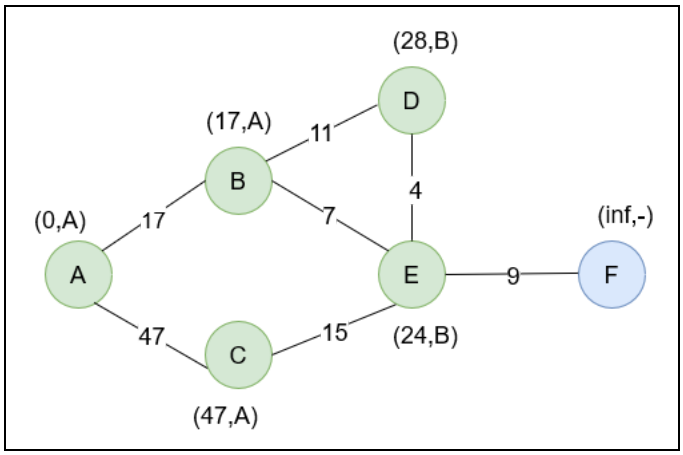
The following diagrams illustrate the working of Dijkstra's algorithm on the given graph.

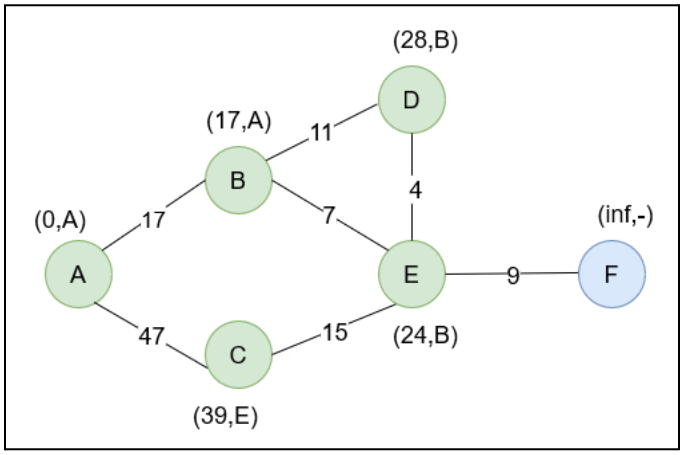


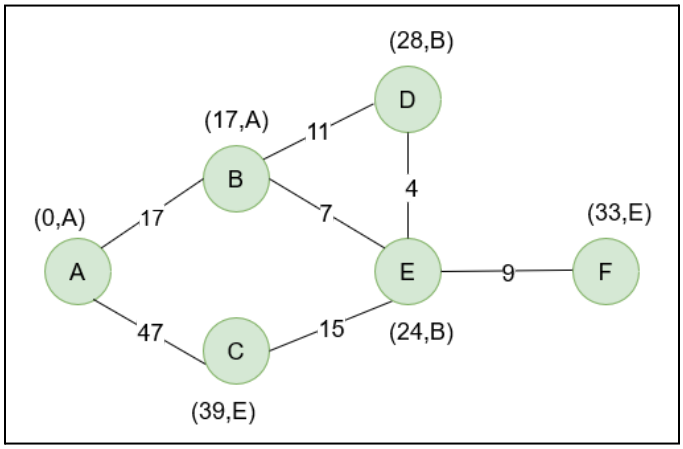

(inf,-)
D
(17,A)
11
B
4
(0,A)
17
7
(inf,-)
A
E
9
F
47
15
C
(inf,-)
(inf,-)


(inf,-)
D
(17,A)
11
B
4
(0,A)
17
7
(inf,-)
A
E
9
F
47
15
C
(inf,-)
(47,A)


(28,B)
D
(17,A)
11
B
4
(0,A)
17
7
(inf,-)
A
E
9
F
47
15
C
(inf,-)
(47,A)

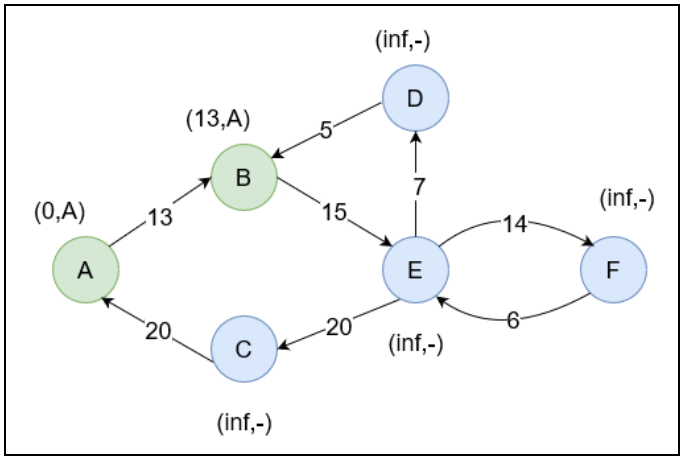


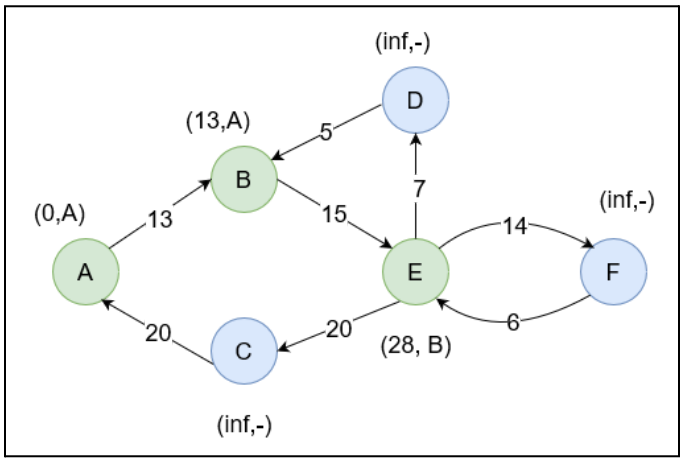


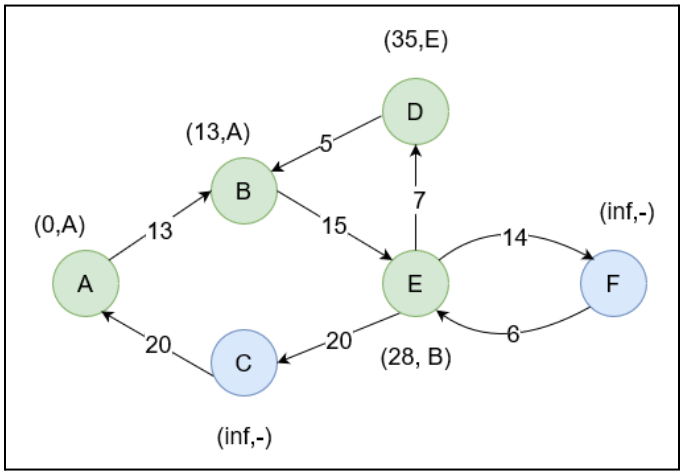


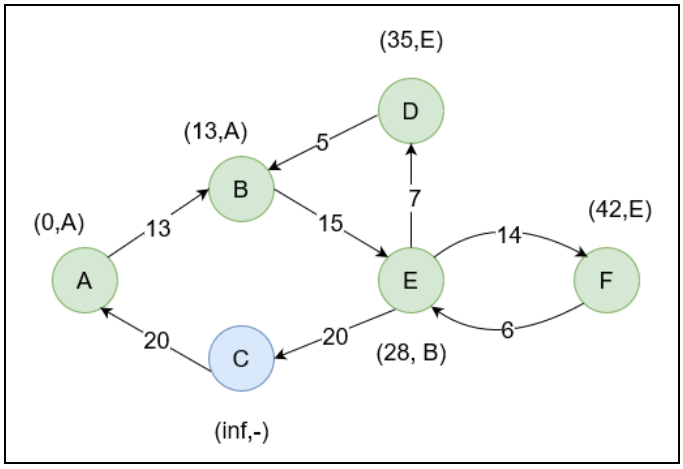
Question 6

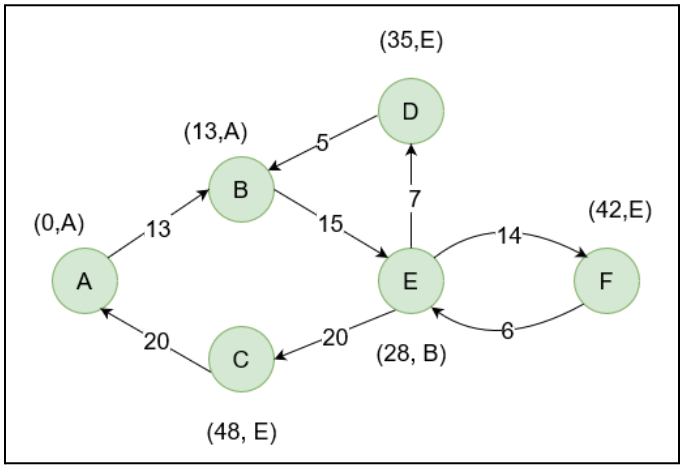
In a similar way to the previous question, we apply the Dijkstra algorithm to this directed graph.


(inf,-)
D
(13,A)
5
B
7
(0,A)
13
15
(inf,-)
14
A
E
F
20
20
6
C
(inf,-)
(inf,-)


(inf,-)
D
(13,A)
5
B
7
(0,A)
13
15
(inf,-)
14
A
E
F
20
20
6
C
(28, B)
(inf,-)


(35,E)
D
(13,A)
5
B
7
(0,A)
13
15
(inf,-)
14
A
E
F
20
20
6
C
(28, B)
(inf,-)





Question 7

A. Prim's Algorithm:
In Prim's algorithm we start from the initial node, A, and explore the graph choosing minimal edges. We leverage priority queue to pick the node with the smallest edge weight. We also mark visited nodes in order not to visit them again and from a cycle.

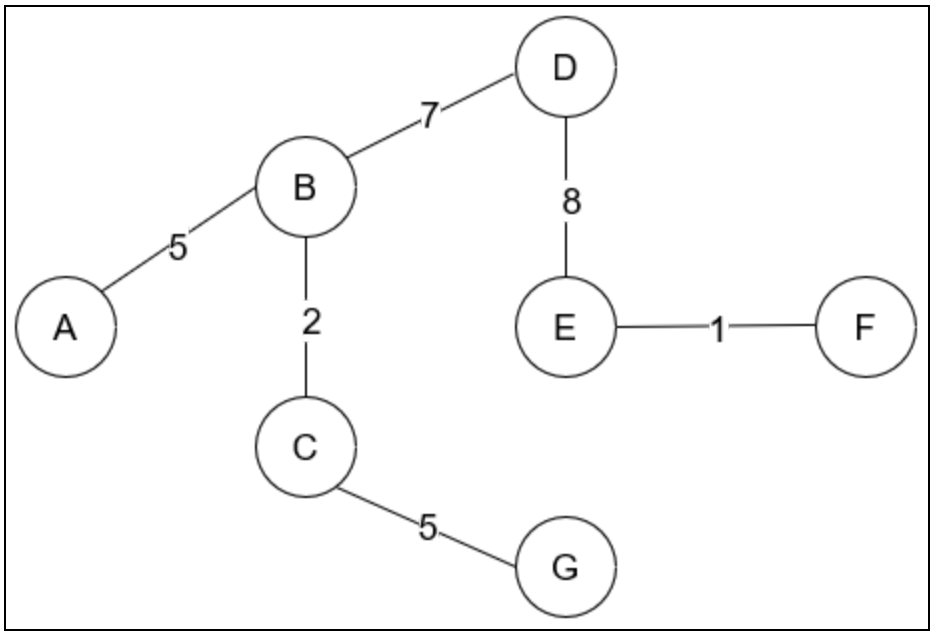
stage	V	E
0	(A)	()
1	(A, B)	((A,B))
2	(A,B,C)	((A,B), (B,C))
3	(A,B,C,G)	((A,B), (B,C), (C,G))

4	(A,B,C,G,D)	((A,B), (B,C), (C,G), (B,D))
5	(A,B,C,G,D,E)	((A,B), (B,C), (C,G), (B,D), (D,E))
6	(A,B,C,G,D,E,F)	((A,B), (B,C), (C,G), (B,D), (D,E), (E,F))

B. Kruskal's Algorithm
In Kruskal's algorithm, we start by sorting the edges by their weights in ascending order. Then, we iterate over the edges and choose edges with minimal weight as long as they don't form a cycle.

Stage	Edges	Components	E
0		((A), (B), (C), (D), (E), (F), (G))	()
1		((A), (B), (C), (D), (E,F), (G))	((E,F))
2		((A), (B,C), (D), (E,F), (G))	((E,F), (B,C))
3		((A,B,C), (D), (E,F), (G))	((E,F), (B,C), (A,B))
4		((A,B,C,G), (D), (E,F))	((E,F), (B,C), (A,B), (C,G))
5		((A,B,C,G,D), (E,F))	((E,F), (B,C), (A,B), (C,G), (B,D))
6		((A,B,C,G,D,E,F))	((E,F), (B,C), (A,B), (C,G), (B,D), (D,E))

The below graph shows the final output of the two algorithms (prim and kruskal). We get a spanning tree with a total weight of 28.

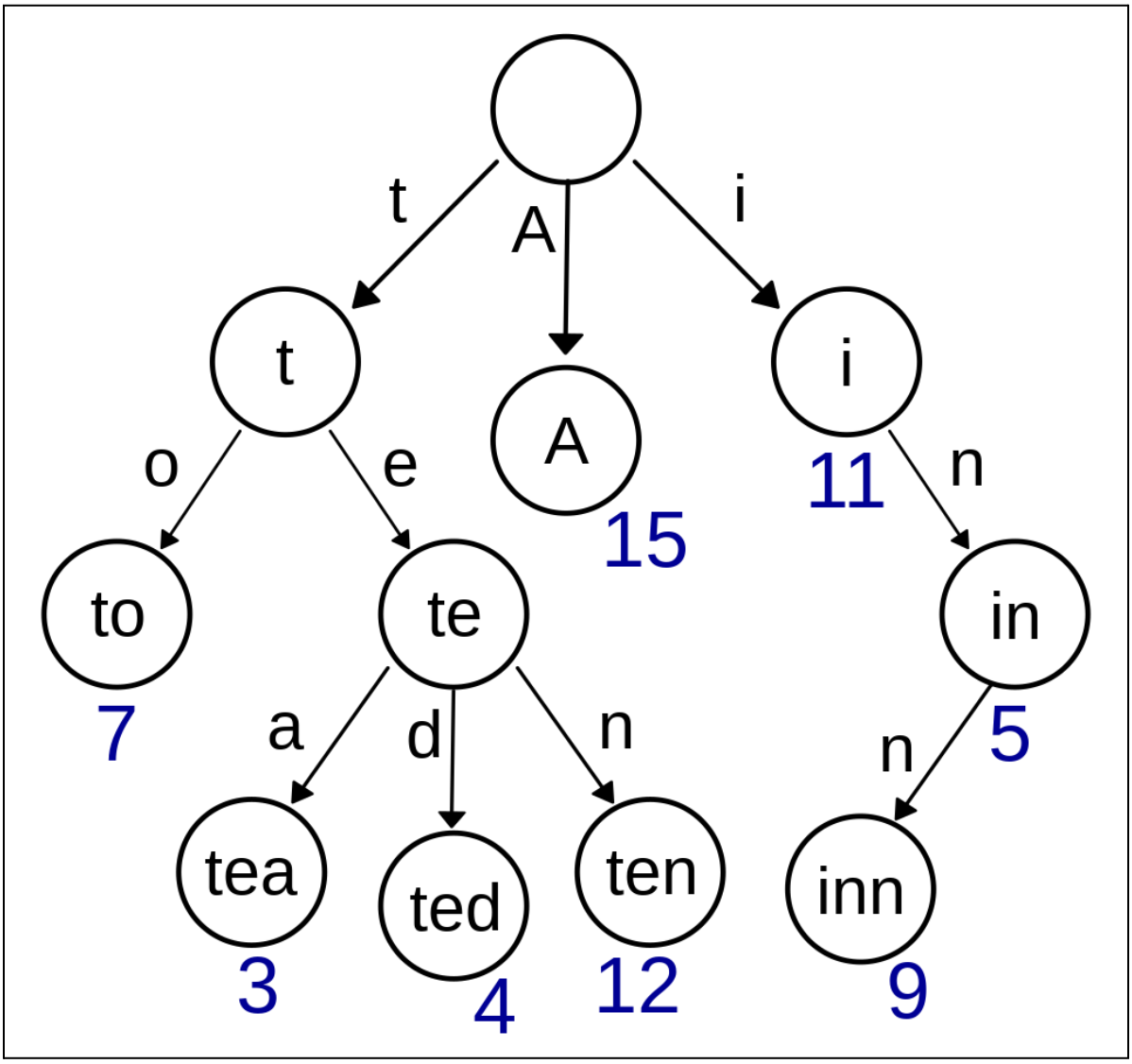


Question 8

Question 9

A. For decoding a morse code, we define a dictionary mapping from morse code letter to english letters. We then iterate over the letters of the input morse strings and add its corresponding English letter. This algorithm has $O(n)$ time complexity.

B. Given a list of morse codes of length n where each code starts with x, there are 2^n possible strings that can be generated by replacing the x letter with either dot or hash. To check whether these generated strings are valid strings (they belong to the dictionary file), we further need to run brute force search requiring $O(2^n * n * m)$ where m is the number of words in the dictionary. To optimize this approach, we leverage the trie data structure for testing whether a given string is valid or not. Trie builds a graph representing a prefix tree of the words in the dictionary. The below figure illustrates the trie data structure containing the words to, tea, ted, ten, etc. Words sharing the same prefix share the same edges in the graph. In our case, we first encode the words in the dictionary file to morse codes then build a graph for the encoded words (our alphabet is morse codes).
Using the trie data structure, for each word in the morse sequence, we replace x with either dash or dot and we utilize the trie data structure to discard strings that don't exist in the dictionary and we also no longer need to match the generated string against the dictionary because we store in the trie graph at each node whether it's an end of word.



The added complexity is to build the trie graph which is $O(mL)$ where L is the maximum word length in the dictionary.

C. This problem can be solved with BFS, DFS, or Dijsktra to find whether a path between the start node and the destination node exists. Given that the graph is unweighted, we use the DFS algorithm.
   - Graph representation: the grid might be very large and contains mostly walls (block type = 1). Therefore, instead of storing the graph as a two dimensional array, we store only the coordinates provided as input. To quickly access nodes in this graph, we store the nodes in a hash table. For a given node (x, y, block_type) we first calculate the node hash value using the below equation

     $node\_hash\_value = x * width + y$

And the node inverse function (which given hash value calculates the x and y):

$x = node\_hash\_value \,/\, width$
$y = node\_hash\_value \,\%\, width$

Then, we store in our hash table the corresponding block_type for each node.

The purpose of using hash table is to quickly access node in the graph without relying to using M * N two dimensional array.

- Traversal: we use stack to traverse the nodes and maintain a visited set. We use this visited set to mark nodes that have been visited in order not to visit them again. As explained in the question, we only visit nodes where the block type is open space (its value is 0). Additionally, we store for each node its parent node, meaning if we visited node v from node u then we store $parent\_node\_map[v] = u$. We'll use this hash table to reconstruct the path.
- Node neighbors: the graph in our case is implicit, meaning that we don't have clear nodes and edges. For a node u with coordinates (x,y) we define its neighboring nodes as the four adjacent nodes: (x+1,y), (x,y+1), (x-1,y) and (x,y-1). While generating the four neighboring nodes for any node (x,y) we check that these generated nodes lie within the maze grid.
- Path reconstruction: if we reach the destination node, we reconstruct the path from $parent\_node\_map$. We start from the destination node, then find its parent, then the parent of this parent, and so on until we reach the destination node which has the parent as -1 (to signify it has no parent).