CS917 – Algorithms Coursework

Question 1

For each function $f(n)$ and time t (represented in microseconds), we want the largest integer n that satisfies $f(n) \leq t$. For n^2, n^3 and 2^n we calculate $\sqrt{t}$, $\sqrt[3]{t}$, $log_2 t$ and round them down (the $n+1$ tests take care of loss of precision in float point calculation, for example without them the result for $f(n) = nlog_2 n$ and $t = 1$ second would be is $int(99.99999999999997) = 99$ rather than 100). As for $nlog_2 n$ and $n!$, we start from $\sqrt{t}$ and 1, trying each integer one by one.

```
import math

second = 1000000
minute = 60 * second
hour = 60 * minute
day = 24 * hour
time_t = {'second': second, 'minute': minute, 'hour': hour, 'day': day}

def n_log2_n(time):
    n = int(math.sqrt(time))
    while n * math.log2(n) < time:
        n += 1
    return n - 1

def n_square(time):
    n = int(math.sqrt(time))
    if math.pow(n + 1, 2) > time:
        return n
    else:
        return n + 1

def n_cube(time):
    n = int(math.pow(time, 1/3))
    if math.pow(n + 1, 3) > time:
        return n
    else:
        return n + 1

def exp_2_n(time):
    n = int(math.log2(time))
    if math.pow(2, n + 1) > time:
        return n
    else:
        return n + 1

def fact_n(time):
    n = 1
    fact = 1
    while fact < time:
        n += 1
        fact *= n
    return n - 1
```

```
for key in time_t.keys():
    time = time_t[key]
    print("n for nlog_2n in 1 {} is {}".format(key, n_log2_n(time)))
    print("n for n^2 in 1 {} is {}".format(key, n_square(time)))
    print("n for n^3 in 1 {} is {}".format(key, n_cube(time)))
    print("n for 2^n in 1 {} is {}".format(key, exp_2_n(time)))
    print("n for n! in 1 {} is {}".format(key, fact_n(time)))
```

The largest size of a problem that can be solved in time t for each function $f(n)$:

	1 **second**	1 **minute**	1 **hour**	1 **day**
$nlog_2n$	$62,746$	$2,801,417$	$133,378,058$	$2,755,147,513$
n^2	$1,000$	$7,745$	$60,000$	$293,938$
n^3	100 *	391	$1,532$	$4,420$
2^n	19	25	31	36
$n!$	9	11	12	13

Question 2

(a) $10n^2 + 9$ in $O(n^2)$. Let $c = 11$, $k = 3$, $10n^2 + 9 < 11n^2$ for all $n > 3$.

(b) $5n^4 - 4n^2 + 3$ in $\Omega(n^4)$. Let $c = 4$, $k = 1$, $5n^4 - 4n^2 + 3 \geq 4n^4$ for all $n > 1$.

(c) $19n^3 - 8n$ in $O(n^4)$. Let $c = 1$, $k = 19$, $19n^3 - 8n < n^4$ for all $n > 19$.

(d) $3n^4 + 2n^2 + n$ not in $\Theta(n^2)$. Cannot find $c_1 > 0$, $k \geq 1$ such that $c_1 \cdot n^2 \geq 3n^4 + 2n^2 + n$ for all $n > k$. Actually, $3n^4 + 2n^2 + n$ is $\Omega(n^2)$ but not $O(n^2)$, and thus it is not $\Theta(n^2)$.

(e) $2n^2 + 16n + 115$ in $\Omega(n)$. Let $c = 16$, $k = 1$, $2n^2 + 16n + 115 \geq 16n$ for all $n > 1$.

Question 3

(a) The number of sub problems 2^n is not a constant, the master theorem is not applicable.

(b) $a = 16$, $b = 4$, $c = 2$, $c = log_b a$, apply case 2.

$f(n) = 16n^2 = \Theta(n^2 \cdot log_k n)$ for $k = 0$, $T(n) = \Theta(n^2)$.

(c) The size by which the problem is divided is smaller than one, theorem not applicable.

(d) $a = 2$, $b = 4$, $c = 2$, $c > log_b a$, but we only know $f(n) = O(n^c)$, not $f(n) = \Omega(n^c)$, so both case 1 and case 3 are not applicable, the master theorem is not applicable.

(e) $a = 8$, $b = 4$, $c = 2$, $c > log_b a$, apply case 3.

$8(n/4)^2 \leq dn^2$ for $d = 1/2$, $T(n) = \Theta(n^2)$.

Question 4

(a) The list is already ordered, so we should use an adaptive sorting algorithm, adaptive bubble sort and in-place insertion sort would do, both with $O(n)$ complexity for the "best case".

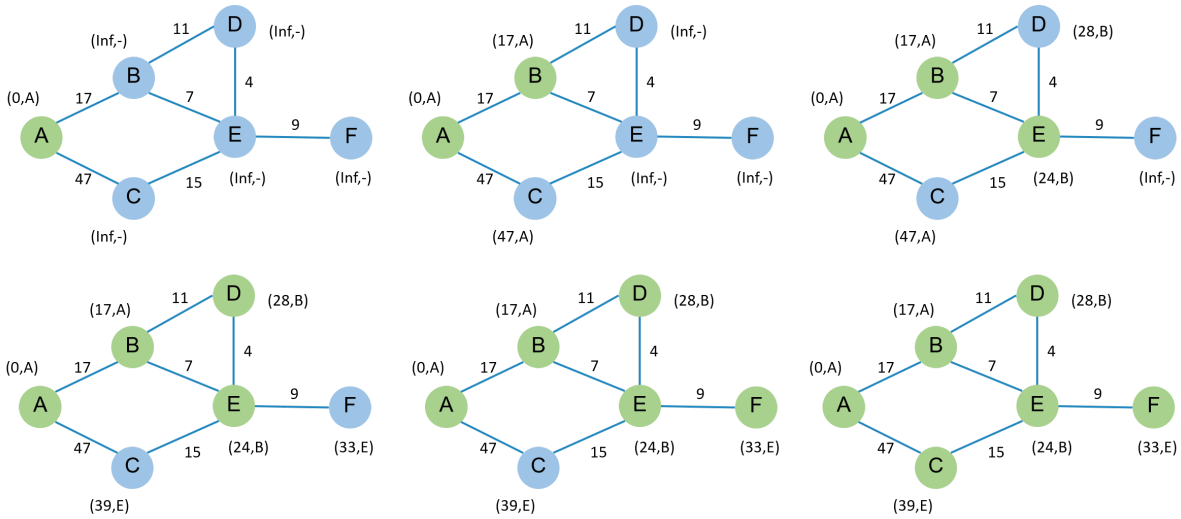
(b) The list comes in the exact reverse order, so we should consider the sorting algorithm with the best complexity for "worst case", which is merge sort with $O(n \cdot logn)$.

(c) The list is closer to an average case, we may pick quick sort or merge sort for the best average complexity $O(n \cdot logn)$, and quick sort if we want lower memory usage.

(d) Two objects with the same values need to preserve their order after sorting, so we need a stable sorting algorithm. Bubble sort, insertion sort and merge sort all satisfy the requirement. Take complexity into consideration and we may choose merge sort from the three.

(e) The ordering of the values must be preserved where the keys are equivalent, this is also about stability of the sorting algorithm, so we will use merge sort as well.

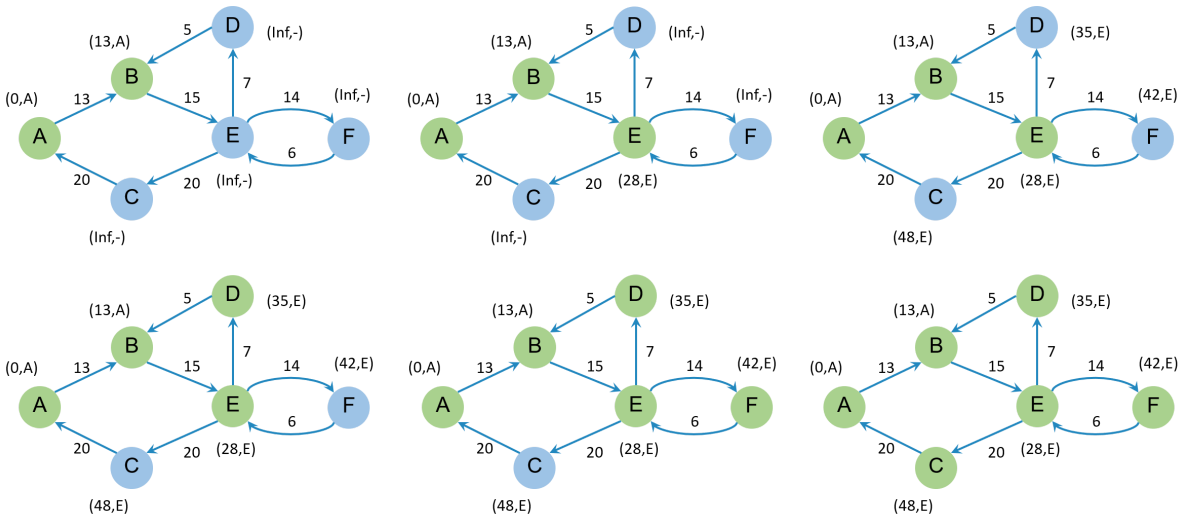
Question 5



Starting from vertex A, visit its neighbours B and C, update their distance to the weight of AB and AC. Pick B with the smaller distance, visit its neighbours D & E, update their distance to $d(B) + w(BD)$ and $d(B) + w(BE)$. Pick E with the smallest distance, visit its neighbours C, D and F, D does not need update, update C's distance to $d(E) + w(CE)$ and F's distance to $d(E) + w(EF)$. Pick D, no updates. Pick F, no updates. Pick C, no updates. All vertices have been processed. The shortest path for all vertices from vertex A is listed as below:

vertex	path	distance
A	A	0
B	$A \rightarrow B$	17
C	$A \rightarrow B \rightarrow E \rightarrow C$	39
D	$A \rightarrow B \rightarrow D$	28
E	$A \rightarrow B \rightarrow E$	24
F	$A \rightarrow B \rightarrow E \rightarrow F$	33

Question 6



Starting from vertex A, visit its neighbour B, update its distance to the weight of AB. Go to B, visit its neighbour E, update its distance to $d(B) + w(BE)$. Go to E, visit its neighbours C, D and F, update C's distance to $d(E) + w(CE)$, D's distance to $d(E) + w(DE)$ and F's distance to $d(E) + w(EF)$. Pick D, no updates. Pick F, no updates. Pick C, no updates. All vertices have been processed. The shortest path for all vertices from vertex A is listed as below:

vertex	path	distance
A	A	0
B	$A \to B$	13
C	$A \to B \to E \to C$	48
D	$A \to B \to E \to D$	35
E	$A \to B \to E$	28
F	$A \to B \to E \to F$	42

Question 7

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)	$((A, B), (B, C), (C, G), (B, D), (D, E), (E, F))$

(a) Apply Prim's Algorithm. Starting from vertex A, pick the edge with the smallest weight from A to all the other vertices, AB, add B to V and (A, B) to E. Pick the edge with smallest weight from (A, B) to all other vertices, BC, add C to V and (B, C) to E... Keep doing this until all vertices in the original graph were added to V and E is the minimum spanning tree we wanted.

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

(b) Apply Kruskal's Algorithm. Initialise each vertex to be a separate component. Put the edges in weight ascending order, for each round take the first edge (the smallest weight). If it joins two separate components, add it to the result; if it would form a circle inside one component, discard it. Keep doing this until there is only one component or no edges left. We get a minimum spanning tree or forest. In this graph a MST, and for each round the edge we extract from queue happen to link two existing components, therefore they were all kept rather than discarded.

Question 8

```
Algorithm 2 RIGHT-ROTATE(T, y)
--------------------------------------------------------------------------------
Require: y.left ≠ T.nil, T.root.p == T.nil
 1: x = y.right                  // set x
 2: y.left = x.right             // turn x's right substree into y's left subtree
 3: if x.right ≠ T.nil then
 4:    x.right.p = y
 5: end if
 6: x.p = y.p                    // link y's parent to x
 7: if y.p == T.nil then
 8:    T.root = x
 9: else if y == y.p.left then
10:    y.p.left = x              // link x to be the left child of y's parent
11: else
12:    y.p.right = x
13: end if
14: x.right = y                  // put y on x's right
15: y.p = x
```

Question 9

(a) Morse Code: store the mapping from Morse string to letters in a dictionary, look it up for each string in input list and concatenate the letters we get to form a word. The test for whether a string exists in the dict keys is to take precautions against illegal Morse string inputs.

```
def morseDecode(inputStringList):
    word = ""
    for sstr in inputStringList:
        if sstr in morse_code.keys():
            word += morse_code[sstr]
    return word
```

(b) Incomplete Morse Code: store all words from the dictionary file in a list. For every Morse string from the input list, find all the indices with missing symbols represented by 'x' (for the given test cases only the first character of every Morse-code letter is missing, but we can generalise the question to allow more than one missing symbols at any position in the string). Then we can use these indices to replace the x's with '-' and '.', generating all possible Morse strings from this incomplete string. When we have translated the strings to all possible letters at this position in the word, we can update the word list by adding these letters to the end of all existing words (processed up to pos - 1). Do this iteratively and eventually we end up with all the possible words for the input Morse list. We then check if each word exists in our list of dictionary words, if a word is in the list we keep it, otherwise we discard it. The remaining words are all valid.

```
Function MORSE-PARTIAL-DECODE(inputStringList)
-------------------------------------------------------------------------------
 1: words <-- [""]                      # a list containing only one empty string
 2: forall str in inputStringList do
 3:    missing_indices <-- []
 4:    forall index in 0..len(str)-1 do
 4:       if str[index] == 'x' then
 5:          add index to missing_indices
 6:       end
 7:    end
 8:    cases <-- [sstr]                 # a list containing the original string
 9:    forall index in missing_indices do
10:       forall case in cases do
11:          replace case with two new ones where \
12:          case[index] is set to '.' and '-' respectively
13:       end
14:    end
15:    forall word in words do
16:       replace word with word + letter \
17:       for all letters translated from cases (possible Morse strings)
18:    end
19: end
20: valid_words <-- [word in words where word in dict]
```

(c) The Maze: the class has three variables, a list $maze$ to store (x, y) pairs for open spaces, the highest x, y values that have been passed so far, max_x and max_y. We don't store (x, y) for walls but simply assume block type is wall for all (x, y) where $x \in 0..max\_x$, $y \in 0..max\_y$ and $(x, y) \notin maze$. We initialise $maze$ to an empty list, max_x and max_y to -1.

For the function $addCoordinate$, add (x, y) to $maze$ if block type is open space and it does not exist in the list yet, remove it from $maze$ if block size is wall and it does exist (in this case we assume the change of a position that have previously been set to another block type is allowed). We also update max_x and max_y if the newly added (x, y) exceeds those boundaries.

For $printMaze$ we traverse through all (x, y) pairs for $x \in 0..max\_x$ and $y \in 0..max\_y$, if (x, y) is in $maze$ we print an empty space, otherwise we use the star character.

For $findRoute$, we output an empty list if (x_1, y_1) or (x_2, y_2) is not in $maze$. Otherwise we perform a DFS search starting from (x_1, y_1). Here we use a copy of the list $maze$ (to not influence the original one kept by the class) to record which part of the route we have already been through, so that we don't search the same route twice. We start with a list containing only the starting point (x_1, y_1), for each step we take the last coordinate from the list, (x, y), and search in all four directions $(x - 1, y)$, $(x + 1, y)$, $(x, y - 1)$ and $(x, y + 1)$, for the first coordinate we come across that is in our local copy of $maze$, we take it as the next step add it to our route, and remove it from local $maze$ copy (marked as visited and will not search in that direction again later as we move backwards). We don't need to worry about out of range indices here because they won't be in the $maze$ anyway. If all four directions are not in local $maze$, we have come to a dead end. In this case we just remove the last coordinate (x, y) from our route and move on to the next loop, where we will be dealing with the coordinate that was previously just prior to (x, y) in our route: we take one step back and try searching in other unvisited directions from there. Our stopping criteria is if (x_2, y_2) is in the route after a loop (output the route and that's what we're looking for), or if the route is empty (this is to say we have visited all the dead ends in the maze and failed to reach (x_2, y_2), thus no route is available, and we just return this empty list).