Q1) Implementing Dijkstra's algorithm in Python

```
PS C:\Users\chols\OneDrive\Documents\Algorithms\HW6> python -u "c:\Users\chols\OneDrive\Documents\Algorithms\HW6\Dijkstra.py"
Starting Dijkstra's algorithm...
-------------------------------

Node(s) with Weight: 0 is added to the 'Visited' ['s']
Relaxed: vertex[A]: OLD: Infinity, NEW: 1, PATHS: {}
Relaxed: vertex[D]: OLD: Infinity, NEW: 4, PATHS: {'A': 's'}
Relaxed: vertex[G]: OLD: Infinity, NEW: 6, PATHS: {'A': 's', 'D': 's'}
Node(A) with Weight: 1 is added to the 'Visited' ['s', 'A', 'D', 'G']
Relaxed: vertex[B]: OLD: Infinity, NEW: 3, PATHS: {'A': 's', 'D': 's', 'G': 's'}
No edge relaxation is needed for node [D]

Relaxed: vertex[E]: OLD: Infinity, NEW: 3, PATHS: {'A': 's', 'D': 's', 'G': 's', 'B': 'A'}
Node(B) with Weight: 3 is added to the 'Visited' ['s', 'A', 'D', 'G', 'B', 'E']
Relaxed: vertex[C]: OLD: Infinity, NEW: 5, PATHS: {'A': 's', 'D': 's', 'G': 's', 'B': 'A', 'E': 'A'}
Node(E) with Weight: 3 is added to the 'Visited' ['s', 'A', 'D', 'G', 'B', 'E', 'C']
Relaxed: vertex[F]: OLD: Infinity, NEW: 6, PATHS: {'A': 's', 'D': 's', 'G': 's', 'B': 'A', 'E': 'A', 'C': 'B'}
Relaxed: vertex[G]: OLD: 6, NEW: 4, PATHS: {'A': 's', 'D': 's', 'G': 's', 'B': 'A', 'E': 'A', 'C': 'B', 'F': 'E'}
Relaxed: vertex[H]: OLD: Infinity, NEW: 5, PATHS: {'A': 's', 'D': 's', 'G': 'E', 'B': 'A', 'E': 'A', 'C': 'B', 'F': 'E'}
Relaxed: vertex[I]: OLD: Infinity, NEW: 6, PATHS: {'A': 's', 'D': 's', 'G': 'E', 'B': 'A', 'E': 'A', 'C': 'B', 'F': 'E', 'H': 'E'}
Node(G) with Weight: 4 is added to the 'Visited' ['s', 'A', 'D', 'G', 'B', 'E', 'C', 'F', 'H', 'I']
No edge relaxation is needed for node [H]

Node(D) with Weight: 4 is added to the 'Visited' ['s', 'A', 'D', 'G', 'B', 'E', 'C', 'F', 'H', 'I']
No edge relaxation is needed for node [E]

No edge relaxation is needed for node [G]
```

```
Node(C) with Weight: 5 is added to the 'Visited' ['s', 'A', 'D', 'G', 'B', 'E', 'C', 'F', 'H', 'I']
No edge relaxation is needed for node [E]

No edge relaxation is needed for node [F]

Relaxed: vertex[t]: OLD: Infinity, NEW: 9, PATHS: {'A': 's', 'D': 's', 'G': 'E', 'B': 'A', 'E': 'A', 'C': 'B', 'F': 'E', 'H': 'E', 'I': 'E'}
Node(H) with Weight: 5 is added to the 'Visited' ['s', 'A', 'D', 'G', 'B', 'E', 'C', 'F', 'H', 'I', 't']
No edge relaxation is needed for node [I]

Node(I) with Weight: 6 is added to the 'Visited' ['s', 'A', 'D', 'G', 'B', 'E', 'C', 'F', 'H', 'I', 't']
No edge relaxation is needed for node [t]

Node(F) with Weight: 6 is added to the 'Visited' ['s', 'A', 'D', 'G', 'B', 'E', 'C', 'F', 'H', 'I', 't']
No edge relaxation is needed for node [I]

No edge relaxation is needed for node [t]

Node(t) with Weight: 9 is added to the 'Visited' ['s', 'A', 'D', 'G', 'B', 'E', 'C', 'F', 'H', 'I', 't']
```

```
Final Shortest Path Results:
----------------------------
Visited (Shortest Distances):
  s: 0
  A: 1
  D: 4
  G: 4
  B: 3
  E: 3
  C: 5
  F: 6
  H: 5
  I: 6
  t: 9
```

```
Path (Predecessors):
  A: s
  D: s
  G: E
  B: A
  E: A
  C: B
  F: E
  H: E
  I: E
  t: C
Total Weight: 46
```

Q2: Application of Dijkstra's SSSP and MST algorithms

There are 15 cities in this planning trip program:

1. Atlanta
2. Boston
3. Chicago
4. Dallas
5. Denver
6. Houston
7. LA (Los Angeles)
8. Memphis
9. Miami
10. New York (NY)
11. Philadelphia
12. Phoenix
13. San Francisco (SF)
14. Seattle
15. Washington DC (WDC)

- Dijkstra's Algorithm
    - 1) Initialize our variables to be able to show what nodes were visited, what paths were taken, and a set to be contain all the nodes
    - 2) In the loop, the algorithm iterates over each of the nodes that have not been visited and each time the nodes are visited the node with the smallest distance is selected
    - 3) The selected node is then constructed using the path dictionary
    - 4) The algorithm updates the distance based on the current node and neighboring node distances
        - Which it then stores the shortest path after each iteration

```
Dijkstra's Algorithm Results:
-----------------------------------------------------
Distance from Denver to Denver: 0 with path (Denver)
Distance from Denver to Dallas: 1064 with path (Denver to Dallas)
Distance from Denver to LA: 1335 with path (Denver to LA)
Distance from Denver to Memphis: 1411 with path (Denver to Memphis)
Distance from Denver to Houston: 1426 with path (Denver to Dallas to Houston)
Distance from Denver to Chicago: 1474 with path (Denver to Chicago)
Distance from Denver to SF: 1894 with path (Denver to LA to SF)
Distance from Denver to Atlanta: 2221 with path (Denver to Dallas to Atlanta)
Distance from Denver to Washington: 2395 with path (Denver to Washington)
Distance from Denver to Phoenix: 2486 with path (Denver to Dallas to Phoenix)
Distance from Denver to Philadelphia: 2594 with path (Denver to Washington to Philadelphia)
Distance from Denver to NY: 2619 with path (Denver to Chicago to NY)
Distance from Denver to Boston: 2839 with path (Denver to Boston)
Distance from Denver to Seattle: 2879 with path (Denver to LA to Seattle)
Distance from Denver to Miami: 3194 with path (Denver to Dallas to Atlanta to Miami)
```

- Prim's Algorithm (MST)
  - 1) Initialize the variables to have one that stores the minimum edge weight, one to store the parent node for each node, and one to track if a node is included in the MST
  - 2) In the loop, the algorithm selected the node with the smallest key and have not been included in the MST
  - 3) It then updates the key and parent values based on what was selected

```
Prim's Algorithm Results:
-----------------------------------------------------
Denver is selected. Distance: 0
Dallas is selected. Distance: 1064
Houston is selected. Distance: 362
Memphis is selected. Distance: 675
Atlanta is selected. Distance: 1157
Miami is selected. Distance: 973
LA is selected. Distance: 1335
SF is selected. Distance: 559
Seattle is selected. Distance: 1092
Philadelphia is selected. Distance: 1413
Washington is selected. Distance: 199
Phoenix is selected. Distance: 1422
Chicago is selected. Distance: 1474
NY is selected. Distance: 1145
Boston is selected. Distance: 306
```

```
        Edge          Weight
-----------------------------------
          LA              LA ................1335
     Houston         Houston .................362
     Memphis         Memphis .................675
     Phoenix         Phoenix ................1422
  Washington      Washington .................199
       Miami           Miami .................973
     Atlanta         Atlanta ................1157
      Dallas          Dallas ................1064
          SF              SF .................559
     Chicago         Chicago ................1474
      Denver          Denver ...................0
Philadelphia    Philadelphia ................1413
     Seattle         Seattle ................1092
          NY              NY ................1145
      Boston          Boston .................306

Total MST:       13176
```

Q3: TinyZip and TinyUnzip

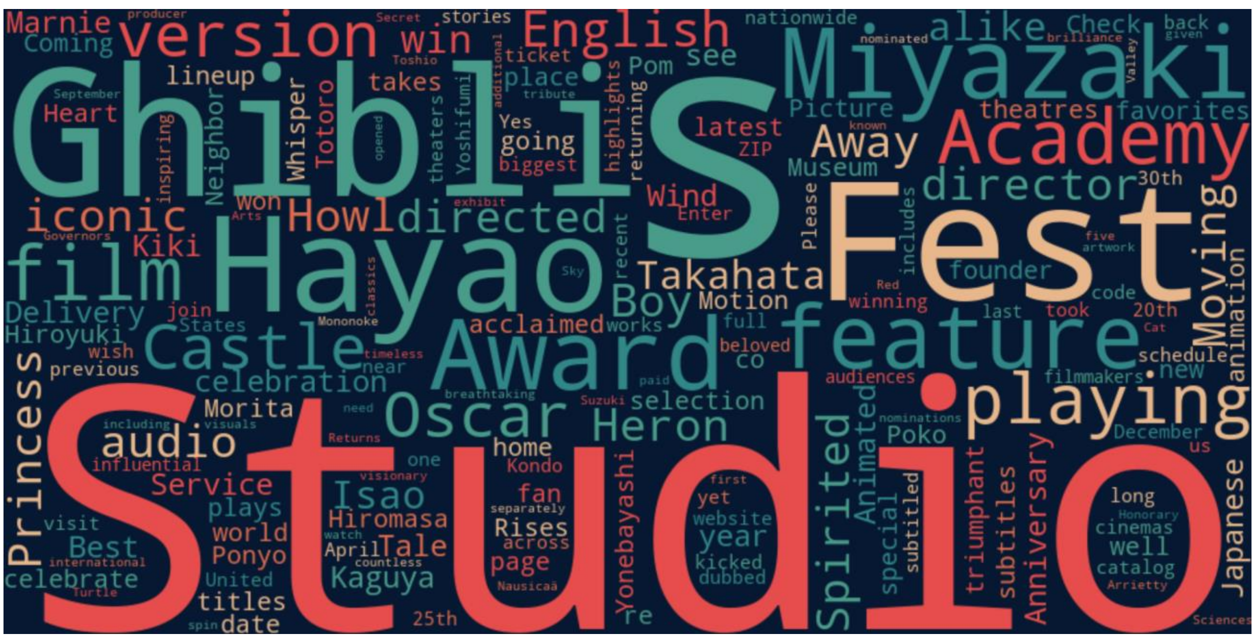
```
PS C:\Users\chols\OneDrive\Documents\Algorithms\HW6> python -u "c:\Users\chols\OneDrive\Documents\Algorithms\HW6\TinyUnZip.py"
Enter a file name to encode: King.txt
------------------------------------------------------------------------------
Character               Weight          Huffman Code
------------------------------------------------------------------------------
r                               413                     0000
s                               419                     0001
I                               23                      00100000
C                               6                       0010000100
P                               3                       00100001010
Y                               3                       00100001011
L                               12                      001000011
N                               24                      00100010
q                               6                       0010001100
z                               6                       0010001101
M                               7                       0010001110
€                               1                       001000111100
-                               2                       001000111101
:                               2                       001000111110
;                               2                       001000111111
k                               51                      0010010
'\n'                            58                      0010011
f                               220                     00101
n                               450                     0011
e                               885                     010
b                               110                     011000
y                               124                     011001
d                               254                     01101
a                               539                     0111
i                               542                     1000
T                               14                      100100000
!                               8                       1001000010
```

```
'                               8                       1001000011
?                               1                       1001000100000
D                               1                       1001000100001
E                               1                       1001000100010
H                               1                       1001000100011
B                               4                       10010001001
F                               4                       10010001010
J                               1                       1001000101100
R                               1                       1001000101101
â                               1                       1001000101110
”                               1                       1001000101111
S                               8                       1001000110
G                               9                       1001000111
,                               71                      1001001
w                               146                     100101
.                               75                      1001100
A                               18                      100110100
W                               18                      100110101
j                               20                      100110110
O                               4                       10011011100
x                               5                       10011011101
"                               12                      1001101111
g                               167                     100111
o                               604                     1010
t                               656                     1011
l                               328                     11000
v                               81                      1100100
p                               93                      1100101
u                               175                     110011
c                               176                     110100
m                               181                     110101
h                               385                     11011
```

```
' '                             1623                   111
---------------------------------------------------------------------------
Results:
---------------------------------------------------------------------------
Expected cost of Huffman code: 39683
Expected cost of ASCII: 72504
Huffman efficiency improvement over ASCII code: 45.27%
Expected cost of optimal FCL cost: 54378
Huffman efficiency improvement over FCL: 27.02%
The size of King.txt: 9063 bytes
The size of King.zip: 4961 bytes
The size of King.unzipped.txt: 9122 bytes
PS C:\Users\chols\OneDrive\Documents\Algorithms\HW6>
```

Extra credit problems: Word Cloud and OBST

- 1. Read all the references of REST and Web API?
  - Yes
- 2. Test Results between BST and OBST
  - The cost for the OBST was almost 6 times less than the BST cost, what the BST does is sorts the words in a way to make them searchable. OBST is another binary search tree but this one minimizes the search cost by considering levels and word frequencies. OBST costs less because it chooses an optimal root for subtrees to minimize the cost as much as possible.




```
PS C:\Users\chols\OneDrive\Documents\Algorithms\HW6> python -u "c:\Users\chols\OneDrive\Documents\Algorithms\HW6\BSTvsOBST.py"
Regular BST traversal cost: 418.93
Optimal BST traversal cost: 74.93
```