**Xiaobin Chen**

**Part 1:**

Step 1

1. The observed reliability is 90.94%.
2. The execution time of the brute force implementation is 27.209ms.

Step 2

1. The observed reliability is 72.55%.
2. The execution time of the brute force implementation is 0.0203ms.

Step 3

1. Using a Recovery Block of these two implementations, from above we know the reliability of SubsetSum is 90.94%, the reliability of SubsetSumDP is 72.55%, the theoretical reliability is:



1. If having SubsetSum go first, we know the approximate execution time of SubsetSum is 28.5289ms and the execution time of SubsetSumDP is 0.0203ms, so :



1. If having SubsetSumDP go first , the approximate executing time is:



Step 4

1. From Step3, we know the Recovery Block should start with SubsetSumDP module,
2. The reliability of Recovery Block is 94.91%
3. The Execution time of Recovery Block is 1.532ms

Step 5

1. For Retry Block, if a module does not work, it tries again with a different representation of the data with the same implementation of the module. So for brute force solution, likelihood of failure F = (1 – 0.9094) = 0.0906, in order to achieve 95% reliability, the times of execution N should satisfied:



So the choose N = 2, and the approximate theoretical execution time should be:



1. If using the dynamic programming solution, likelihood of failure F = (1 – 0.7255) = 0.2745, in order to achieve 95% reliability, the times of execution N should satisfied:



So the choose N = 3, with T = 0.0203ms, the approximate theoretical execution time should be:



Step 6

1. Form step 5, we know using SubsetSumDP to implement Retry Block is more efficient.
2. For the transformation function, I wrote a Fisher–Yates shuffle function which time complexity is o(n). I do not shuffle the original array directly, but shuffle the index array instead (e.g. {0, 1, 2, 3, 4} -> {3, 2, 1, 0, 4}). Then use the new index array to generate new input. Once result is generated, because the current index array contains the mapping information to the original index array, so we can use index array together with result Boolean array to reconstruct the output in time complexity is o(n) .
3. The reliability of Recovery Block is 91.23%
4. The Execution time of Recovery Block is 0.0281ms

Step7

1. The expected reliability of an NVP implementation is the same as for the Recovery Block, only all modules fail that means it fails. The reliability of SubsetSum is 90.94%, the reliability of SubsetSumDP is 72.55%, So the approximate theoretical reliability is:



1. With the assumptions in step7, and that the first module to finish returns the correct answer, we can know: there is 72.55% chance that it will take 0.0203ms to finish because there is 72.55% chance that SubsetSumDP will return correct result in 0.0203ms.

And there is 37.45% chance that it will take 27.209ms to finish, because it is 37.45% chance that the SubsetSumDP module will fail and we need to wait for the result of SubsetSum module which will take 27.209ms to finish.

So the approximate theoretical execution time is



Step8

**Part 2:**

1. The execution times as reported by the original implementation of the code.

Number of vertices = 87575

Number of edges = 121961

Time to build graph: 224886ms

Time to find shortest path (first time) from city8 to city8000: 113201ms

Time to find shortest path (second time) from city8 to city8000: 117331ms

Time to find cost of shortest path from city8 to city8000: 115356ms

Time to find shortest path from city8 to city3200: 114221ms

1. Changes:
2. Change the private field *names* in the class *WeightedGraph*, the original type is String[], new change it to a HashMap<String, Integer> type, for the reason that, the getIndex() method is frequently called to get the corresponding index of a given vertex label, so using a hashmap to find index O(1) is obviously faster than the original naïve O(n) implementation. This change is for improving data structure efficiency.
3. Add a private field *idxToNames* in the class *WeightedGraph* which type is String[] to support fast look up from index to vertex names which is originally supported by previous private field *names*. This change is for improving data structure efficiency.
4. In method *addWeightedEdge(),* Change line *addEdge(vertex1, vertex2, dist) to addEdge(i, j, w). I*n the original version, the line *addEdge(vertex1, vertex2, dist)* is just did nothing but first check the validity of the label *vertex1* and *vertex2* (which is already done by previous lines of code, so this part is unnecessary work) and then add a new edge. The work of adding a new edge into *Edges* is actually done by *addEdge(i, j, w)* in the last line of *addEdge(vertex1, vertex2, dist).* So actually we just need this line of code, and this change will save some unnecessary work.
5. Follow the change above, in method *addWeightedEdge(),* replace *addEdge(i, j, w)* withthree lines of code:

*Edges[i].*insertFirst*(idxToNames[j],w);* *Edges[j].insertFirst(idxToNames[i],*w); *numEdges++;* Because the check of index *i* and *j* is already done before the call of *addEdge(i, j, w),* so the actual word needed is done by these three lines of code.

1. The *getWeight()* in *WeightedGraph* method is frequently get called, in which the method *find()* of *EdgeLinkList* is get called. The implementation of *find()* is a simple implementation, which time complexity is O(n) to perform the look up. So change the *find()* method to have o(1) look up time complexity. In order to make this change, we need a HashMap data structure. So add a private field *HashMap<String, EdgeLink> VertexToEdgeLink* in the *EdgeLinkList* class, which maps the vertex name to the corresponding EdgeLink. Then add the corresponding code to the other places, e.g. the initialization in constructor, using *HashMap.get(key)* to replace the original code in *find()* method.
2. After the above change, the actual execution time is not decreased a lot, the improvement is negligible. The reason may be that the calculation of hash value of string outweighed the benefits of fast lookup. So we may need another kind of key: integer. So this time we need to change the *EdgeLink*, by adding an *int* field *iData*, that is, the index of the vertex. So we store the index besides the label of the vertex. Then change the corresponding code in other related places. Then change *HashMap<String, EdgeLink> VertexToEdgeLink* added in *EdgeLinkList* to *HashMap<Integer, EdgeLink> VertexToEdgeLink.*