Answer to the question no: 1

Given that me & my friend Benji want to minimize the total travel time to meet, we need to find the area that minimizes the sum of distance from both of our houses.

Pseudocode:

```
def optimum meet (G, source):
    distance, parent 50 = Dijkstra (G, source)
    min_distance = infinity
    meeting_point = -1
    for each vertex v in G:
        total_distance = distance [source] + distance [v]
        if total_distance ~~= distance~~ < min_distance:
            min_distance = total_distance
            meeting_point = v
    return meeting_point
```

Here, we apply dijktras algorithm to find shortest distances from my house (vertex 1) to all other vertex. Then we iterate over all vertices to calculate the total distance ~~& the meeting~~ from my house to each vertex & from Benji's house (vertex 50) to that vertex. We update the minimum distance & the meeting point accordingly. Finally, we return the vertex that minimizes the total travel time. It will have time complexity of $O(E.\log_2(V))$. where, E is the number of edges & v is the number of vertices.

Answer to the question no: 2

Here, from the question, considering one-way roads & Benji wanting KFC, we need to find the KFC outlet that minimizes my total travel time from my house to KFC & then to Benjis house.

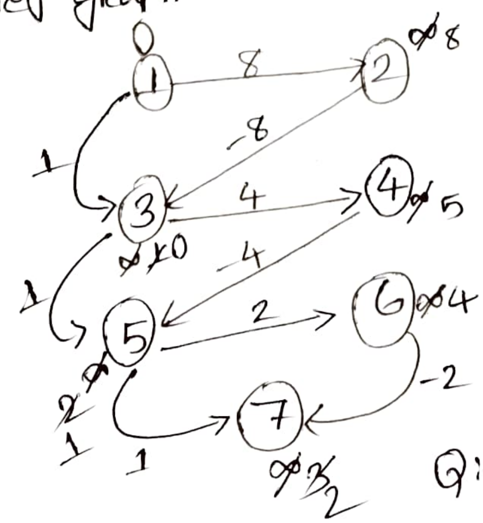
Pseudocode:

```
def OptimumKFC(G, a, K):
    min_distance = infinity
    optimal_kfc = -1
    foreach KFC_location x in K:
        distancefromsource, parents = Dijkstra(G, a)
        totaldistance = distancefromsource[x] + distancefromsource[50]
        if totaldistance < mindistance:
            mindistance = total distance
            optimal_kfc = x
    return optimal_KFC
```

Here, we iterate over KFC location & apply dijktra's algorithm to find the shortest path from my house (vertex 1) to all other vertex. Then, we calculate the total distance from my house to the current KFC location & from there to Benji's house (vertex 50). We update the minimum distance & the optimal KFC location accordingly. Finally, we return the KFC location that minimizes the total travel time. It will have time complexity of $O(E \cdot \log^2(V))$.

Answer to the question no: 3

Directed weighted graph:-



Nodes	Cost
1 →	0
2 →	~~∞~~ 8
3 →	~~∞~~ ~~1~~ 0
4 →	~~∞~~ 5
5 →	~~∞~~ ~~2~~ 1
6 →	~~∞~~ 4
7 →	~~∞~~ ~~3~~ 2

Q: ~~1~~, ~~2~~, ~~3~~, ~~4~~, ~~5~~, ~~6~~, ~~7~~

Initially,

(a)

$d = [\infty, 0, \infty, \infty, \infty, \infty, \infty]$, $P_i = [Nil, Nil, Nil, Nil, Nil, Nil, Nil]$, $S = [\,]$

(1) Iteration: 1

Choose vertex with minimum d value: 2

Update neighbors: - vertex 3 : $d[3] = \min(\infty, 0-8) = -8$, $P_i[3] = 2$

Queue = [3]

(2) Iteration: 2

Choose vertex with minimum d value: 3

Update neighbors: - vertex 4 : $d[4] = \min(\infty, -8+4) = -4$, $Pi[4] = 3$

vertex 5 : $d[5] = \min(\infty, -8+1) = -7$, $Pi[5] = 3$

Queue = [4, 5]

(3) Iteration: 3

Choose vertex with minimum d value: 4

Update neighbors: - vertex 5 : $d[5] = \min(-7, -4-4) = -8$, $Pi[5] = 4$

Queue = [5]

(4) Iteration: 4

Choose vertex with minimum d value: 5

Update neighbors: - vertex 6 : $d[6] = \min(\infty, -8+2) = -6$, $Pi[6] = 5$

Queue = [6]

Choose vertex with minimum d value: 6

(5) Update neighbors: - vertex 7 : $d[7] = \min(\infty, -6-2) = -8$, $Pi[7] = 6$

Queue = [7]

(6) Iteration 6:

Choose the vertex with minimum d value: 7, Queue = [ ]

Thus,

Shortest Path costs: $d = [\infty, 0, -8, -4, -8, -6, -8]$

$P_i = [Nil, Nil, 2, 3, 4, 5, 6]$

(b)

The algorithm works by iteratively selecting the vertex with the smallest tentative distance & relating it's outgoing edges. In this case, the algorithm correctly find the shortest path costs for some vertices because it's able to propagate the correct information about the shortest path through the graph, with updating distances. But it could have not work for the negative values. As, for negative values digstra's algorithm can give wrong answers. (b) It will have $O(|V|^2 + |E| \times \log |V|)$ time complexity.

(c)

The modified Dijkstra's algorithm can still find the correct shortest path costs in the given graph. While it may not terminate as early as the traditional version, it will eventually explore all paths and update distances as necessary. The modification essentially mimics the behavior of the original algorithm, ensuring that the correct shortest path costs are eventually determind for all vertices.

(e)

The worst-case time complexity of this modified algorithm will be $O(|V|^2 + |E|)$. As, each vertex could be inserted into the queue once for each edge, leading to $O(|V| \times |E|)$. Additionally, for each vertex, the algorithm needs to find the minimum distance in the priority queue, which takes $O(|V|)$ time. Therefore, the overall time complexity will be,

$O(|V|^2 + |E|)$

Answer to the question number: 4

Let, G be an n-vertex graph, Let, G's MST be MST(G). Let, MST(A) represent A's MST. G is split into A & B by the algorithm. Then the method determines MST(A) & MST(B) using recursion, The program then finds which edge between A & B is the lightest and that edge is what constitutes the MST of G.

Assuming, A is a subset of G's vertices,

Let, G represent graph & A be the collection of vertices. B is the set of vertices.

$MST(A) = \{(1,2)\}$

$MST(B) = \{(3,4)\}$

The edge (1,4) that connects A & B is the thinnest edge. According to the assertion, MST (G) = MST(A) U MST(B) "the lightest edge that connects A & B" $= \{(1,2)\} \cup \{(3,4)\} \cup \{(1,4)\}$
$= \{(1,2), (1,4), (3,4)\}$

The MST of G, however is the edge, (2,3)

As a result, the claim is false and the method mentioned in the problem description cannot always be relied upon to provide the right solution.

The edge (2,3) ~~with~~ which weights 3 is the MST of the graph. The problem's method that is given will locate edge (1,4) which weigts 1.

The algorithm does not promise to always give right response. This is due to the algorithm's restriction to the graphs lightest edge connecting two halves. In other cases, a heavier ~~ga~~ edge that connects the two parts & creates a legitimate MST may also exist.

[Proved]

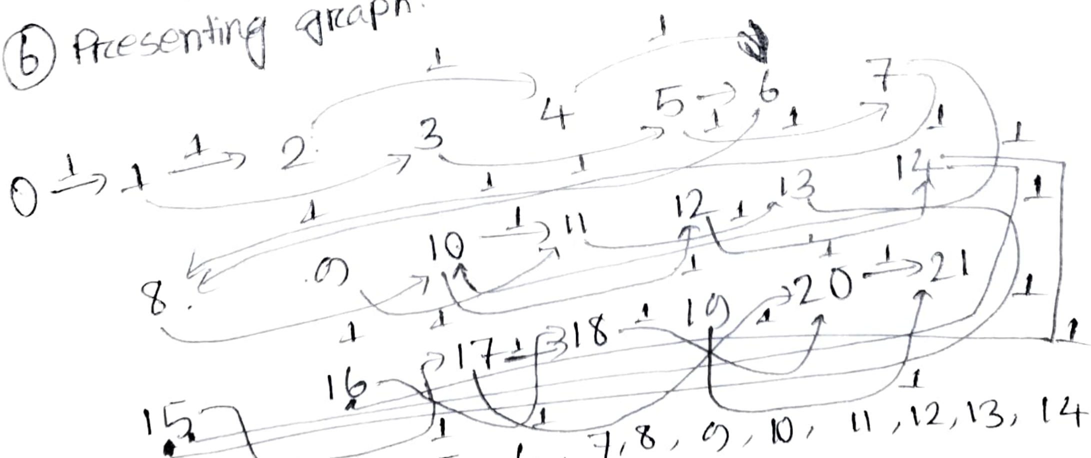
Answer to the question no: 5

ⓐ The Dijkstra's algorithm might be relavant here. As, the algorithm will find the ~~f~~ shortest path from the source to all other vertex, in a weighted graph. Here, the edges between vertices represent the different banknotes and the weights on these edges represent the number of bank notes needed to make the desired amount.

Pseudocode:

def coinchange (notes, target):
- ↳ Creating a graph to connect amounts using notes.
- ↳ Use dijkstra's algorithm to find the shortest path, 0 to target.
- ↳ If no path exits, return "Impossible"
- ↳ otherwise, backtrack from target to 0 to determine the notes used.
- ↳ Return the total notes & sequence used.

ⓑ Presenting graph:



Nodes: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21

Nodes: 15, 16, 17, 18, 19, 20, 21

Example: [1, 2, 5, 10] notes, make 21

vertices: 0 to 21 (inclusive).

Edeges:

From	0	to	1	(weight	1	, using banknote of value	1)
"	1	"	2	("	1	, " " " "	1)
"	1	"	3	"	1	, " " " "	2
"	2	"	4	"	1	, " " " "	2
"	3	"	5	"	1	, " " " "	2
"	4	"	6	"	1	, " " " "	2
"	5	"	6	"	1	, " " " "	1
"	5	"	7	"	1	, " " " "	2
"	6	"	8	"	1	, " " " "	2
"	7	"	8	"	1	, " " " "	1
"	7	"	10	"	1	, " " " "	5
"	8	"	10	"	1	, " " " "	2
"	9	"	11	"	1	, " " " "	2
"	10	"	11	"	1	, " " " "	1
"	10	"	12	"	1	, " " " "	2
"	11	"	13	"	1	, " " " "	2
"	14	"	14	"	1	, " " " "	2
"	13	"	15	"	1	, " " " "	2
"	14	"	15	"	1	, " " " "	1
"	14	"	16	"	1	, " " " "	2
"	15	"	17	"	1	, " " " "	2
"	16	"	18	"	1	, " " " "	2
"	17	"	18	"	1	, " " " "	1
"	17	"	20	"	1	, " " " "	5
"	18	"	20	"	1	, " " " "	2
"	19	"	21	"	1	, " " " "	2
"	20	"	21	"	1	, " " " "	1

This will give us amount 21 for, notes [1, 2, 5, 10]

Answer: 3 notes needed : 10, 10, 1