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

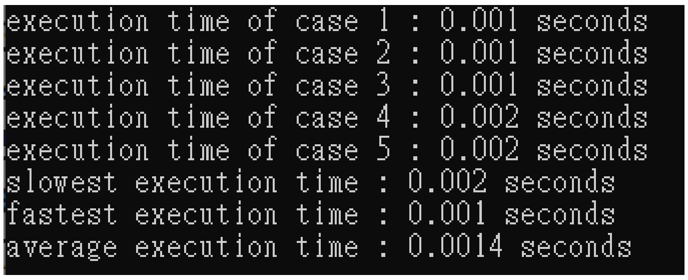
    1. Choose the most right number as the pivot, compare every number in the array with it, if the number is smaller than it, swap it with the current index in each iteration.
    2. Do it recursively with array to the left of the pivot

and to the right of it then we will get a sorted array eventually.

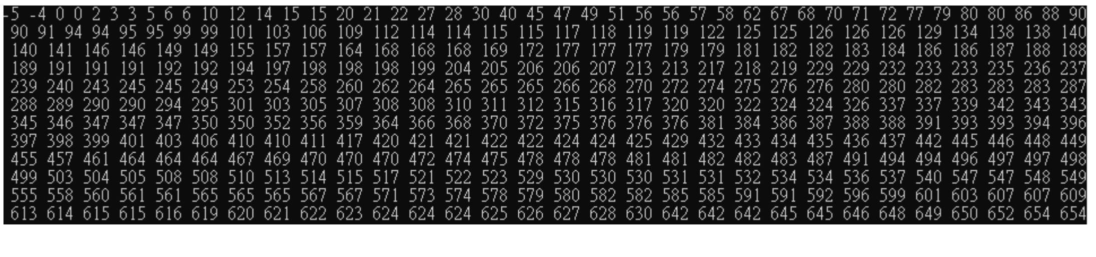
- Merge sort
  1. Divide the array into two parts of it in equal size and do it recursively.
  2. When it is unable to split further, merge the split part of the array into a sorted array by linear comparison as the array must be a sorted one respectively due to recursion.

Results

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

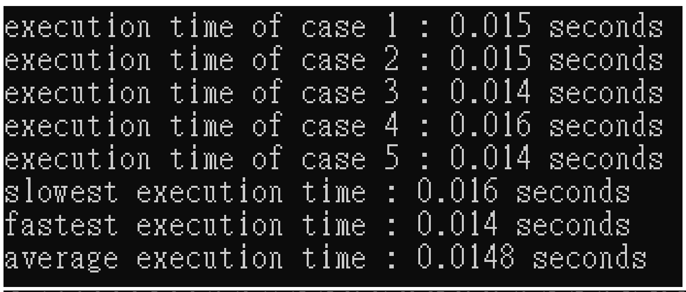


```
execution time of case 1 : 0.001 seconds
execution time of case 2 : 0.001 seconds
execution time of case 3 : 0.001 seconds
execution time of case 4 : 0.002 seconds
execution time of case 5 : 0.002 seconds
slowest execution time : 0.002 seconds
fastest execution time : 0.001 seconds
average execution time : 0.0014 seconds
```

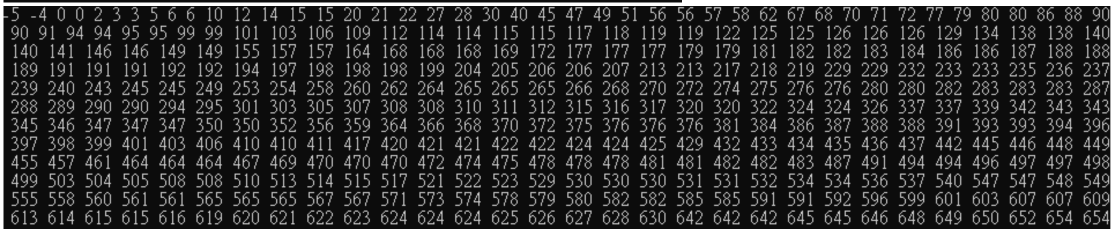


```
-5 -4 0 0 2 3 3 5 6 6 10 12 14 15 15 20 21 22 27 28 30 40 45 47 49 51 56 56 57 58 62 67 68 70 71 72 77 79 80 80 86 88 90
 90 91 94 94 95 95 99 99 101 103 106 109 112 114 114 115 115 117 118 119 119 122 125 125 126 126 126 129 134 138 138 140
 140 141 146 146 149 149 155 157 157 164 168 168 168 169 172 177 177 177 179 179 181 182 182 183 184 186 186 187 188 188
 189 191 191 191 192 192 194 197 198 198 198 199 204 205 206 206 207 213 213 217 218 219 229 229 232 233 233 235 236 237
 239 240 243 245 245 249 253 254 258 260 262 264 265 265 265 266 268 270 272 274 275 276 276 280 280 282 283 283 283 287
 288 289 290 290 294 295 301 303 305 307 308 308 310 311 312 315 316 317 320 320 322 324 324 326 337 337 339 342 343 343
 345 346 347 347 347 350 350 352 356 359 364 366 368 370 372 375 376 376 376 381 384 386 387 388 388 391 393 393 394 396
 397 398 399 401 403 406 410 410 411 417 420 421 421 422 422 424 424 425 429 432 433 434 435 436 437 442 445 446 448 449
 455 457 461 464 464 464 467 469 470 470 470 472 474 475 478 478 478 481 481 482 482 483 487 491 494 494 496 497 497 498
 499 503 504 505 508 508 510 513 514 515 517 521 522 523 529 530 530 530 531 531 532 534 534 536 537 540 547 547 548 549
 555 558 560 561 561 565 565 565 567 567 571 573 574 578 579 580 582 582 585 585 591 591 592 596 599 601 603 607 607 609
 613 614 615 615 616 619 620 621 622 623 624 624 624 625 626 627 628 630 642 642 642 645 645 646 648 649 650 652 654 654
```

   2. Merge sort



```
execution time of case 1 : 0.015 seconds
execution time of case 2 : 0.015 seconds
execution time of case 3 : 0.014 seconds
execution time of case 4 : 0.016 seconds
execution time of case 5 : 0.014 seconds
slowest execution time : 0.016 seconds
fastest execution time : 0.014 seconds
average execution time : 0.0148 seconds
```



```
-5 -4 0 0 2 3 3 5 6 6 10 12 14 15 15 20 21 22 27 28 30 40 45 47 49 51 56 56 57 58 62 67 68 70 71 72 77 79 80 80 86 88 90
 90 91 94 94 95 95 99 99 101 103 106 109 112 114 114 115 115 117 118 119 119 122 125 125 126 126 126 129 134 138 138 140
 140 141 146 146 149 149 155 157 157 164 168 168 168 169 172 177 177 177 179 179 181 182 182 183 184 186 186 187 188 188
 189 191 191 191 192 192 194 197 198 198 198 199 204 205 206 206 207 213 213 217 218 219 229 229 232 233 233 235 236 237
 239 240 243 245 245 249 253 254 258 260 262 264 265 265 265 266 268 270 272 274 275 276 276 280 280 282 283 283 283 287
 288 289 290 290 294 295 301 303 305 307 308 308 310 311 312 315 316 317 320 320 322 324 324 326 337 337 339 342 343 343
 345 346 347 347 347 350 350 352 356 359 364 366 368 370 372 375 376 376 376 381 384 386 387 388 388 391 393 393 394 396
 397 398 399 401 403 406 410 410 411 417 420 421 421 422 422 424 424 425 429 432 433 434 435 436 437 442 445 446 448 449
 455 457 461 464 464 464 467 469 470 470 470 472 474 475 478 478 478 481 481 482 482 483 487 491 494 494 496 497 497 498
 499 503 504 505 508 508 510 513 514 515 517 521 522 523 529 530 530 530 531 531 532 534 534 536 537 540 547 547 548 549
 555 558 560 561 561 565 565 565 567 567 571 573 574 578 579 580 582 582 585 585 591 591 592 596 599 601 603 607 607 609
 613 614 615 615 616 619 620 621 622 623 624 624 624 625 626 627 628 630 642 642 642 645 645 646 648 649 650 652 654 654
```

2. Stability

   3. Quick sort : unstable

   4. 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

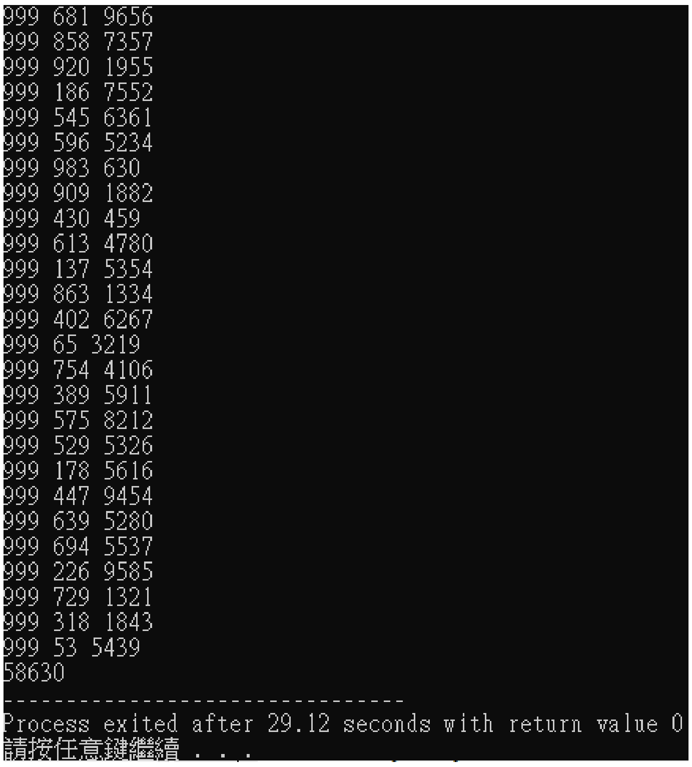
    1. First, create an adjacency list of pairs of edges, and use a priority queue to save the adjacent pairs of edges of the starting node.
    2. Take out the smallest edges from the queue and then update the key of the node if the key can be updated smaller then remove it from the queue.
    3. Repeat 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.

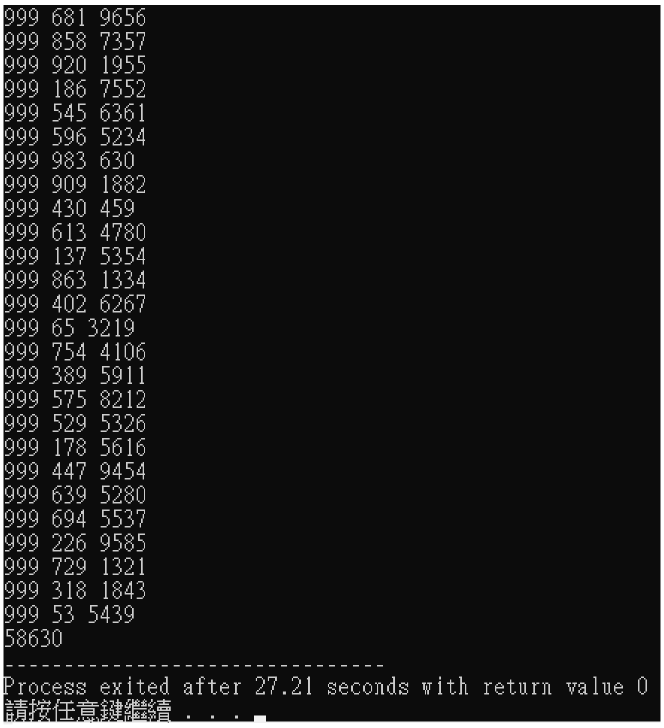
- Results

  - Prim algorithm



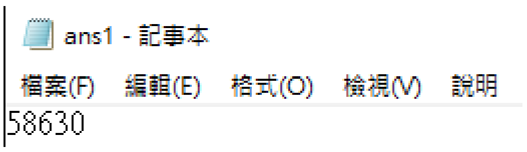
```
999 681 9656
999 858 7357
999 920 1955
999 186 7552
999 545 6361
999 596 5234
999 983 630
999 909 1882
999 430 459
999 613 4780
999 137 5354
999 863 1334
999 402 6267
999 65 3219
999 754 4106
999 389 5911
999 575 8212
999 529 5326
999 178 5616
999 447 9454
999 639 5280
999 694 5537
999 226 9585
999 729 1321
999 318 1843
999 53 5439
58630
--------------------------------
Process exited after 29.12 seconds with return value 0
請按任意鍵繼續 . . .
```

  - Kruskal algorithm



```
999 681 9656
999 858 7357
999 920 1955
999 186 7552
999 545 6361
999 596 5234
999 983 630
999 909 1882
999 430 459
999 613 4780
999 137 5354
999 863 1334
999 402 6267
999 65 3219
999 754 4106
999 389 5911
999 575 8212
999 529 5326
999 178 5616
999 447 9454
999 639 5280
999 694 5537
999 226 9585
999 729 1321
999 318 1843
999 53 5439
58630
--------------------------------
Process exited after 27.21 seconds with return value 0
請按任意鍵繼續 . . .
```

- Answer

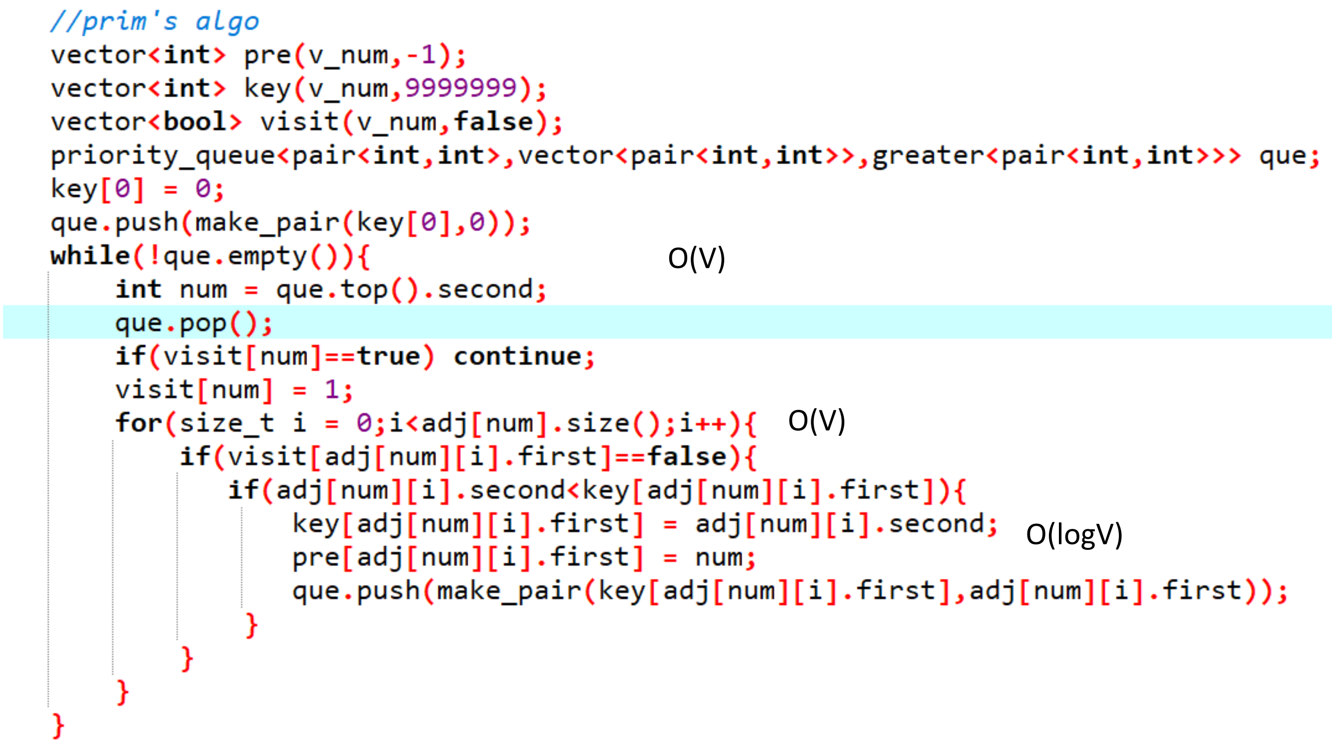


```
58630
```

- Discussion

1. Time complexity

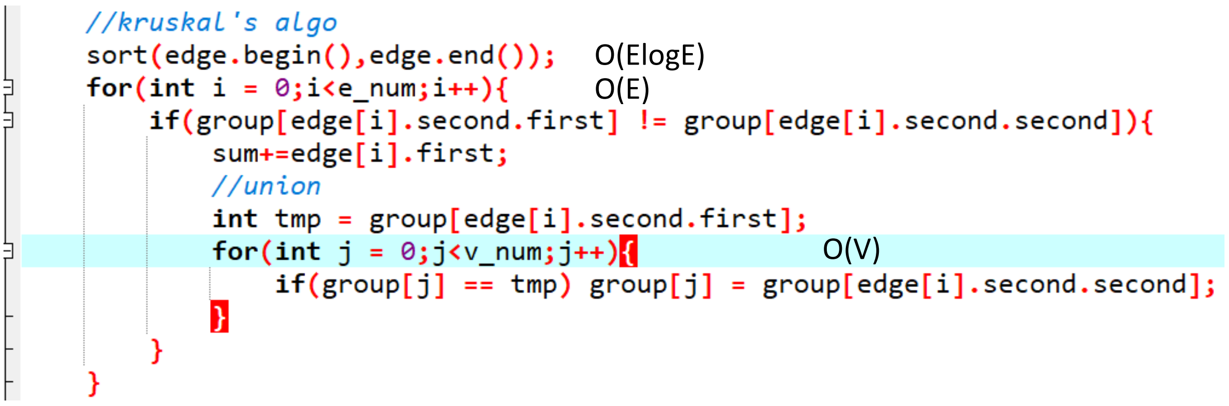
- Prim algorithm



```
//prim's algo
vector<int> pre(v_num,-1);
vector<int> key(v_num,9999999);
vector<bool> visit(v_num,false);
priority_queue<pair<int,int>,vector<pair<int,int>>,greater<pair<int,int>>> que;
key[0] = 0;
que.push(make_pair(key[0],0));
while(!que.empty()){                          O(V)
    int num = que.top().second;
    que.pop();
    if(visit[num]==true) continue;
    visit[num] = 1;
    for(size_t i = 0;i<adj[num].size();i++){  O(V)
        if(visit[adj[num][i].first]==false){
            if(adj[num][i].second<key[adj[num][i].first]){
                key[adj[num][i].first] = adj[num][i].second;  O(logV)
                pre[adj[num][i].first] = num;
                que.push(make_pair(key[adj[num][i].first],adj[num][i].first));
            }
        }
    }
}
```

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

- Kruskal algorithm



```
    //kruskal's algo
    sort(edge.begin(),edge.end());   O(ElogE)
    for(int i = 0;i<e_num;i++){      O(E)
        if(group[edge[i].second.first] != group[edge[i].second.second]){
            sum+=edge[i].first;
            //union
            int tmp = group[edge[i].second.first];
            for(int j = 0;j<v_num;j++){              O(V)
                if(group[j] == tmp) group[j] = group[edge[i].second.second];
            }
        }
    }
```

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.

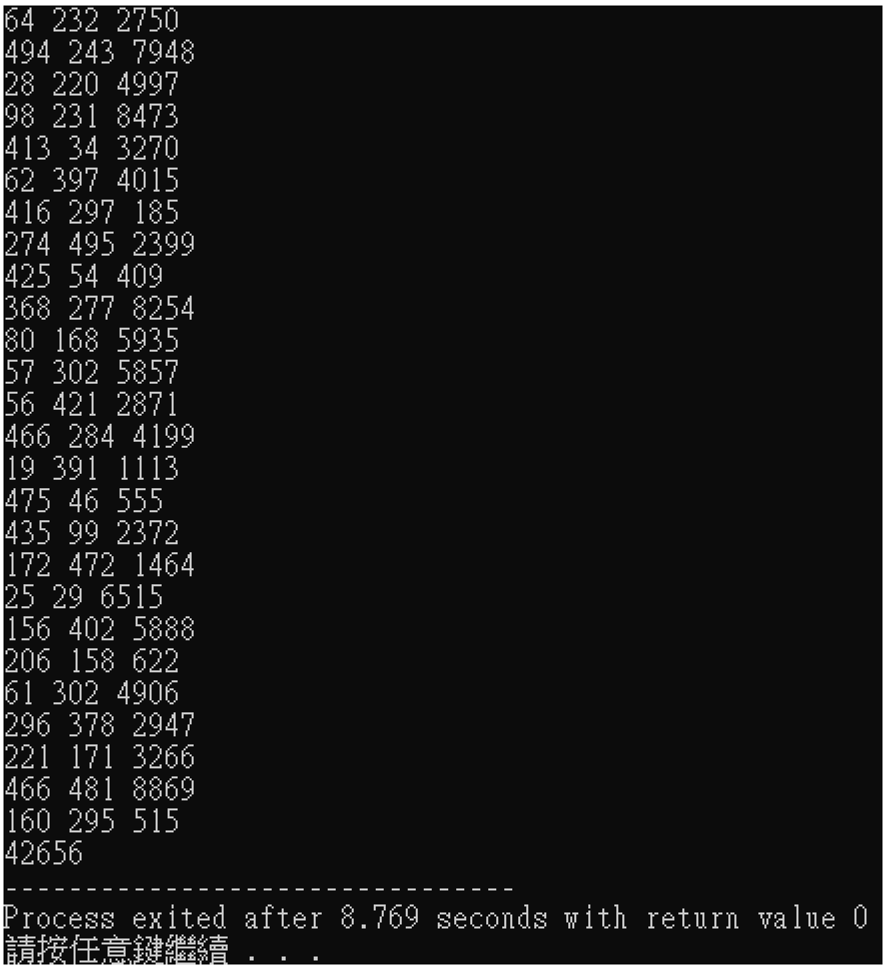
2. Take out the smallest edges from the queue and then update the key of the node if the key can be updated smaller. Remove the vertex from the queue and set it reaches its smallest distance.
3. Repeat the process until the queue is empty.

- 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;

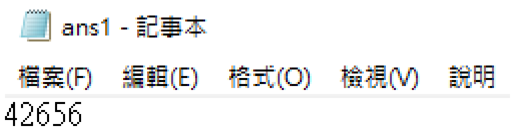
otherwise, a negative cycle is detected.

Results

1. Dijkstra algorithm

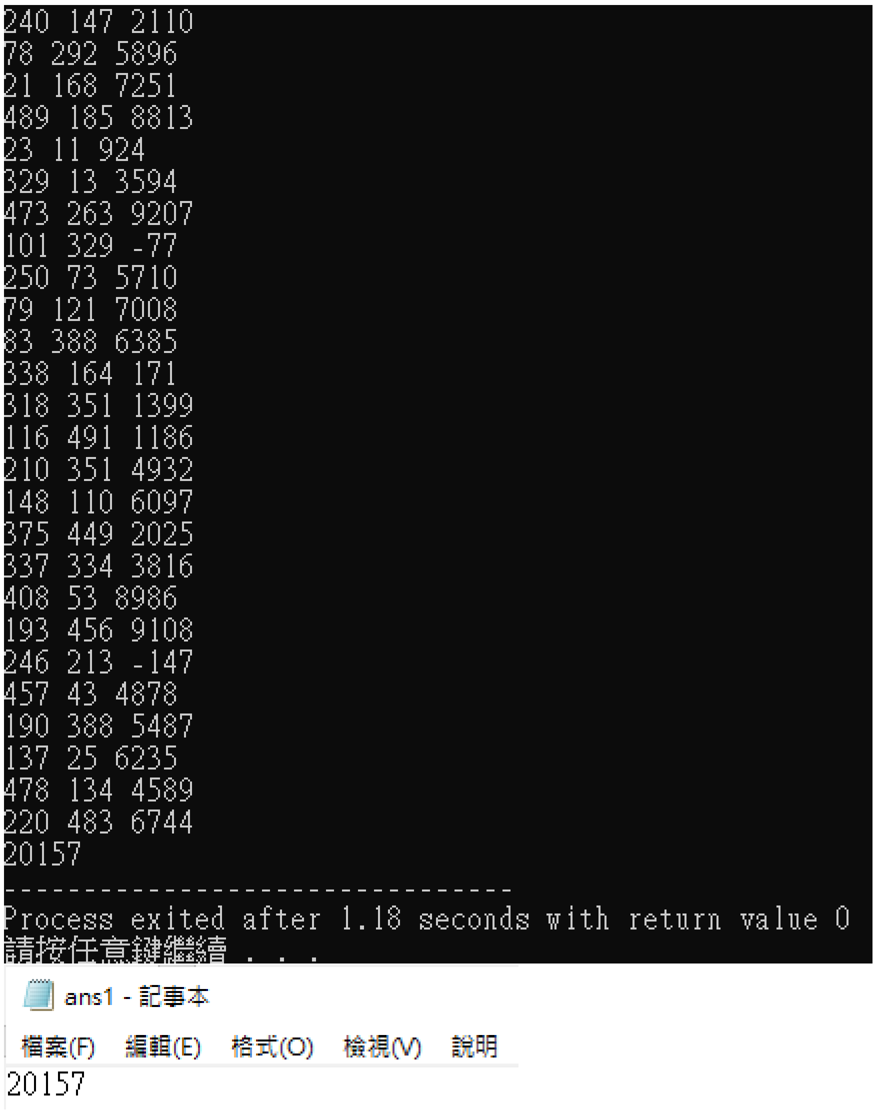


```
64 232 2750
494 243 7948
28 220 4997
98 231 8473
413 34 3270
62 397 4015
416 297 185
274 495 2399
425 54 409
368 277 8254
80 168 5935
57 302 5857
56 421 2871
466 284 4199
19 391 1113
475 46 555
435 99 2372
172 472 1464
25 29 6515
156 402 5888
206 158 622
61 302 4906
296 378 2947
221 171 3266
466 481 8869
160 295 515
42656
--------------------------------
Process exited after 8.769 seconds with return value 0
請按任意鍵繼續 . . .
```



```
42656
```

2. Bellman-Ford algorithm



```
240 147 2110
78 292 5896
21 168 7251
489 185 8813
23 11 924
329 13 3594
473 263 9207
101 329 -77
250 73 5710
79 121 7008
83 388 6385
338 164 171
318 351 1399
116 491 1186
210 351 4932
148 110 6097
375 449 2025
337 334 3816
408 53 8986
193 456 9108
246 213 -147
457 43 4878
190 388 5487
137 25 6235
478 134 4589
220 483 6744
20157
--------------------------------
Process exited after 1.18 seconds with return value 0
請按任意鍵繼續 . . .
```

```
20157
```

● Discussion

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

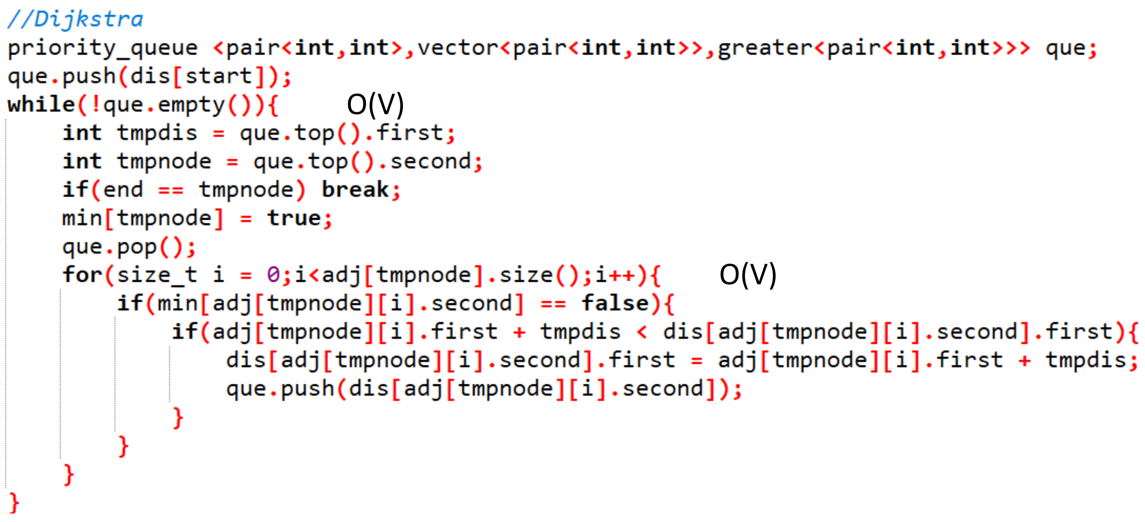
After finishing the distance vector of all the vertices, if we can still find any of the vertex can be updated by a smaller value, a negative cycle is detected.

2. 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.

3. Time complexity

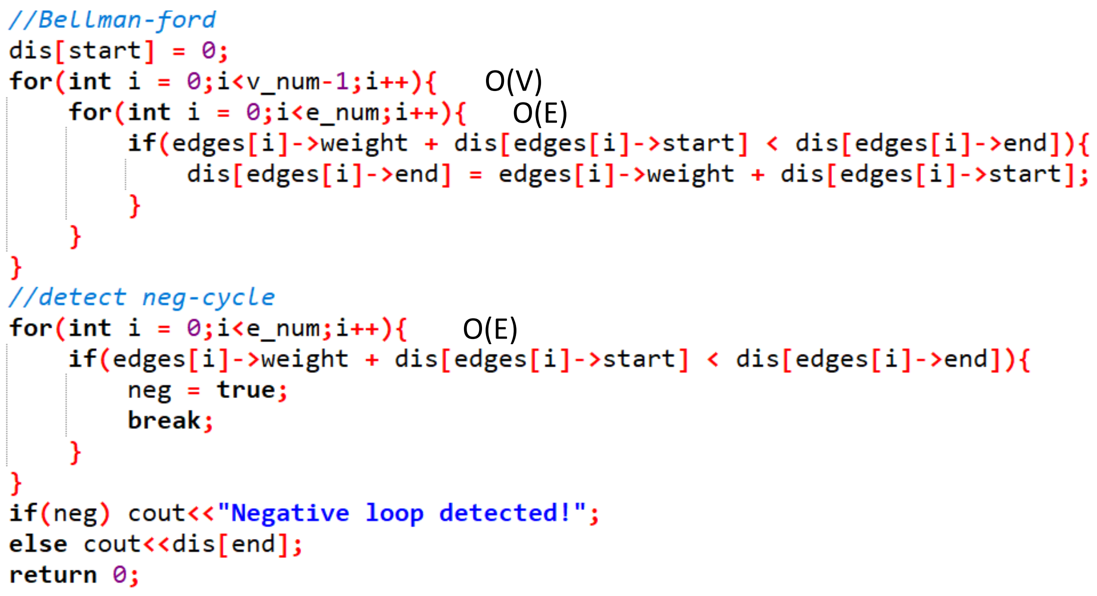
■ Dijkstra algorithm



```
//Dijkstra
priority_queue <pair<int,int>,vector<pair<int,int>>,greater<pair<int,int>>> que;
que.push(dis[start]);
while(!que.empty()){      O(V)
    int tmpdis = que.top().first;
    int tmpnode = que.top().second;
    if(end == tmpnode) break;
    min[tmpnode] = true;
    que.pop();
    for(size_t i = 0;i<adj[tmpnode].size();i++){      O(V)
        if(min[adj[tmpnode][i].second] == false){
            if(adj[tmpnode][i].first + tmpdis < dis[adj[tmpnode][i].second].first){
                dis[adj[tmpnode][i].second].first = adj[tmpnode][i].first + tmpdis;
                que.push(dis[adj[tmpnode][i].second]);
            }
        }
    }
}
```

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

■ Bellman-Ford algorithm



```
//Bellman-ford
dis[start] = 0;
for(int i = 0;i<v_num-1;i++){     O(V)
    for(int i = 0;i<e_num;i++){    O(E)
        if(edges[i]->weight + dis[edges[i]->start] < dis[edges[i]->end]){
            dis[edges[i]->end] = edges[i]->weight + dis[edges[i]->start];
        }
    }
}
//detect neg-cycle
for(int i = 0;i<e_num;i++){     O(E)
    if(edges[i]->weight + dis[edges[i]->start] < dis[edges[i]->end]){
        neg = true;
        break;
    }
}
if(neg) cout<<"Negative loop detected!";
else cout<<dis[end];
return 0;
```

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.