Part 1 : Sorting

* Introduction of different method
  + Quick sort

A kind of Divide and Conquer algorithm. It divides the array by using the pivot into an array with numbers smaller than the pivot and the ones larger than it and then repeat the process by recursion.

* + Merge sort

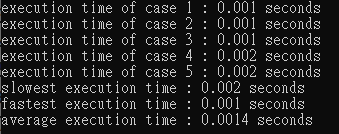
A kind of Divide and Conquer algorithm. It divides the array into smaller arrays of equal size with recursion, sort them individually and combine them in an sequential order.

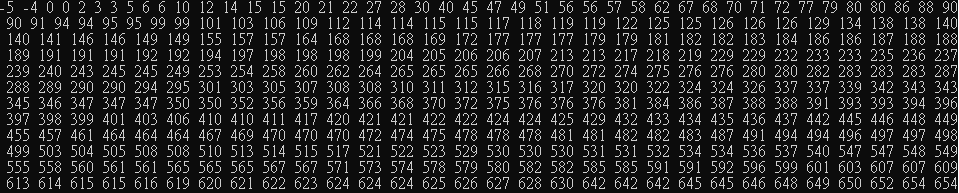
* Implementation details
  + Quick sort

1. Choose the most right number as the pivot, compare every number in the array with it, if the number is smaller than it, swap it with the current index in each iteration.
2. Do it recursively with array to the left of the pivot and to the right of it then we will get a sorted array eventually.
   * Merge sort
3. Divide the array into two parts of it in equal size and do it recursively.
4. When it is unable to split further, merge the split part of the array into a sorted array by linear comparison as the array must be a sorted one respectively due to recursion.

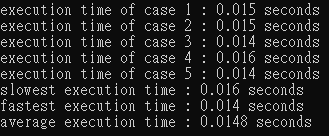
* Results

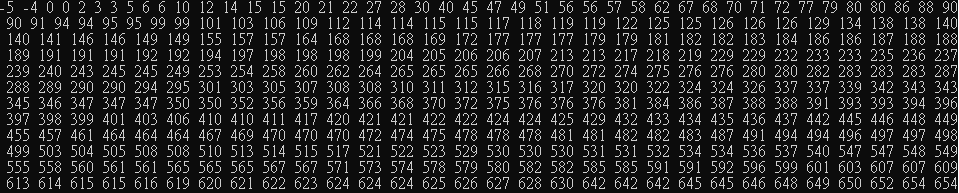
1. Execution time and results(5 case with 9000 numbers each case)
   * 1. Quick sort





* + 1. Merge sort





1. Stability
   * 1. Quick sort : unstable
     2. Merge sort : stable

* Discussion

1. Although time complexity of both algorithms is O(nlogn), I discovered that merge sort will spend more time. I guess it’s because merge sort has to copy the array in each recursion process while quick sort has not.
2. The most challenging part of these algorithms is to measure the execution time. At the beginning, I use the testcase given the TA. However, there are two problems. First, the execution time of all cases with quick sort is 0, as its actual execution time is less than a millisecond. Second, the input of each line in command has an upper limit, so it is unable to read in a case with too many cases. To solve those 2 problems, I wrote a program to output a file that has 5 cases of 9000 numbers in each, and separate all the numbers line by line.

* Conclusion

It turns out that although their time complexity are same, the execution time may have a little bit different due to process. However, if we need a stable sort, we still have to use merge sort instead of quick sort.

Part 2 : Minimum spanning tree

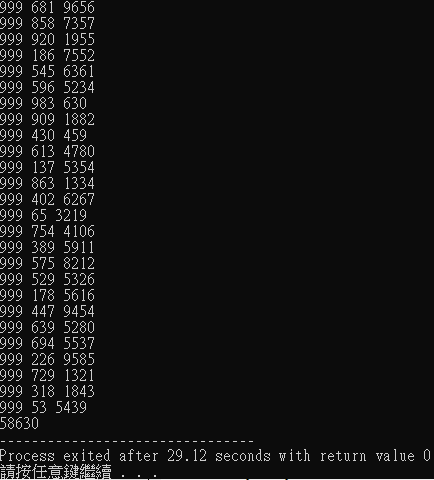
* Introduction of different method
  + Prim algorithm

A kind of greedy algorithm. It always chooses the smallest vertex in every step, this is an algorithm concentrating on vertices.

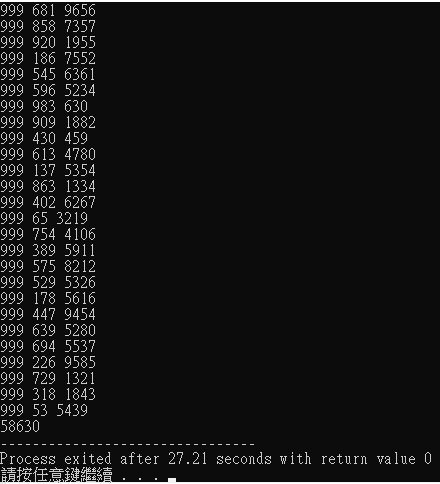
* + Kruskal algorithm

A kind of greedy algorithm. It always chooses the smallest edge in every step, this is an algorithm concentrating on edges.

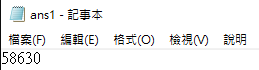
* Implementation details
  + Prim algorithm
    1. First, create an adjacency list of pairs of edges, and use a priority queue to save the adjacent pairs of edges of the starting node.
    2. Take out the smallest edges from the queue and then update the key of the node if the key can be updated smaller then remove it from the queue.
    3. Repeat the process until the queue is empty.
  + Kruskal algorithm
    1. First, create an adjacency list of pairs of edges, and create a vector of three-integer-pairs to save edges.
    2. Sort the edges by weight, and create sets of each node, then take each edge by the smallest weight, if the two nodes of the edge are not in the same set, take the edge and union them.
    3. Repeat the process until all the vertices are in the same set, i.e. a minimum spanning tree has generated.
* Results
  + Prim algorithm

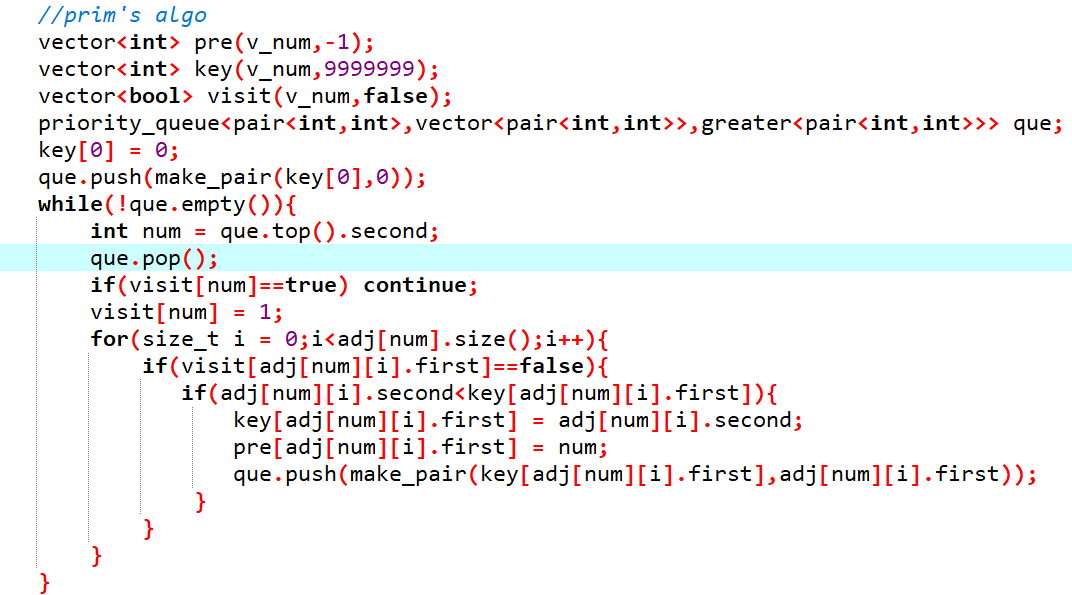


* + Kruskal algorithm



* + Answer



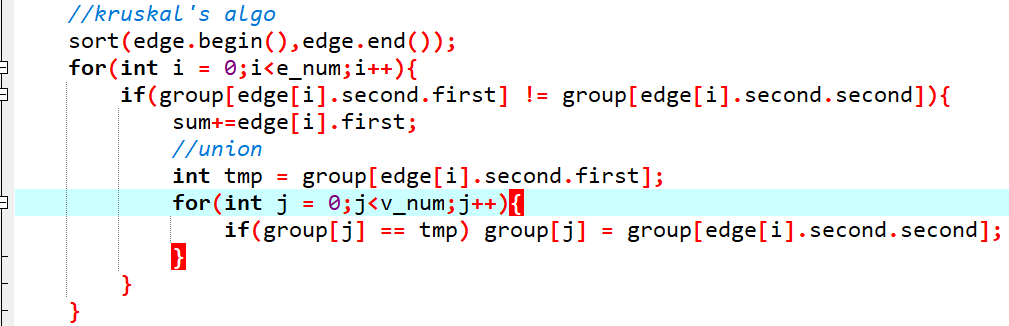
* Discussion
  + 1. Time complexity
  + Prim algorithm

O(V)

O(logV)

O(V)

T(n) = O(V)\*O(V)\*O(logV) = O(V^2\*logV) = O(ElogV)

* + Kruskal algorithm

O(V)

O(E)

O(ElogE)

T(n) = O(ElogE) + O(E)\*O(V) = O(ElogE)

2. Which is better under what condition

If the number of edges is much more than the number of vertices, we should use Prim’s algorithm as its time complexity is O(ElogE). Otherwise, we should use Kruskal’s algorithm.

* Conclusion

This is the part that I spend the most of my time on this homework. First, I need to study using pair to save data into priority queue in Prim’s algo. Second, I studied adjacency lists and forests because if I use adjacency matrix to save edges, some cases involved lots of vertices would cause TLE. I spend approximately 2 days on the two algorithms.

Part 3 : Shortest Path

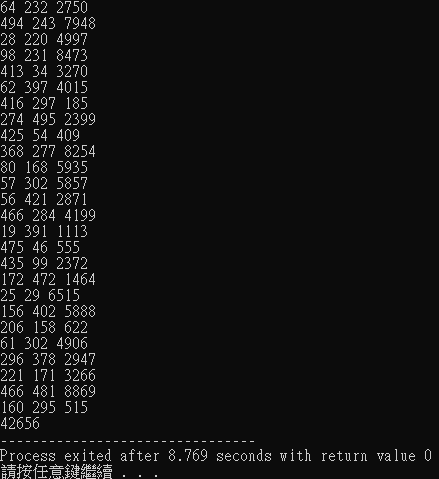
* Introduction of different method
  + Dijkstra algorithm

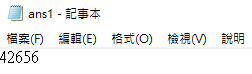
It’s a kind of greedy algorithm. Choose path from the adjacent vertices which has the smallest distance from the starting point every time by using priority queue or min-heap.

* + Bellman-Ford algorithm

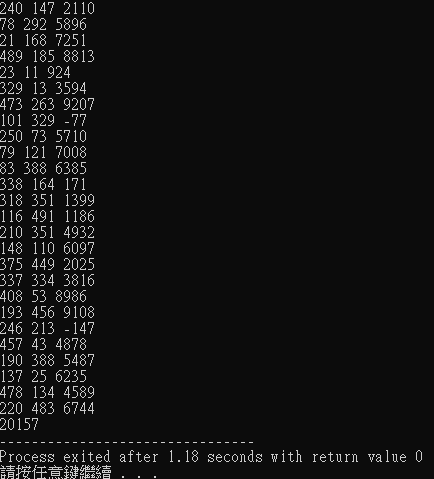
Traverse every edge by v-1 times, if the vertex can be updated to a smaller distance, then update it. After traversal, if we can still find a vertex which is able to update to a smaller distance, then a negative cycle is detected.

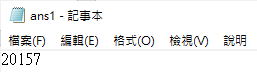
* Implementation details
  + Dijkstra algorithm
    1. First, create an adjacency list of pairs of edges, and use a priority queue to save the adjacent pairs of edges of the starting node. Create a bool vector to save whether the vertex reaches its smallest distance. Push the distance pair of the vertex that has the smallest distance into the queue.
    2. Take out the smallest edges from the queue and then update the key of the node if the key can be updated smaller. Remove the vertex from the queue and set it reaches its smallest distance.
    3. Repeat the process until the queue is empty.
  + Bellman-Ford algorithm
    1. First, struct an edge object vector to save all edges, then create a distance vector to save the distance of every vertex from the starting vertex.
    2. As every vertex has only at most v-1 in-degree (i.e. the chance to update its distance), we traverse every edge v-1 times to update all the values.
    3. We have to traverse all the edges one more time to ensure that every vertex is not able to update. If so, there’s no negative cycle detected; otherwise, a negative cycle is detected.
* Results
  + 1. Dijkstra algorithm





* + 1. Bellman-Ford algorithm





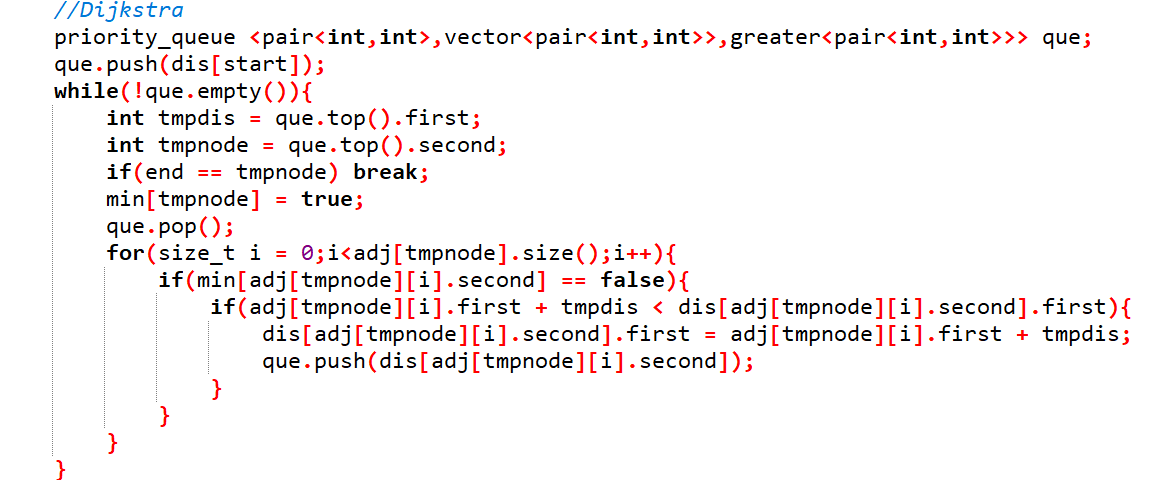
* Discussion
  + 1. How to detect negative cycle using Bellman-Ford?

After finishing the distance vector of all the vertices, if we can still find any of the vertex can be updated by a smaller value, a negative cycle is detected.

* + 1. How to print the path of the shortest path?

We need to create a vector to save each vertex’s parent node, then after updating all vertices to the smallest distance, print the vertex number sequentially by following the parent vertex vector.

* + 1. Time complexity
  + Dijkstra algorithm

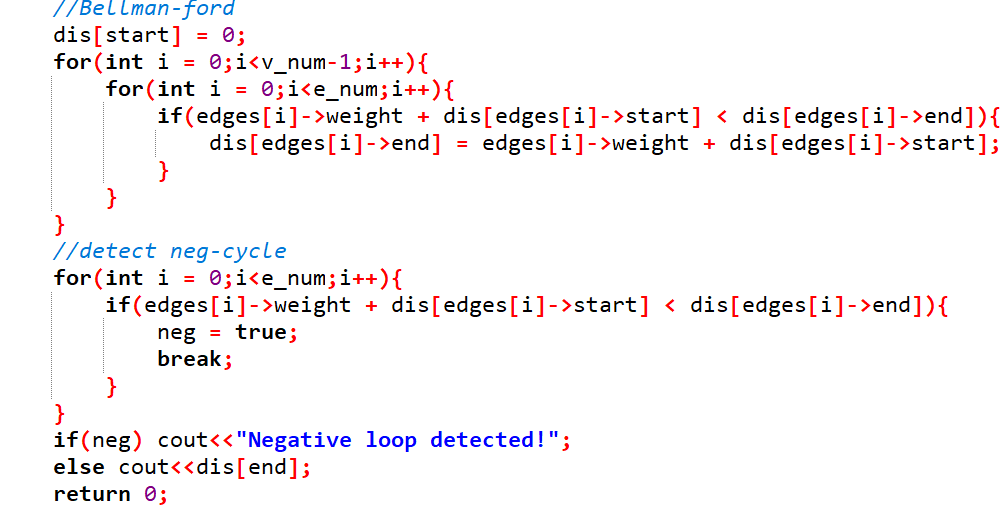


O(V)

O(V)

T(n) = O(V)\*O(V) = O(V^2)

* + Bellman-Ford algorithm



O(E)

O(V)

O(E)

T(n) = O(v)\*O(E) + O(E) = O(VE)

* Conclusion

If there’s no negative cycle, we should use Dijkstra as its time complexity is O(V^2); however, if there’s a negative cycle, we should use Bellman-Ford instead.