910 Exercise3 Report
Name: Jiyu Yan
Student ID:1851015

Q1

(a) $\frac{60*60*817}{4} = \frac{2941200}{4} = 735300$

(b) $\sqrt[3]{\frac{2941200}{3}} = 99$

(c) $\frac{2941200-10}{100000} = 2.94199 \rightarrow \log n < 2.4119 \rightarrow n < 2^{2.94119} = 714271828$

(d) $\frac{2941200*2}{87} = 67613$

(e) $log2941200 = 21$

Q2

(a) True. $8n^2 + 4 < 9n^2$ for $n > 2$, $9n^2 \sim O(n^2)$

(b) True. $16n^3 - 4n + 3 < n^4$ for $n > 16$

(c) True. $3n^4 - n^2 + 7 > n^2$ for $n > 1$

(d) False. Can not find any k and c to satisfy $3n^4 - 2n^2 + 3 < c * n^2$ for $n > k$,
$3n^4 - 2n^2 + 3 < 4 * n^4$ for $n > 1$. $4n^4 \sim O(n^4)$.

(e) True. $6n + 143 < 7n$ for $n > 143$, $7n \sim O(n)$
$6n + 143 > 5n$ for $n > 0$, $5n \sim O(n)$

Q3

(a) $log_b a = log_2 8 = 3 > c = 1 \rightarrow T(n) = \theta(n^3)$

(b) Can't apply because here $a = 2^n$ which is not a integer number of sub questions.

(c) Can't apply because $\frac{2}{3}$ is not a integer.

(d) $log_b a = log_4 16 = 2 = c \rightarrow f(n) = \theta(n^2 * log^k n) = \theta(n^2) \rightarrow k = 0 \rightarrow T(n) = \theta(n^2 * logn)$

(e) $log_b a = log_3 9 = 2 < c = 3 \rightarrow T(n) = \theta(n^3)$

Q4

(a) Insertion sort. Because it insert i element to first sorted i-1 elements by compare i with the last one(i-1) at first. For this input lists it just need to compare several times and insert 5 before 6.

(b) Quick sort. The pivot need to be picked from the middle in the list. For this case the pivot would be 3 and after just one switch 6(bigger than 3) and 1(smaller than 3), the list is already sorted. It also save memory usage compared to merge sort.

(c) Merge sort. This is the worst cast for some versions of quick sort. Merge sort could make sure the $O(nlogn)$ complexity for this reversed sorted list.

(d) Quick sort. The pivot need to be picked from the middle in the list(which is 4 here, it's better than 1 in the first place). The time complexity is $O(nlogn)$.

(e) Merge sort. It is stable so equal elements keep their original order.

Q5

stage	current vertex	labels and distances											
		A		B		C		D		E		F	
0	-	A	0	-	∞	-	∞	-	∞	-	∞	-	∞
1	A	A	0	A	20	A	47	-	∞	-	∞	-	∞
2	B	A	0	A	20	A	47	B	40	B	27	-	∞
3	E	A	0	A	20	A	47	E	30	B	27	E	35
4	F	A	0	A	20	A	47	E	30	B	27	E	35
5	D	A	0	A	20	A	47	E	30	B	27	E	35
6	C	A	0	A	20	A	47	E	30	B	27	E	35

Q6

stage	current vertex	labels and distances											
		A		B		C		D		E		F	
0	-	A	0	-	∞	-	∞	-	∞	-	∞	-	∞
1	A	A	0	A	19	-	∞	-	∞	-	∞	-	∞
2	B	A	0	A	19	-	∞	-	∞	B	32	-	∞
3	E	A	0	A	19	E	48	E	35	B	32	E	33
4	F	A	0	A	19	E	48	E	35	B	32	E	33
5	D	A	0	A	19	E	48	E	35	B	32	E	33
6	C	A	0	A	19	E	48	E	35	B	32	E	33

Q7

(a)

stage	V	E
0	(A)	()
1	(A,C)	((A,C))
2	(A,C,E)	(A,C),(C,E))
3	(A,C,E,F)	(A,C),(C,E),(E,F))
4	(A,C,E,F,D)	(A,C),(C,E),(E,F),(E,D))
5	(A,C,E,F,D,B)	(A,C),(C,E),(E,F),(E,D),(E,B))
6	(A,C,E,F,D,B,G)	(A,C),(C,E),(E,F),(E,D),(E,B),(C,G))

(b)

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

Q8

(a) **Morse Code**

For this part, I implement a morsedict function which store a dictionary of morse code. Key is the morse code like '.-' and value is the number or letter related to it. It receives a morse code and returns that letter or number.

```
1     mdict = {'.-': 'A',
2         '-...': 'B',
3         ...}
```

In morseDecode function, check every letter from input string with morsedict above and append each value to a new list. So the string in this list is the answer.

(b) **Incomplete Morse Code**

```
1   def guesslist(num):
2     guess = [''.join(i) for i in (itertools.product(['.', '-'], repeat=num))]
3     return guess
```

The num in guesslist function is the length of the input and hence is the number of 'x' in list. Using the product from itertools here could generate all possible replacement for every 'x' in list. Then we just need to change every 'x' in list using

the output. The complexity of this function is 2^n.(n is the length of input list and the num)

```
def morse_all(test):
xnum = len(test)
guess = guesslist(xnum)
all = []
for replace in guess:
  possible = []
  i = 0
  for each in test:
    each = replace[i] + each[1:]
    possible.append(each)
    i += 1
  all.append(possible)
return all
```

This morse_all function could replace all 'x' in list with '.' or '-' generated from guesslist function. For example the input length is 2, ['x', 'x.'], then output would be all 4 possible replacements, [['.', '..']['.', '-.']['-', '..']['-', '-.']]. Based on function above with 2^n , since 'each in test' has n length, so far the complexity is $(3n+3)*2^n$

```
def morsePartialDecode(inputStringList):
  dicwords = inputdic('./dictionary.txt')
  allpossible = morse_all(inputStringList)
  ans = []
  for i in allpossible:
    word = morseDecode(i)
    if word in dicwords:
      ans.append(word)
  return ans
```

This function could decode all possible morse code list getting from morse_all function and compare to dictionary, then output all valid words. The total complexity is $(3n + 3) * 2^n + 3 * 2^n \sim (3n + 6) * 2^n$

(c) **The Maze**

The main idea is based on A* algorithm. Because it is nearly the best solution for finding route in maze. The big improving compared with other finding path algorithms is that A* not only compute the current shortest path from the start point, but also estimate the shortest path from current point to the end position. In my implemention I use Manhattan distance to compute and estimate the distance between 2 points.

```
class Maze:
  def __init__(self):
    self.width = 0
    self.height = 0
    self.information = collections.defaultdict(lambda: '*')
```

The information here is a dictionary for every coordinate in maze.

```
def addCoordinate(self, x, y, blockType):
  if x >= self.width:
    self.width = x+1
  if y >= self.height:
    self.height = y+1
```

```
    block = '␣' if blockType == 0 else '*'
    self.information[x, y] = block
```

The default value of 'self.information' is '*' meaning wall and it would be ' ' meaning open space when you add '0'. The width and height would change by current input.

```
def printMaze(self):
  maze = [['*'] * self.width for i in range(self.height)]
  for i in self.information:
    if self.information[i] == '␣':
      maze[i[1]][i[0]] = self.information[i]

  for i in maze:
    print('|'.join(i))
```

For printing a maze.

```
def manDistance(self,x1 ,y1, x2, y2):
  mdistance = abs(x1-x2) + abs(y1-y2)
  return mdistance
```

Computing and evaluating distance between two nodes with Manhattan distance.

```
def findRoute(self, x1, y1, x2, y2):
  openset = set()
  closedset = set()
  parent = {}
  route = []
  g_node = {}

  def f_node(node):
    fvalue = g_node[node] + self.manDistance(node[0], node[1], x2, y2)
    return fvalue

  def min_f(set):
    fmin = 1000000
    for each in set:
      f = f_node(each)
      if f < fmin:
        fmin = f
        x = each[0]
        y = each[1]
    return x, y

  def reRoute(node):
    route.append(node)
    if node == (x1, y1):
      route.reverse()
      return route
    reRoute(parent[node])

  openset.add((x1, y1))
  g_node[x1, y1] = 0
  while len(openset) is not 0:
    current = min_f(openset)
    if current == (x2, y2):
      reRoute(current)
    openset.remove(current)
    closedset.add(current)
    temp = set()
```

```
38      if self.information[current[0], current[1]+1] == '␣':
39        temp.add((current[0], current[1]+1))
40      if self.information[current[0], current[1]-1] == '␣':
41        temp.add((current[0], current[1] - 1))
42      if self.information[current[0]+1, current[1]] == '␣':
43        temp.add((current[0]+1, current[1]))
44      if self.information[current[0]-1, current[1]] == '␣':
45        temp.add((current[0]-1, current[1]))
46
47      for eachNeighbor in temp:
48        if eachNeighbor in closedset:
49          continue
50        new_g = 1 + g_node[current]
51
52      if eachNeighbor not in openset:
53        openset.add(eachNeighbor)
54      elif g_node[eachNeighbor] <= new_g:
55        continue
56      g_node[eachNeighbor] = new_g
57      parent[eachNeighbor] = current
58    return route
```

The main idea based on A* algorithm is selecting the path that minimizes $f(n) = g(n) + h(n)$. n is the next node, g(n) is the cost from start to n and h(n) estimates cost from n to the end node. The f_node() function is used for computing f(n), g_node for g(n), min_f() is used to find coordinate that minimize f(n). Openset stores nodes need to be calculate and closedset stores nodes after calculation. Function reRoute() would find a list of (x,y) tuples that map a route.