Part A

Q1. 1. T(G) only has one element. Since there is only one minimum spanning tree for an undirected graph with injective weight function.

2. We can use Kruskal's algorithm to form a minimum spanning tree. In algorithm, all edges are stored. Every time, the current minimum weight edge is selected. If adding the edge to the loop causes the loop, drop it. Otherwise, adding the selected edge to the minimum spanning tree. By looping such process, a minimum spanning tree is formed. Since the edges are with distinct weight, there is only one sorted outcome for edges. No matter how many times the algorithm is applied. It always returns a same minimum spanning tree. So, there is only one minimum spanning tree for an undirected graph with injective weight function.

Q2. 1. Pseudo code:

Reference: https://www.geeksforgeeks.org/minimum-insertions-to-form-a-palindrome-dp-28/

```
1.int find_minimum (String s):
2.      int n = length of s;
3.      create a table with size n*n, and fill it with 0
4.      int l, h, gap = 0;
5.      for gap in range (1, n):
6.              l = 0;
7.              for h in range (gap, n):
8.                      if s[l] = s[h]:
9.                              table [l] [h] = table [l+1] [h-1];
10.                     else:
11.                             table [l] [h] = min (table [l] [h-1], table [l+1] [h] + 1);
12.                     l++;
13.     return table [0] [n-1];
```

2. The above algorithm is a dynamic programming. For using the dynamic programming, we need make sure the two requirements that 'optimal substructure' and 'overlapping sub-problem' are always hold.

In algorithm, the optimal number of insertions for string s is based on the optimal insertions for string (s-1) (substring (0.. s-1) and substring (1..s)). The string (s-1)'s optimal insertion is based on string (s-2) and so on .. So, the requirement of 'optimal substructure' is hold.

When calculating the insertion for a substring (0..s-1) and (1..s), there is always some overlap calculation. For example, the string 'abcde'. The substring is 'abcd' and 'bcde'. The substrings of 'abcd' are 'abc' and 'bcd'. The substrings of 'bcde' are 'bcd' and 'cde'. The substring 'bcd' is overlapped. If continue to split the string, there are more overlaps. So, the requirement of 'overlapping sub-problem' is hold.

In algorithm, it divides the problem into subproblems by checking the value of s[0] and s[n]. If the values of s[0] and s[n] are same, it means there is no need for inserting an element to make the s[0] and s[n] to be same and problem is convert into subproblem s[1..n-1]. For example, 'abcdea' => 'bcde'. If the values of s[0] and s[n] are different, there are two ways to insert an element to make s[0] and s[n] are same. One is to insert the s[n]'s value at beginning of string. Another is to insert the s[0]'s value at the end of string. For example, 'abc' => 'cabc' or 'abca'. After one insertion, the problem is convert into subproblem minimum (s[0..n-1] or s[1..n]). The algorithm can always find the optimal solution.

3. Line 2~4 all take the constant time. Line5 loops n times. Inside of each loop, line 7 maximum loops n times, other lines all take constant time => maximum n^2 => O(n^2). The time complexity is O(n^2).

4. The memory complexity is O(n^2). Line 3 creates a table with size n*n which takes O(n^2). The other lines all take a constant memory complexity.

Q3. 1. When Huffman tree is a balanced binary tree, the efficiency of Huffman code is no better than fixed-length encoding. When each character has same frequency, the Huffman tree is balanced with height $\lceil \log(n) \rceil$ (n types of character). Each character is a leaf with code length $\lceil \log(n) \rceil$. Assume each character has frequency f, the total bits are f*n*$\lceil \log(n) \rceil$. For n types of character, the fixed code has the length $\lceil \log(n) \rceil$. The total bits for fixed code are f*n*$\lceil \log(n) \rceil$. The Huffman tree has the same performance with the length code which is no better than fixed length code.

2. The Mr T is incorrect. The AVL tree has a property that heights of left and right subtrees cannot be more than one for all nodes. So, the minimum height of AVL tree with n elements are $\lfloor \log(n) \rfloor$ which is the optimal performance. The optimal performance of AVL Huffman tree is the worst case of full binary Huffman tree (see question 1). So, Mr T is incorrect.

Part B

Q1. 1. Algorithm Reference: https://www.geeksforgeeks.org/combinations-from-n-arrays-picking-one-element-from-each-array/

Pseudo code:

```
1.b = array with size of total furniture.
2.int find_maximum (array a, int total_money):
3.      int n = length (a);
4.      selection = array with size (size of total furniture) and initialize to 0;
5.      int sum = 0;
6.      int maximum = 0;
7.      while (true):
8.              for i in range(n):
9.                      if (sum + array[i][selection[i]] < total_money):
10.                             sum = sum + array[i][selection[i]];
11.                     else if (sum + array[i][selection]) == total_money and I == n -1):
12.                             sum = sum + array[i][selection[i]];
13.                     else
14.                             sum = -1;
15.                             break;
16.             maximum = max (maximum, sum);
17.             if maximum = max:
18.                     for j in range(selection):
19.                             b[i] = selection[i];
```

```
20.             int next = n -1;
21.             while (next >= 0 and selection[next] + 1 >= length (a [next])):
22.                     next --;
23.             if (next < 0):
24.                     break;
25.             for i in range (next + 1, n):
26                      selection[i] = 0;
27.  return sum;
```

Line 9~15 is checking whether the i-th selection could let sum over the total money. In line 9, it considers the condition that adding the i-th selection, the sum is still lower than total money. In line 10, it considers the condition that adding the i-th selection, the sum is equal to the total money and it is the selection for last model. In other condition, the sum is over than total money which is counted as sum = -1. Since the int maximum is used to store the maximum value, which is initialized with 0, it always gives up the selection over the total money. Once the sum does not meet the total money, break the calculation loop. It voids the useless further calculation. Line 17~20 is updating the optimal selection if any.

2. In algorithm, the int 'next' is incremented from 0 to total types of furniture. Once the selected model for furniture type 'next' is incremented to the total model numbers of furniture (which starts from 0). The line 21's while loop condition is satisfied, the 'next' value increase 1.

So, the algorithm total while loop T times for the 'next' value. For each 'next' value, it loops K_i times. => The time complexity is O($\prod_0^T K_i$) where K_i is the i-th model types.

Q2. 1. Dynamic programming.

2. Overlapping sub-problem:

In question, we are required to find the maximum spend money when given a budget. Say, there is total n types of furniture we need to buy. The maximum spending money for n types of furniture can be solved by finding the maximum spend money for (n-1) types of furniture and so on. The problem can be solved by solving the subproblem. In this case, it satisfies the 'overlapping sub-problem'.

Optimal substructure

When we find the maximum spend money for n types of furniture, it should be (the price spend on type n) + (maximum value spend on 0..n-1 types). There are a lot of same subproblem solutions used again and again. For example, the budget is 10. There are three types of furniture shown below.

10	3	
2	1	2
2	3	4
2	5	4

Consider two types of furniture are selected:

1. If model with price '3' is selected as third type and model with price '5' is selected as second type, the maximum spend money should be (8 + maximum money spends on type 1 when budget is 2).
2. If model with price '4' is selected as third type and model with price '4' is selected as second type, the maximum spend money should be (8 + maximum money spends on type 1 when budget is 2).

The subproblem 'maximum money spends on type 1 when budget is 2' is used twice. In this case, it satisfies the 'optimal substructure'.

Q3. 1. Pseudo code:

```
// total_types refers to the total type of furniture. Total_money refers to the budget.
1.  create a table 'table' with size [total_money][total_types];
2.  create a table 'models' with size [total_furniture][types_of_models];
3.  create a table 'selected_model' with size [total_money][total_types];
4.  for i in range (total_types):
5.      for j in range (total_money+1):
6.              int value = 0;
7.              for m in range (total_model):    //total_model refers the number of
    models for furniture i
8.                      if (j > models[i][m] and j > 0):
9.                          int remaining = total_money – models[i][m];
10.                         value = max (value, models[i][m] + table[remaining][i-1];
11.                     else if (j > models[i][m] and j = 0):
12.                         value = max (value, models[i][m]);
13.                     else if (j = models[i][m]):
14.                         value = models[i][m];
15.             update the 'value' to the 'table';
16.             store the selected model index to the 'selected_mode';
17. // find the selection by checking the 'table'
18. create an array 'selection' with size total_types
19. int previous = 0;
20. for i in range(total_types, 1):
21.     if i = total_types:
22.             selection [i] = table[total_money][total_types-1];
23.             previous = table [total_money-model_price][total_types-1];
24.     else:
25.             selection [i] = selected_table[previous][i]
26.             previous = previous – selected_model_price
```

Line 4~16 is using three loops to update the maximum spend money for given money for certain types of furniture. Line 18~26 is using the data stored in tables to find the optimal selection. It follows a rule: the maximum money spent for n types of furniture = (model price + maximum money spends on n-1 types of furniture given the budget total_money-model price).

2. Proof of correctness:

It always holds that the maximum money spend on first furniture type is the price of model (if the total money is affordable) or 0 (if the total money is not affordable) which is stored in table.

When calculates the maximum money spend on first two furniture types, it is (the money spends for second type furniture + remaining money spends on first type). Since the maximum money spends on first type furniture under different budget is stored in table, we can call the corresponding value from table directly. => Maximum spend on first two furniture is guaranteed to be true.

By applying the same process to calculate the first three types of furniture, first four, first five... until the last type of furniture. It always holds the maximum spend money.

3. For creating table part: Line 4 loops T times. Line 5 loops M times. Line 7 loops K times. Other lines are constant time which could be ignored. => O(T*M*K).

For finding selection part, line 20 loops T times. Other lines are constant time which could be ignored. => O(T).

=> The time complexity = O(T*M*K) + O(T)

= O(T*M*K)

4. Eight tests are created. The parameters are shown in below table.

	Test1	Test2	Test3	Test4	Test5	Test6	Test7
T	2	6	10	20	40	60	80
M	500	500	1200	100	100	3321	5500
Avg_K	3	3	6	4	4	8	8

The complete search uses 184839ms CPU time to finish the test. So, in here, assume the complete search only can do the calculation up to test4 size. The test results are in below table (all for CPU time ms):

	Test1	Test2	Test3	Test4	Test5	Test6	Test7
dynamic	0.457	0.661	2.301	0.27	0.581	18.516	33.054
complete	0.03	0.105	1267.2	184839	-	-	-

Since each test has different T,M and Avg_K, it is not necessary to draw in diagram for showing the linear relation (based on previous time complexity driven and real data). So, we can do some comparisons to analysis.

It is obvious that the complete search has really good performance on test1 and test2 compared to dynamic programming. However, from the test3, the performance become really worse. By previous analysis, the time complexity for complete search is O(K^T). By increasing the T value, the running time is 'crazy' growth which is same with time complexity.
The dynamic programming's performance is not good as complete search in test1 and test2. It makes sense since the dynamic programming's time complexity is O(T*K*M). It creates table which takes longer time. But for larger test case, more 'overlapping subproblems' are solved by using the value stored in table. The dynamic solves a lot of time and has excellent performance. In test3, the M is 1200 which is about 10 times of test4. Compared to their performances, the test3 takes around 10 times of test4's CPU time. It suits the driven time complexity. The test2's T is three times of tests1's T. As a result, the running time of test2 is around three times of test1. It is similar for task4 and task5. The test5's T value is two times of test4's T value. As a result, the running time of test5 is around two times of test5's value. When T,M,Avg_M value all increases, the total time increases (see the test5 with test6 and test7).

Overall, the theoretical result is same with experiment result.