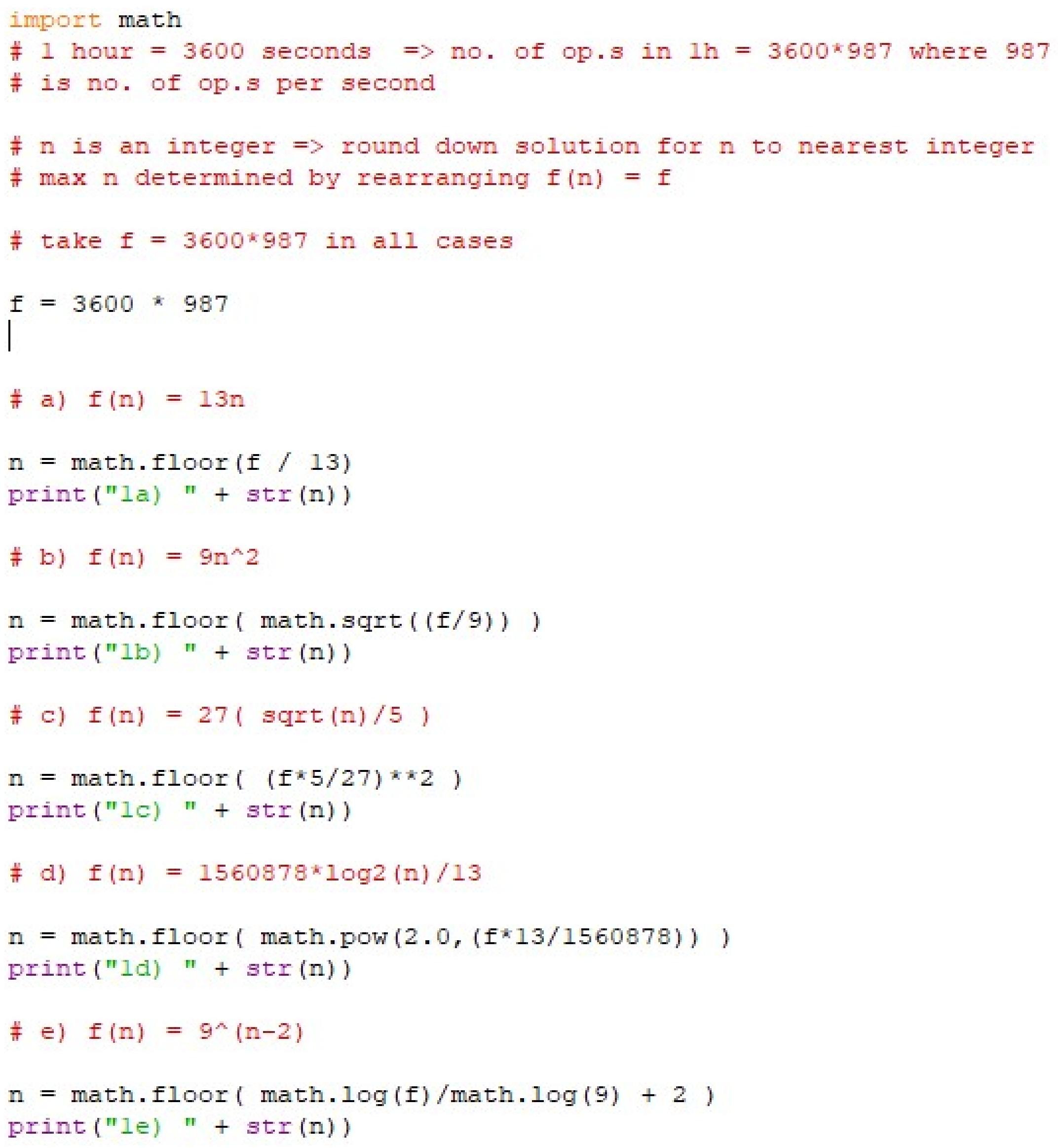
CS917 PX425

Algorithms Coursework Solutions

Gregory Wright 1703762

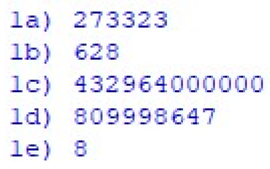
Question 1

Python Code / workings:



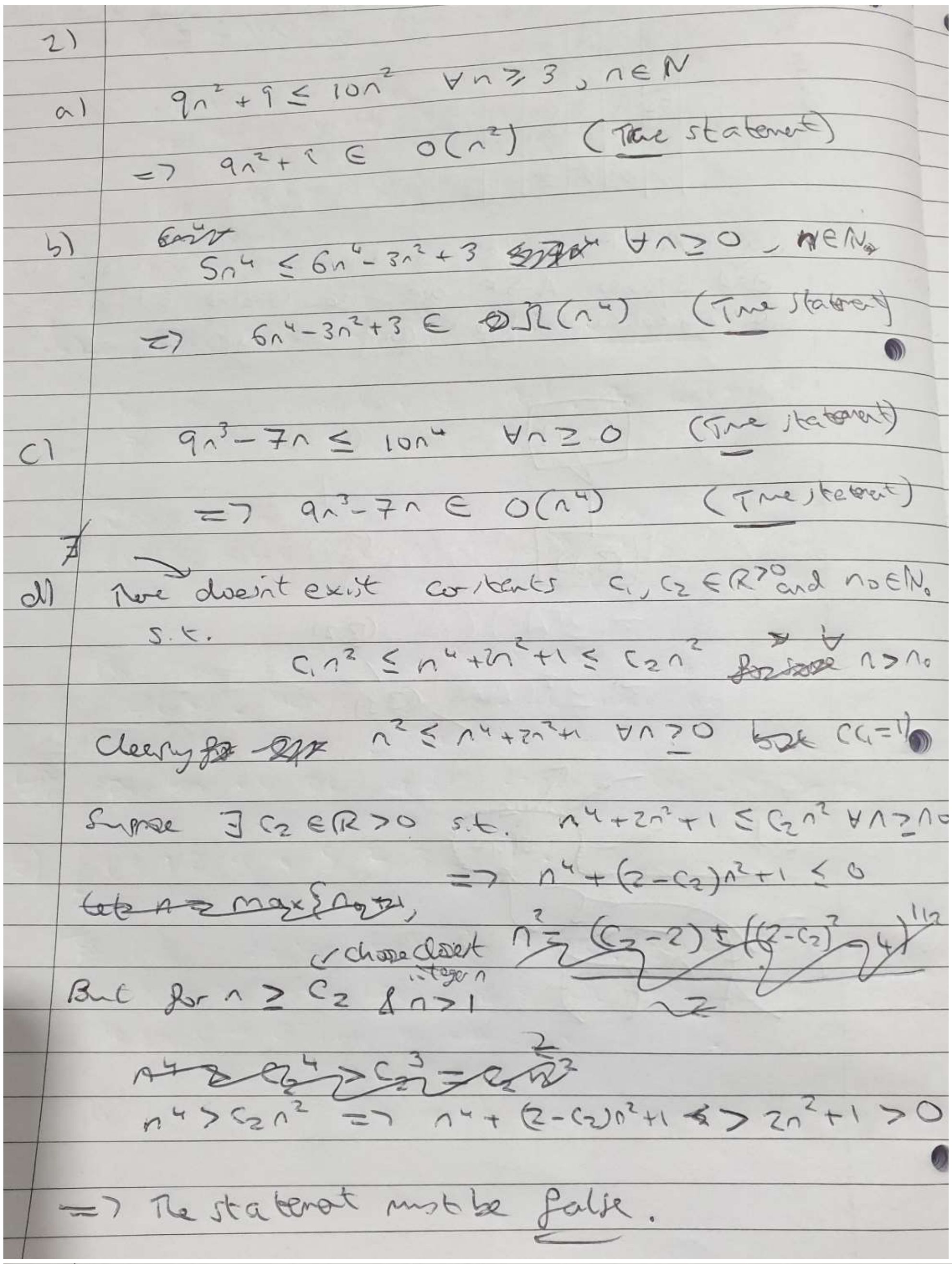
```
import math
# 1 hour = 3600 seconds  => no. of op.s in 1h = 3600*987 where 987
# is no. of op.s per second

# n is an integer => round down solution for n to nearest integer
# max n determined by rearranging f(n) = f

# take f = 3600*987 in all cases

f = 3600 * 987


# a) f(n) = 13n

n = math.floor(f / 13)
print("1a) " + str(n))

# b) f(n) = 9n^2

n = math.floor( math.sqrt((f/9)) )
print("1b) " + str(n))

# c) f(n) = 27( sqrt(n)/5 )

n = math.floor( (f*5/27)**2 )
print("1c) " + str(n))

# d) f(n) = 1560878*log2(n)/13

n = math.floor( math.pow(2.0,(f*13/1560878)) )
print("1d) " + str(n))

# e) f(n) = 9^(n-2)

n = math.floor( math.log(f)/math.log(9) + 2 )
print("1e) " + str(n))
```

Output:



```
1a) 273323
1b) 628
1c) 432964000000
1d) 809998647
1e) 8
```

Question 2



2)

a) $9n^2 + 9 \leq 10n^2 \quad \forall n \geq 3,\ n \in \mathbb{N}$

$\Rightarrow 9n^2 + 9 \in O(n^2)$ (True statement)

b) $5n^4 \leq 6n^4 - 3n^2 + 3 \quad \forall n \geq 0,\ n \in \mathbb{N}_0$

$\Rightarrow 6n^4 - 3n^2 + 3 \in \Omega(n^4)$ (True statement)

c) $9n^3 - 7n \leq 10n^4 \quad \forall n \geq 0$ (True statement)

$\Rightarrow 9n^3 - 7n \in O(n^4)$ (True statement)

d) There doesn't exist constants $c_1, c_2 \in \mathbb{R}^{>0}$ and $n_0 \in \mathbb{N}_0$ s.t.

$c_1 n^2 \leq n^4 + 2n^2 + 1 \leq c_2 n^2 \quad$ ~~for some~~ $\forall\ n > n_0$

Clearly ~~for~~ $n^2 \leq n^4 + 2n^2 + 1 \quad \forall n \geq 0$ take $c_1 = 1$

Suppose $\exists\ c_2 \in \mathbb{R} > 0$ s.t. $n^4 + 2n^2 + 1 \leq c_2 n^2 \quad \forall n \geq n_0$

$\Rightarrow n^4 + (2 - c_2)n^2 + 1 \leq 0$

~~Let $n \geq \max\{n_0, 1\}$,~~

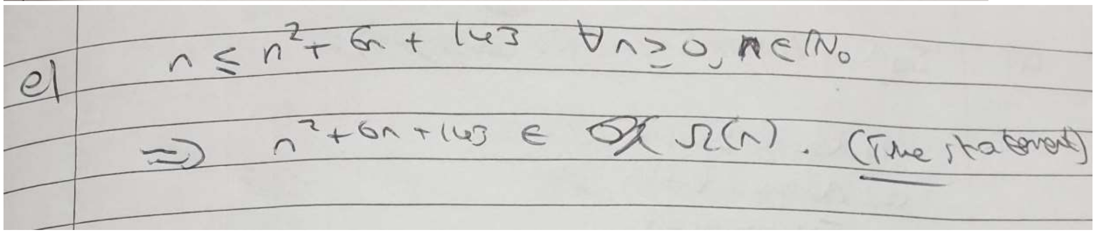
~~$n^2 = (c_2 - 2) \pm ((2-c_2)^2 - 4)^{1/2}$~~

But for $n \geq c_2$ & $n > 1$ (choose closest integer n)

~~$n^4 \geq c_2^4 > c_2^3 = c_2 n^3$~~

$n^4 > c_2 n^2 \Rightarrow n^4 + (2 - c_2)n^2 + 1 > 2n^2 + 1 > 0$

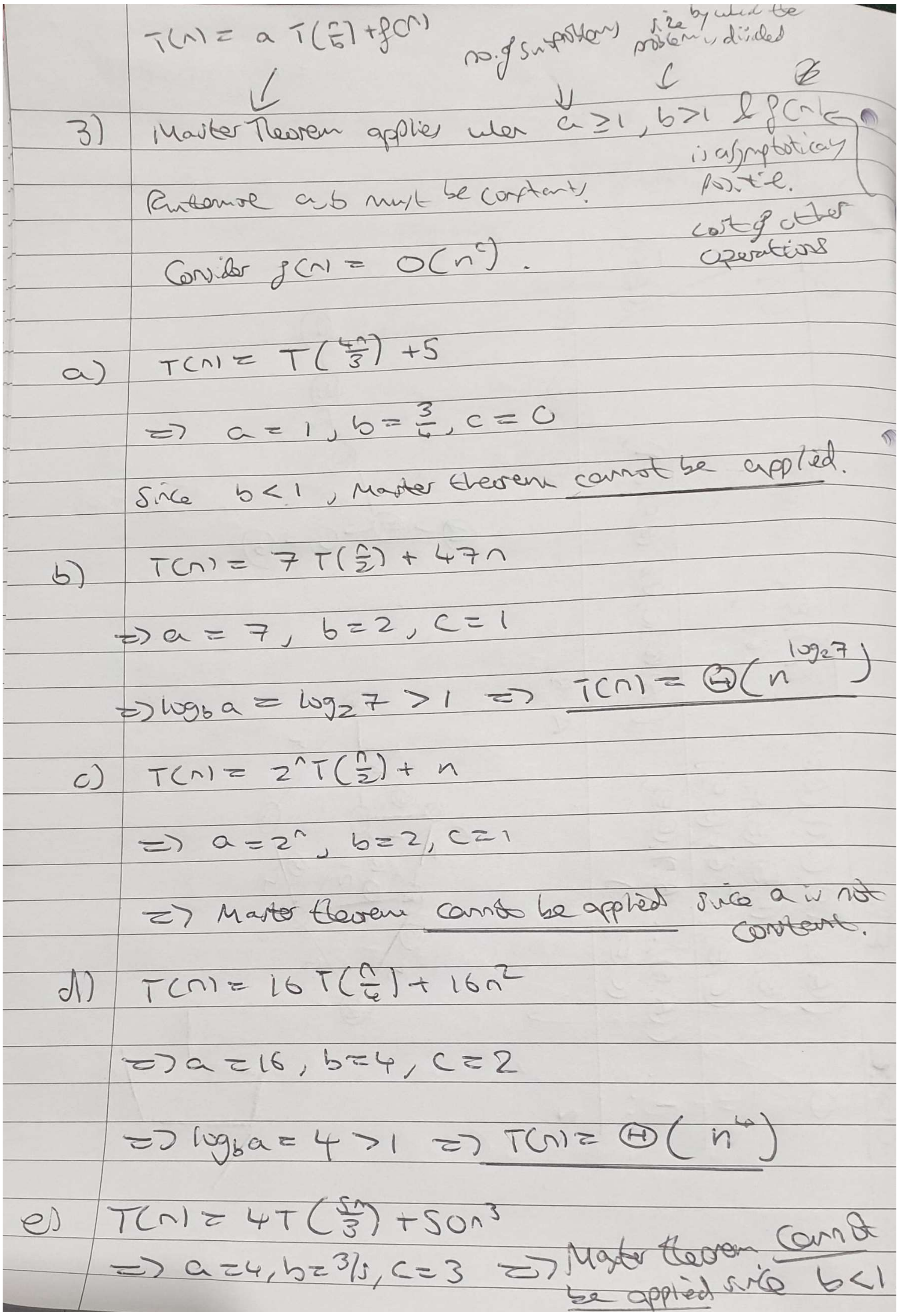
$\Rightarrow$ The statement must be false.



e) $n \leq n^2 + 6n + 143 \quad \forall n \geq 0,\ n \in \mathbb{N}_0$

$\Rightarrow n^2 + 6n + 143 \in \Omega(n)$. (True statement)

Question 3



$T(n) = a\,T(\frac{n}{b}) + f(n)$

no. of subproblems

size by which the problem is divided

3) Master Theorem applies when $a \geq 1$, $b > 1$ & $f(n)$ is asymptotically positive.

Furthermore a, b must be constants.

Consider $f(n) = O(n^c)$.

cost of other operations

a) $T(n) = T(\frac{4n}{3}) + 5$

$\Rightarrow a = 1,\ b = \frac{3}{4},\ c = 0$

Since $b < 1$, Master theorem cannot be applied.

b) $T(n) = 7\,T(\frac{n}{2}) + 47n$

$\Rightarrow a = 7,\ b = 2,\ c = 1$

$\Rightarrow \log_b a = \log_2 7 > 1 \Rightarrow T(n) = \Theta(n^{\log_2 7})$

c) $T(n) = 2^n T(\frac{n}{2}) + n$

$\Rightarrow a = 2^n,\ b = 2,\ c = 1$

$\Rightarrow$ Master theorem cannot be applied since a is not constant.

d) $T(n) = 16\,T(\frac{n}{4}) + 16n^2$

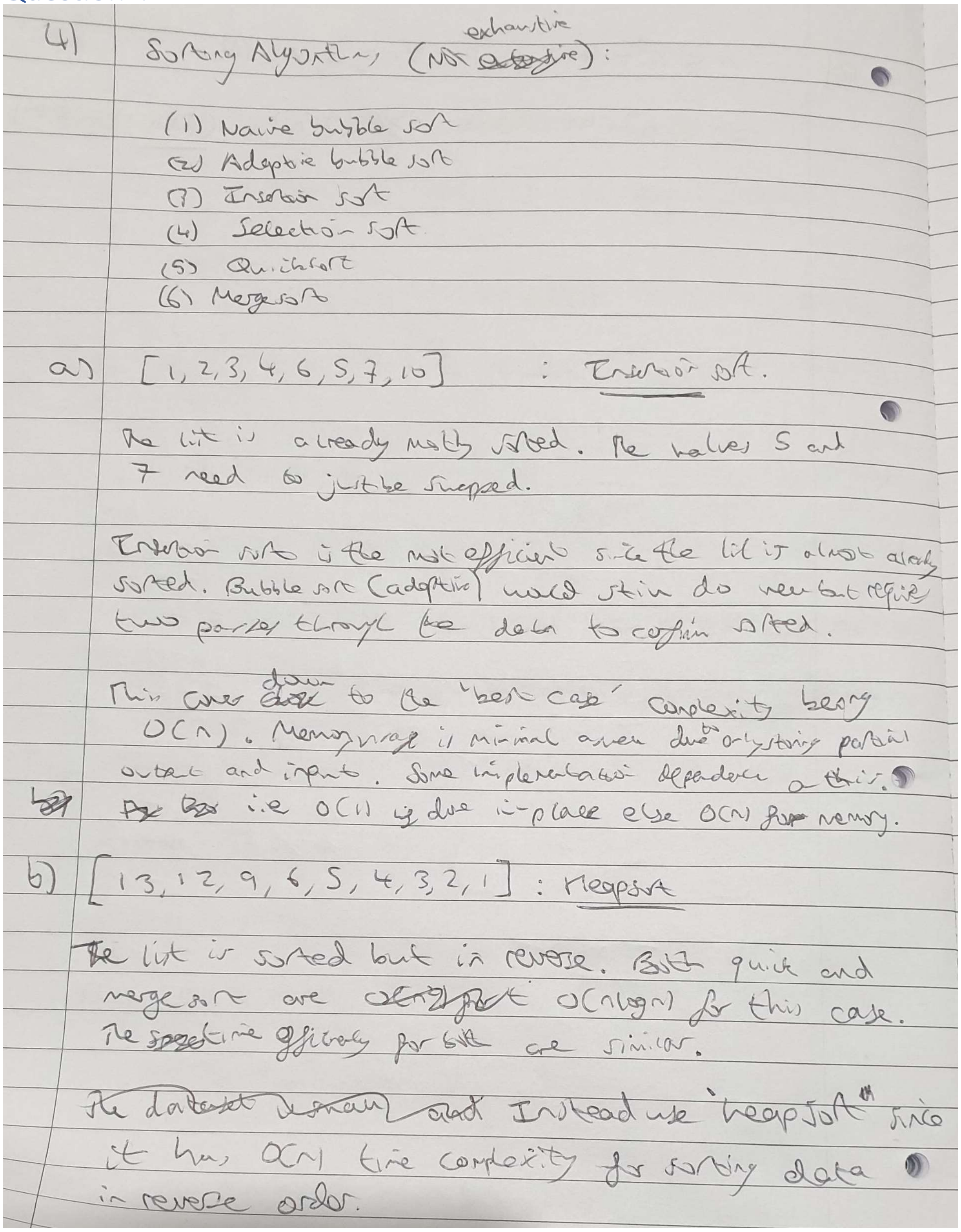
$\Rightarrow a = 16,\ b = 4,\ c = 2$

$\Rightarrow \log_b a = 4 > 1 \Rightarrow T(n) = \Theta(n^4)$

e) $T(n) = 4T(\frac{5n}{3}) + 50n^3$

$\Rightarrow a = 4, b = 3/5, c = 3 \Rightarrow$ Master theorem cannot be applied since $b < 1$

Question 4



4) Sorting Algorithms (Not ~~[illegible]~~ exhaustive):

(1) Naive bubble sort
(2) Adaptive bubble sort
(3) Insertion sort
(4) Selection sort
(5) Quicksort
(6) Mergesort

a) [1, 2, 3, 4, 6, 5, 7, 10] : Insertion sort.

The list is already mostly sorted. The values 5 and 7 need to just be swapped.

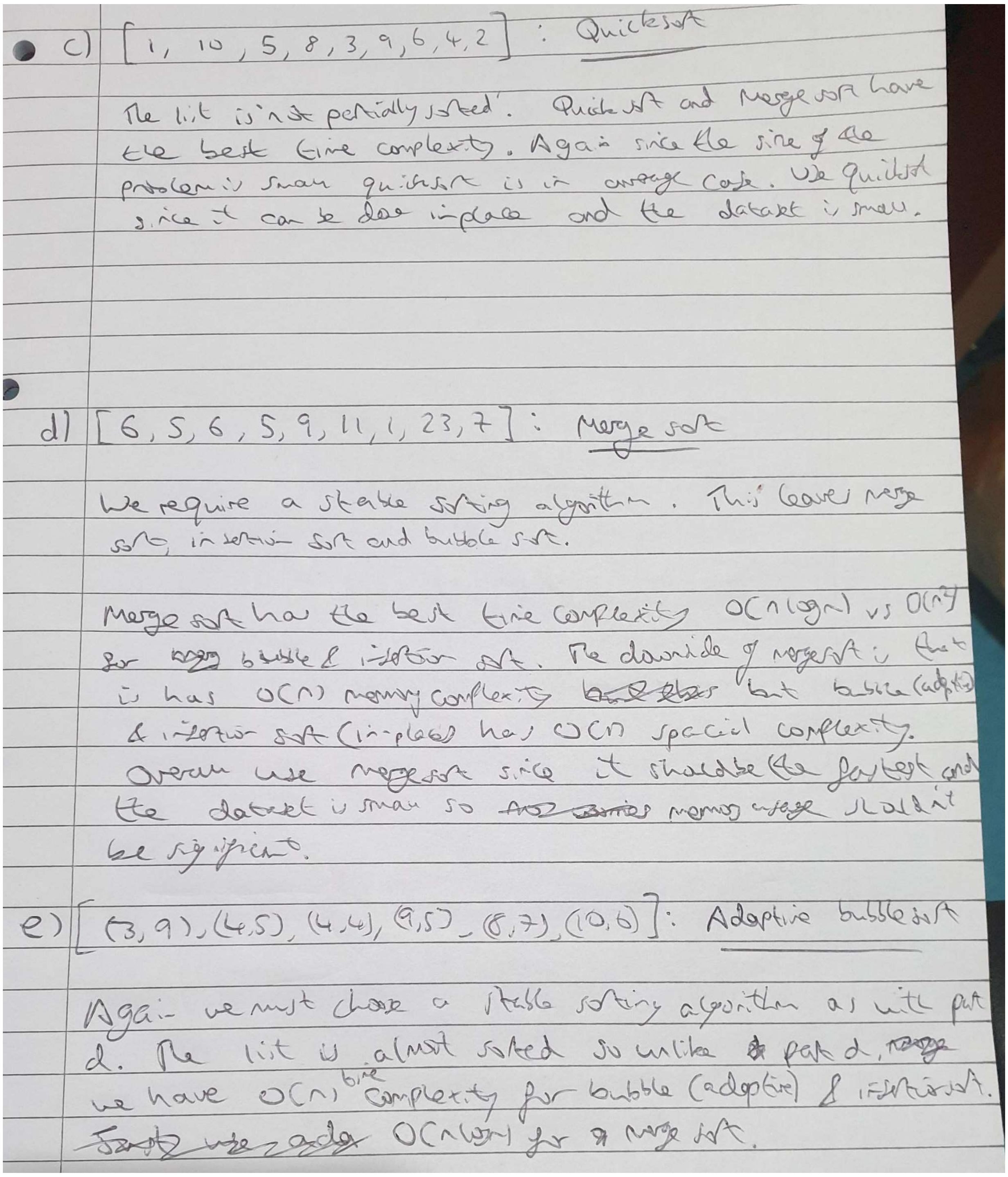
Insertion sort is the most efficient since the list is almost already sorted. Bubble sort (adaptive) would still do well but require two passes through the data to confirm sorted.

This comes ~~close~~ down to the 'best case' complexity being $O(n)$. Memory usage is minimal also due to only storing partial output and input. Some implementation dependence on this. ~~b)~~
~~As~~ ~~Bo~~ i.e $O(1)$ if done in-place else $O(n)$ for memory.

b) [13, 12, 9, 6, 5, 4, 3, 2, 1] : Heapsort

The list is sorted but in reverse. Both quick and merge sort are ~~$O(n^2)$ fast~~ $O(n\log n)$ for this case. The ~~speed~~ time efficiency for both are similar.

~~The dataset is small~~ and Instead use 'heapsort' since it has $O(n)$ time complexity for sorting data in reverse order.



c) [1, 10, 5, 8, 3, 9, 6, 4, 2] : Quicksort

The list is not partially sorted. Quicksort and Mergesort have the best time complexity. Again since the size of the problem is small quicksort is in average case. Use Quicksort since it can be done in place and the dataset is small.

d) [6, 5, 6, 5, 9, 11, 1, 23, 7] : Merge sort

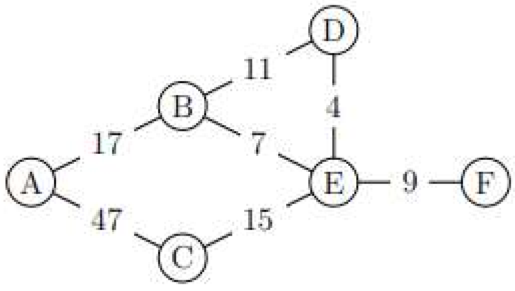
We require a stable sorting algorithm. This leaves merge sort, insertion sort and bubble sort.

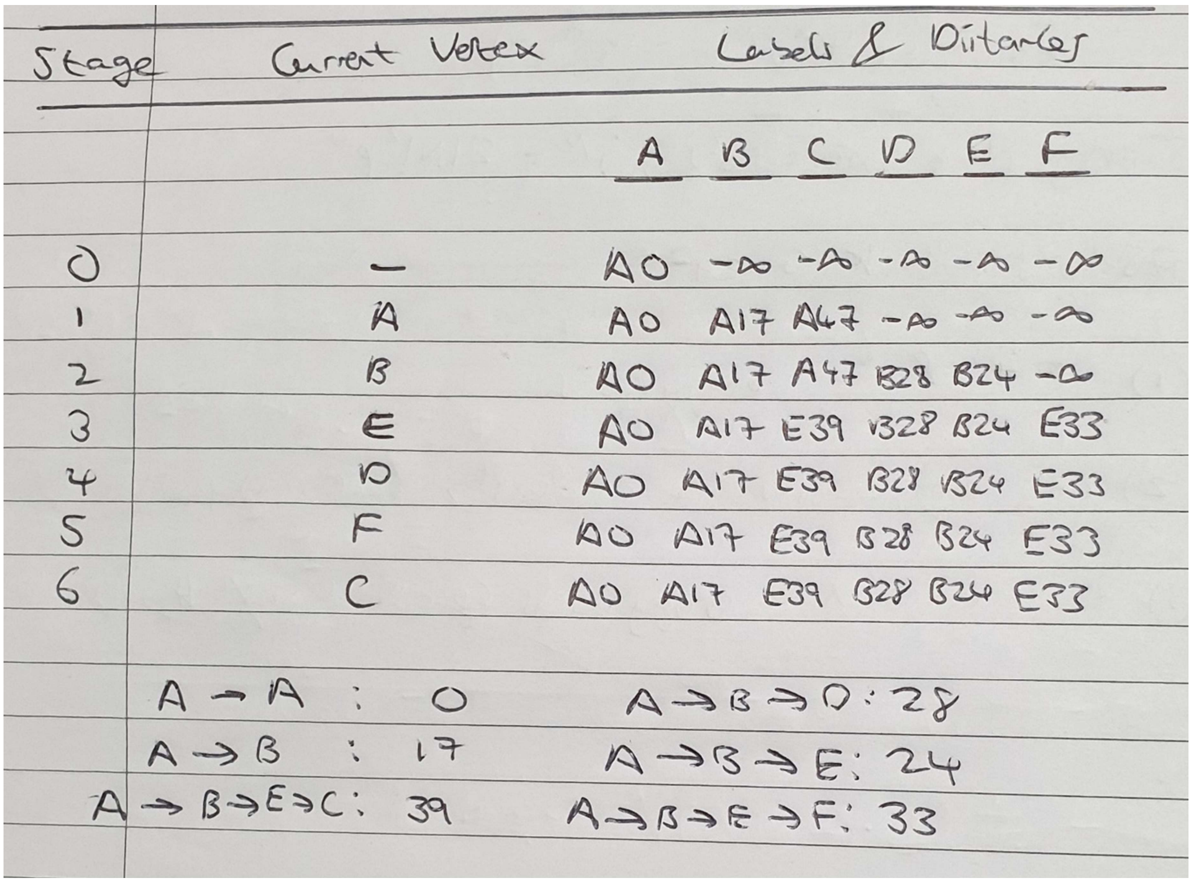
Merge sort has the best time complexity O(n log n) vs O(n²) for ~~merg~~ bubble & insertion sort. The downside of mergesort is that it has O(n) memory complexity ~~[illegible]~~ but bubble (adaptive) & insertion sort (in-place) has O(1) spacial complexity. Overall use mergesort since it should be the fastest and the dataset is small so ~~[illegible]~~ memory usage shouldn't be significant.

e) [(3,9), (4,5), (4,4), (9,5), (8,7), (10,6)] : Adaptive bubble sort

Again we must choose a stable sorting algorithm as with part d. The list is almost sorted so unlike ~~a~~ part d, ~~merge~~ we have O(n) time complexity for bubble (adaptive) & insertion sort. ~~[illegible]~~ O(n log n) for ~~a~~ merge sort.

Question 5



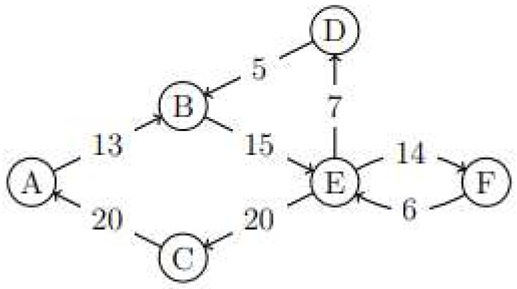


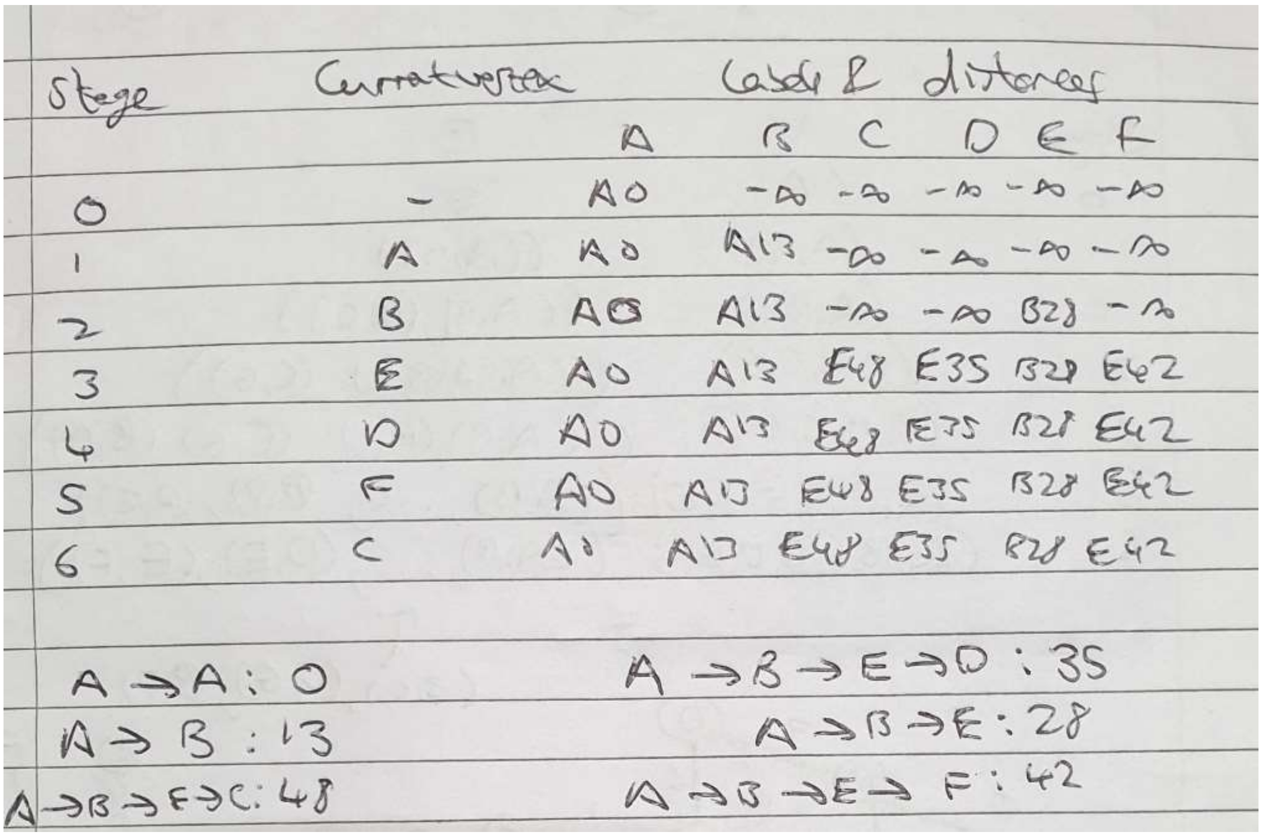
Stage	Current Vertex	Labels & Distances					
		A	B	C	D	E	F
0	–	A0	-∞	-∞	-∞	-∞	-∞
1	A	A0	A17	A47	-∞	-∞	-∞
2	B	A0	A17	A47	B28	B24	-∞
3	E	A0	A17	E39	B28	B24	E33
4	D	A0	A17	E39	B28	B24	E33
5	F	A0	A17	E39	B28	B24	E33
6	C	A0	A17	E39	B28	B24	E33

A → A : 0	A → B → D : 28
A → B : 17	A → B → E : 24
A → B → E → C : 39	A → B → E → F : 33

Dijkstra's algorithm allows us to find the shortest path for all nodes from the given vertex, in this case A. Starting from A we store the distances (/weights of edges) of each connecting node (A,B,C here) from A and record the node we connected from (A here). We then record that A has been considered. Next, we take the node (B) that hasn't been considered with the smallest value from A (since you cannot find a shorter path to it from A) and again calculate the distances to the adjacent nodes. If the cumulative distance from A to itself and then to the connecting node is less than the current recorded distance, we update it in the table. We also record the current node as the predecessor node for those updated. We then record this node as considered. The process is then repeated till every node is considered. The resulting final line in the table allows you to recover the optimal paths taken as shown. Note multiple paths with the same shortest distance are not recoverable. I.e. A->B->D and A->B->E->D.

Question 6





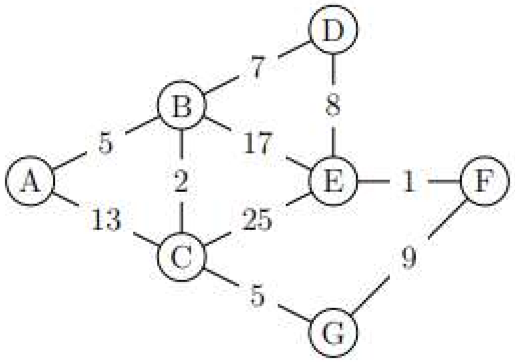
Stage	Currentvertex	Labels & distances					
		A	B	C	D	E	F
0	–	A0	-∞	-∞	-∞	-∞	-∞
1	A	A0	A13	-∞	-∞	-∞	-∞
2	B	A0	A13	-∞	-∞	B28	-∞
3	E	A0	A13	E48	E35	B28	E42
4	D	A0	A13	E48	E35	B28	E42
5	F	A0	A13	E48	E35	B28	E42
6	C	A0	A13	E48	E35	B28	E42

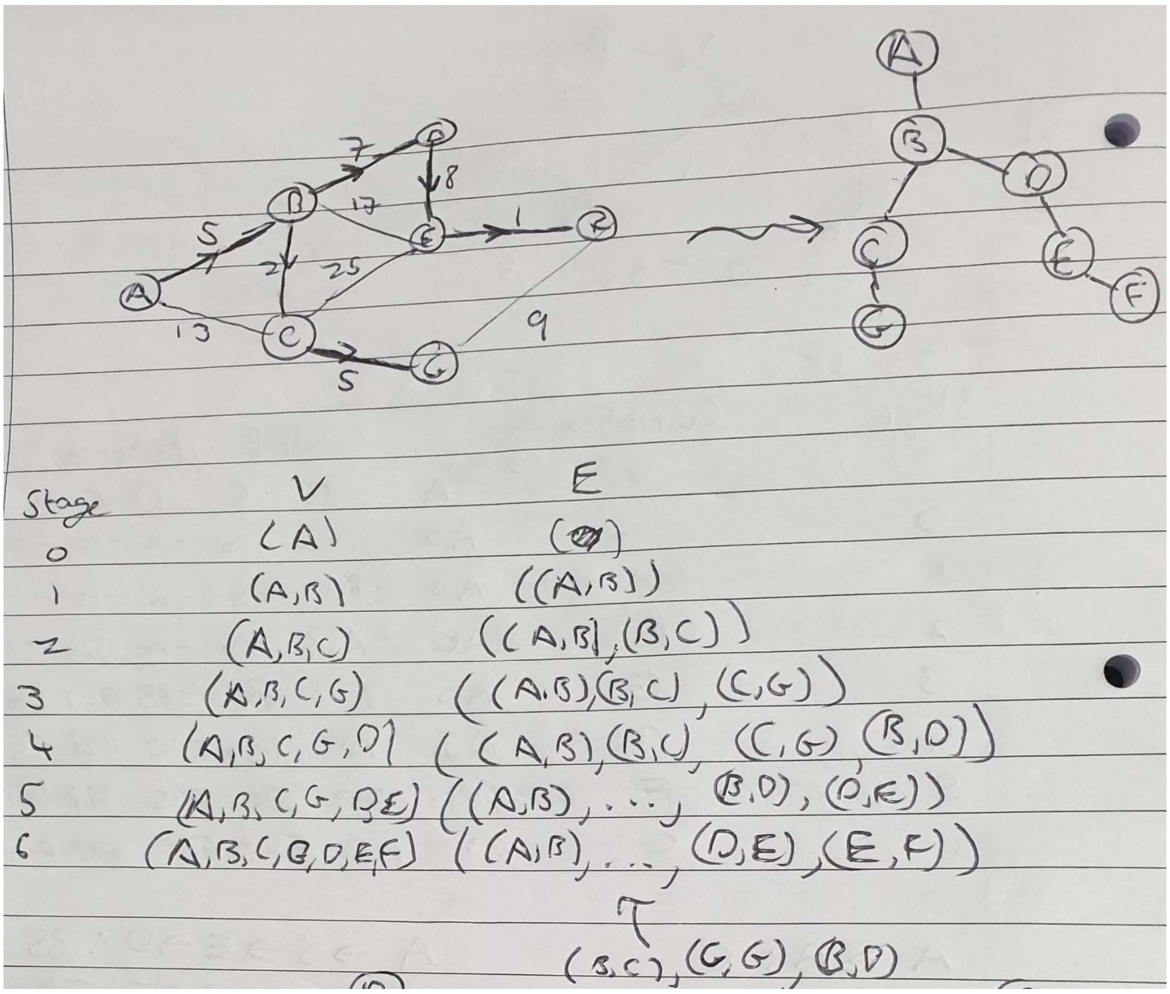
A → A : 0
A → B : 13
A → B → E → C : 48

A → B → E → D : 35
A → B → E : 28
A → B → E → F : 42

The procedure is the same as in question 5 except now the graph being directed needs to be considered. The only real difference in the procedure is that when you consider a node, you can only consider adjacent nodes with edges pointing in the correct direction. i.e B can only 'see' E but cannot see 'A' or 'D'. This means that when B is the 'current vertex', you must only consider E as a connecting node.

Question 7a (Prim’s Algorithm)




A
B
C
D
E
F
G
5
13
2
7
17
8
25
1
5
9

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,D),(D,E))
6	(A,B,C,G,D,E,F)	((A,B),...,(D,E),(E,F))

(B,C),(C,G),(B,D)

First consider Vertex A. Look at the edge weights with A and choose the minimum. This is (A,B).. Exclude this edge and record it within ‘E’. Record all vertices connected to edges in ‘E’ as ‘V’. Now consider all the remaining edges that are connected to A or B again with minimum weight but also does not create any loops with those within ‘E’. Record this edge in ‘E’. This is the edge (B,C). Now consider the vertices A,B,C and their remaining edges. Choose the edge with the minimum weight and no loops forming. Repeat this process till all vertices are connected via edges. This result is the minimum spanning tree. Note this tree is shown above.

Question 7b (Kruskal's Algorithm)

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

First record all the edges in one list, then the vertices in another. Next consider the edge with the smallest weight, this is (E,F) in this case. Record this in 'E', remove the edge from 'Edges' and consider the components. These being the disjoint nodes and the connected nodes (E,F). Next consider the edge with the smallest weight in 'Edges'. Remove this and repeat the same process. Complete this process until you arrive at a tree containing all nodes. As with Prim's algorithm, consider only edges that do not create loops and remove these from the 'Edges' list when they arise. Note this happens in stage 4 for (A,C).

Question 8

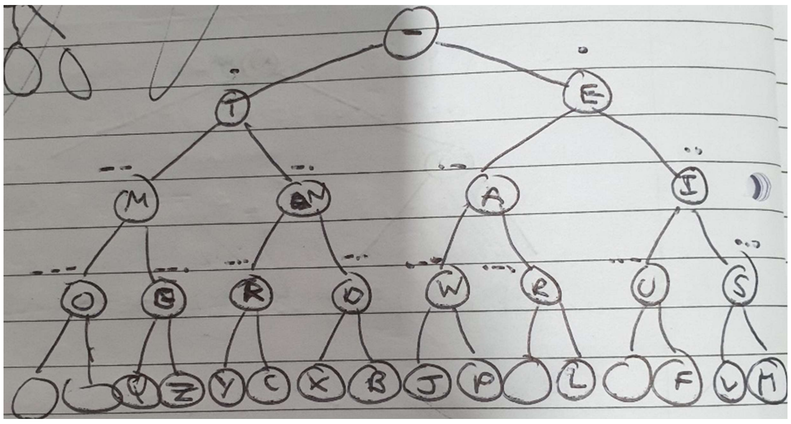
Algorithm 2 RIGHT-ROTATE(T, y)

```
Require: y.left 6 != T.nil, T.root.p == T.nil
 1: x = y.left                   // 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 9a

The problem is relatively straightforward. All that's required is a function that maps the morse code provided to the corresponding letters and finally combine it into a word.

The first challenge is deciding how to store the corresponding morse code to letter conversions. The simplest approach is a dictionary with (morse, letter) key-value pairs. I instead chose to use a binary search tree to store the letters of the alphabet. A 'left' traverse corresponds to a dash and 'right' traverse a dot. The tree is shown below.



The tree is implemented using the class 'BinaryTree'. The functionality is kept limited to what is required. Only insert() and search() functions are needed. In both cases the functions are recursively called to either find the location to insert a new letter or to search for one. The tree is used over the dictionary due to it's optimal space and time efficiencies, requiring only four traversals within the tree to retrieve any letter.

The morseDecode() function then simply searches the tree for the corresponding letters and merges them into a word to return.

Question 9b

The 2nd part is now more challenging. To account for the 'x' case in the tree, the search() function is updated. If the case arises that an 'x' is provided, then both a left and right traversal are performed. This returns a tuple containing both possible letters. Note, one missing dot/dash implies up to two possible letters. There could still be only one possible letter, for example '--.-' = Q however '.-.-' has no corresponding letter. The search() function is adjusted to account for this.

The searchWordList() function is used to compare a list of possible words to the 'dictionary.txt' provided and returns a list of words within that dictionary. The function uses a simple O(n) linear search to search the list for each word. A binary search algorithm was considered but this would require the dictionary to be sorted. The trade-off didn't seem to be sufficient to sort dictionary.txt given the number of morse codes letters to convert vs the size of the dictionary. In real-world application the dictionary should be sorted to give O(logn) complexity which is significantly better than O(n) for the size of the dictionary. Furthermore issues with testing on your end i.e. you calling the sorting function prior to testing made it unfeasible.

The findWords() function recursively finds every combination of letters with complexity O(2^n) from the list of tuples (two possible letters) for the word. The result is then stored in 'words'.

The morsePartialDecode() function then brings this all together, doing the same procedure as part a but now also opening dictionary.txt and returning the list of possible words from findWords() after being passed to searchWordList().

Note that the case that 'x' is not the first / only missing dot-dash hasn't been fully considered.

Question 9c

The maze is implemented using an undirected graph. The Maze class effectively being a class for a graph with some additionally functionality.

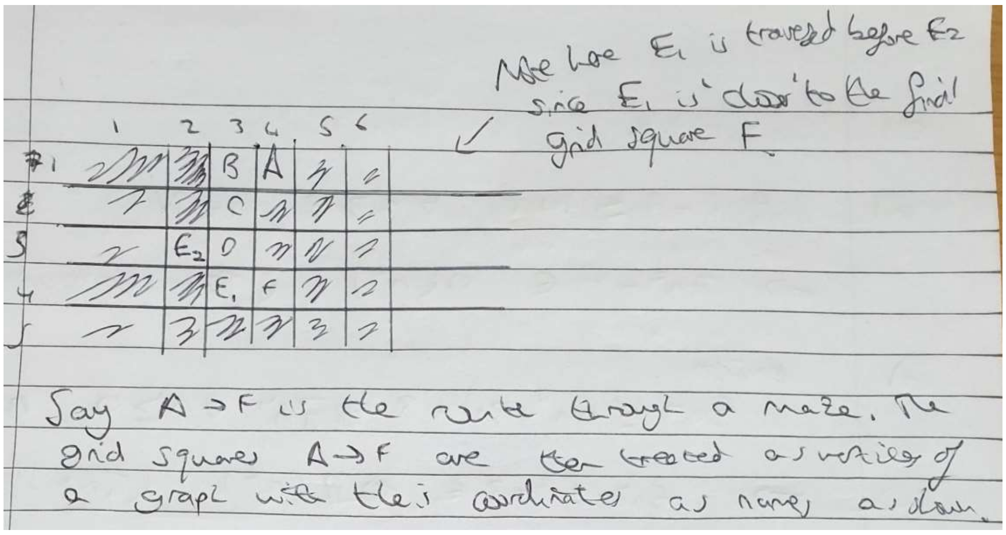
The addCoordinate() function adds a vertex to the graph if it's an 'empty space' else it is ignored since the program assumes everything else are walls by default. Prior to this the function increases the size of the grid appropriately to a new (M,N) based on the size of x,y passed.

There are some assumptions made about the maze and its solution:

1. There are no 'pockets' i.e regions of the maze completely sealed off from the rest. i.e. a connected graph. (Giving two coordinates within the same 'pocket' would still function as expected but two vertices in different 'pockets' may not)
2. If there are paths with greater than 1 width, these are not handled optimally. (i.e. think about a 3x3 region of 'empty space' with additional paths leading away/entering it.
3. The program heuristically finds a 'good' solution rather than necessarily the optimal or every solution.
4. The local optimal solution is one that approaches the endpoint rather than away from it.
5. Loops are handled appropriately.
6. The maze is considered a 'sparse' graph with each vertex only connecting to upto 4 other vertices << V (no. of vertices) most likely. Hence adjacency lists are used over matrices to store the edges.
7. The maze isn't so large that DFS leads to a stack overflow error.
8. If the distance between start and endpoints is less than half the grid size (diagonal) then use BFS otherwise use heuristic DFS.

The getFavouriteNeighbour() function is used to give the 'first choice' for the neighbouring square to go to next. This is the neighbour that is the closest to the endpoint.

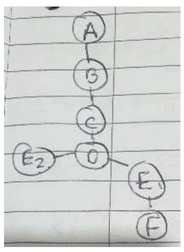
The dfs() or depth first search function (DFS), recursively traverses the graph using getFavouriteNeighbour() to determine which path to traverse first. It traverses then backtracks along each path until a solution is found. An example is shown below

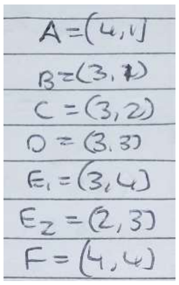


Note here E_1 is traversed before E_2 since E_1 is 'closer' to the final grid square F.

Say A → F is the route through a maze. The grid squares A → F are then treated as vertices of a graph with their coordinates as names, as shown.

1
2
3
4
5
6
B
A
C
E_2
D
E_1
F



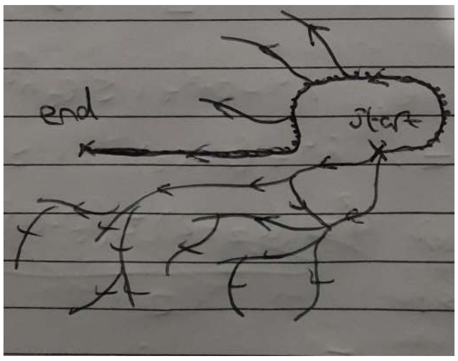


A = (4,1)
B = (3,1)
C = (3,2)
D = (3,3)
E_1 = (3,4)
E_2 = (2,3)
F = (4,4)

A heuristic depth first search is used over say a breadth first search (BFS) since large mazes => large n nodes may take non-trivial times to solve for a solution. Instead the heuristic DFS approach will stop once it finds a solution which on average is faster so long as the heuristic is viable. The time to solve for DFS and BFS can vary massively for the same maze.

Note the BFS will always find the shortest path.

If the start and end nodes are 'close' to one another vs the size of the maze (M,N) then BFS search is likely faster. BFS is then used instead of DFS. For example if the correct path initially goes away from the endpoint, the heuristic DFS will not traverse the path till it has traversed every path towards the endpoint as shown below.



Note V < E < 4V where V is the number of vertices and E the number of edges. We can then say for both BFS and DFS has complexity O(V+E) = O(E).