Paper Report

Sorting

I. Quick Sort

A. Introduction

1. Quick Sort 是在數列中選定一個數字當作 pivot（通常是最左邊或最右邊），然後把比這個數字小的數字移到它的左邊，比它大的移到右邊，接著在左右分別繼續這個過程，即可完成 sorting。

B. Implement Details

1. double execute(void):
   a) 我把執行的工作放在這邊而不是主程式，因為執行時間要測平均、最快和最慢，所以我測試 Execution time 的時候是讓同一筆測資在主程式跑十次算平均然後再分別記下最快和最慢，為了清楚看到執行怎麼運作所以另外寫了這個 function。
   b) 這個 function 裡面就是先在while 迴圈吃進總共有 N 個數字，如果 N 是 0 的話就會 break，接著設一個 array 把輸入的 N 個數字存進去。然後呼叫 quickSort 整理，最後輸出。
   c) 因為我的主程式是要計算各種執行時間，所以我這個 function 最後會回傳總共用了多久。
2. void quickSort(int* arr, int leftIndex, int rightIndex):
   a) 我的設定是整個 function 如果 leftIndex >= rightIndex 就不會動（怕他們到時候該在左邊的數字又跑到右邊，這樣就沒辦法整理好），接著設兩個變數 leftSearch, rightSearch 是拿來找左右分別有哪裡需要換位子的，然後我設定 pivot 在最左邊。
   b) 接著我用了巢狀 do-while，讓他們至少能走一步，最外層的迴圈是檢查 leftSearch 是不是還在 rightSearch 左邊，裡面有兩個 do-while 分別要找在左邊但是比 pivot 大的，和在右邊但是比 pivot 小的。當分別找到這兩個淘氣的數字，（而且 leftSearch 還在 rightSearch 左邊），就用 swap 把兩個數字換位子。當最外面的 do-while 也結束了就表示 leftSearch 在 rightSearch 右邊，所以用 swap 把 pivot 跟 rightSearch 換位子。
   c) 最後再呼叫兩次這個 function (recursive)，分別再整理原 pivot 的左邊和右邊。左邊是從 leftIndex 到 rightSearch-1，右邊是從 rightSearch+1 到 rightIndex，rightSearch 的位置上是原本的 pivot。

C. Discussion

1. Quick Sort 我寫的蠻快的，唯一卡比較久的 bug 是在兩個 do-while 迴圈那裡，因為我寫完第一個迴圈我就把它複製下來改成第二個，結果我忘記把 ++ 改成 —，還忘記把 < 改成 >，所以它就開始跑無限迴圈，不過我一開始找不到問題在哪所以弄了很久，發現之後我就下定決心以後不管兩行 code 長的多像我都不會用複製的了。
2. 不過有一個我現在還是不太清楚的地方，就是巢狀迴圈外的 swap，我不知道為什麼我寫 pivot 而不是 arr[leftIndex] 的時候，最後的輸出會是錯的，明明兩個變數的值是一樣的。

II. Merge Sort

A. Introduction

1. Merge Sort 也可以被叫做 Divide and Conquer，是先把數列拆成一個個只有一個數字的數列，再兩兩一組比較大小後合併，最後合併完整個就能完成 sorting。

B. Implement Details

1. double execute(void):
   a) 這部分和主程式都跟 Quick Sort 寫的一模一樣。
2. void mergeSort(int* arr, int left, int right):
   a) 先設定一個中位數，設定是如果右邊 index 大於左邊的話，就設定中位數的 index: mid 是 (left + right) / 2，接著呼叫兩次這個 function (recursive)，分別再整理 mid 的左邊和右邊。左邊是 left 到 mid，右邊是 mid+1 到 right。
   b) 最後用 merge 把左右數列合併。
3. void merge(int* arr, int left, int mid, int right):
   a) 首先設定左右兩個數列 tmpArrL, tmpArrR，把輸入的 arr 裡面的 element 分成兩邊用迴圈複製後存入。
   b) 然後進入 for 迴圈，會從 arr 的最左邊走到最右邊，在 index 各自小於 array 長度的條件下，如果 tmpArrL 的 element 比 tmpArrR 的小，就存入 arr，反之亦然。
   c) 跳出 for 迴圈後會再遇到兩個 while 迴圈，是為了確保如果其中一個 index 還是小於 array 長度，它剩下的 element 還能被存回去。

C. Discussion

1. 這個程式開始我的變數名稱就有取比較短，因為我決定不用複製的，但變數名稱太長會很麻煩，不過還是盡量讓變數名稱好懂。

2. 設定 tmpArrL, tmpArrR 的時候我的編譯器一直出錯，因為我不能用變數初始化這兩個 array，假設成 null pointer 的時候還會出現 segmentation fault，所以我中途還有把這些 array 改成 vector，但是從 array 變成 vector 再變回 array 也很麻煩，最後這個 function 變得很亂，我就用 array 重寫了一遍，然後才想到 hackerrank 上給的測資有限制大小不會超過 1000，我就直接把他們初始成 500，才解決了問題。
3. 在 merge 那個 function 的 for 迴圈裡面我一開始沒有放 index 各自小於 array 長度的條件，也沒有用 while 確認沒有剩下的 element，結果輸出就跑出 0 ，應該是因為我把左右數列初始成 0 ，然後他們各自跑太遠，後來加上這些條件就好了。

III. Results and Discussion between Quick Sort and Merge Sort

A. Results

1. Stability
   a) 根據助教給的資料，Quick Sort 不是 stable 的，而 Merge Sort 是。
2. Execution Time (1)
   a) 我的測試方式是利用迴圈跑十次相同測資（都是 hackerrank 的 sample input），每一次執行都會在最前面和最後面有 clock()，然後平均和記下最快與最慢時間。
   b) Quick Sort:
      (1) Average Time: 0.000122
      (2) Fastest Time: 0.000109
      (3) Slowest Time: 0.000195
   c) Merge Sort:
      (1) Average Time: 0.000114
      (2) Fastest Time: 0.000105
      (3) Slowest Time: 0.000162
   d) 可見 Merge Sort 所花的時間都比 Quick Sort 短。
3. Execution Time (2)
   a) 對 Quick Sort 來說最糟的情況是所有東西已經由大排到小了，所以每一次要找都要全部跑過，花的時間會最久，最好的情況是極值剛好在中間，可以把 sort 的地方平均分配。
   b) Merge Sort 的話是每一個數字都需要換位子的時候會花最久的時間。

B. Discussion

1. Merge Sort 時間短可能是因為 Quick Sort 最慢的情況下時間複雜度是 n^2，但 Merge Sort 一樣是 nlogn
2. 綜合 Stability 和 Execution Time 的比較之後，Merge Sort 對已經幾乎排好的數列來說是比較好的選擇，比較亂的就可以用 Quick Sort。
   (Reference: https://en.wikipedia.org/wiki/Sorting_algorithm#Stability)

IV. Conclusion

A. What I've learned & How many time I spent

1. 真的不要用複製的方式寫 code，發現 bug 有多蠢的時候會很想死。
2. Sorting 其實比起另外兩部分算寫的蠻快的，各花了一天下午，寫的時候其實沒什麼大問題，時間都是花在除錯。

B. Feedback to TA

1. 其實一開始我好像搞錯了 execution time 要分析什麼，後來問了同學才補上第二部分的分析，如果能再說明得更清楚就好了！謝謝！

Minimum Spanning Tree

I. Prim's Algorithm

A. Introduction

1. Prim's Algorithm 是先選定一個開始的點，然後從鄰近它的 edges 選一個最短的往下走，一直到所有 vertex 都被連到了就能完成 minimum spanning tree。

B. Implement Details

1. int main()
   a) 先分別輸入有幾個 vertex 和幾條 edge，然後用 graph 的 constructor 建立一個矩陣來根據起點和終點放 edge 的長度（下一點詳細說明），最後呼叫 graph 裡面的 prim。
2. class graph
   a) 裡面存放有 vertex 的個數、和一個由含有兩個 int 的 pair 組成的 vector 指標，拿來做名為 matrix 的陣列。
   b) 裡面有一個 constructor 和兩個函數。
3. graph :: void edge(int start, int end, int length)
   a) 利用 make_pair 把一個點和一個邊長合成一組，存入 matrix
4. graph :: void prim()

a) 先初始化，把距離設成無限大（我是設成助教所說的最大長度加一），然後把開始的點設成 0，它的距離也是 0。
b) 利用 priority_queue 排出一個由小到大排的 queue，然後先把開始的點存進去。
c) 進入 while 迴圈，確認 queue 不為空之後把最前面的 vertex 設為 current，如果已經經過了就離開，不然就繼續往下（此時丟掉最前面的 vertex 然後改成已經過），進入 for 迴圈，拿來判斷的是 current 的下一個點 next，如果還沒經過 next，而且往 next 走的路比他目前有的距離短，就把距離更新成那段路，然後把 next 這個點存進 queue，繼續判斷下一個點。
d) 最後用迴圈把總路途記下來，輸出。

C. Discussion

1. 原本要輸入的矩陣其實是用 array，但是叫進 function 的時候一直出問題，編譯器一直說格式不符，無論用 array 或 pointer 都無解，最後就改用 vector，之前沒有用過二維的 vector，透過這個學會了用兩層 resize 配置空間（雖然最後版本沒有用這個），vector 在很多地方比起 array 真的方便很多。
2. Prim 是我除錯除最久，也重寫了最多版本的演算法，可能是因為裡面的迴圈太多寫到我自己的邏輯也亂掉，過程是從一開始只有用 array，然後想說用 heap 做做看，到另外寫一個取最小值的 function，再到最後這個利用 pair 和 priority_queue 這些第一次用的東西，終於產出了 output 不會比正確答案多 14 的 prim 演算法（我真的不管怎麼寫答案都多 14，甚是難過）。

II. Kruskal's Algorithm

A. Introduction

1. Kruskal's Algorithm 是先把所有 edges 照大小排序，在不形成 cycle 的情況下從小到大連起來，一直到所有 vertex 都被連到了就能完成 minimum spanning tree。

B. Implement Details

1. int main()
   a) 先分別輸入有幾個 vertex 和幾條 edge，然後用二維的 vector 建立一個矩陣來根據起點和終點放 edge 的長度，如果沒有 edge 就設定那格是 0。然後呼叫 kruskal。
2. class Edge、int parent[1000]、int child[1000]

a) 這是一些接下來的 function 們會用到的變數和 class，所以我宣告成全域的。
b) Edge 裡面記錄了這條 edge 的長度、起點、終點。
c) parent 是為了記錄每一個 vertex 的上一個點是誰，為了防止出現 cycle 準備的。
d) child 是要記一下這個點後面跟了幾個小孩。

3. bool compare(const Edge left, const Edge right)
   a) 這是為了後面會遇到 sort 寫的，因為需要所有 edge 由小到大排。
4. int findParent(int a)
   a) 這個 function 會檢查傳進來的 vertex 是不是已經是它那條鏈上最開始的點，如果是的話它的 parent 就會是自己（因為我後面會把它們初始成自己）如果不是的話就利用 recursive 繼續往上找。
5. bool checkCycle(int a, int b)
   a) 這是要找如果這條 edge 加上 minimum spanning tree 有沒有 cycle 形成，有的話就不能放。
   b) 輸入的 a 和 b 分別會是這條 edge 的起點和終點，先用 findParent 幫他們找到他們的 root，如果 root 相同就表示把它們連起來會形成 cycle，就 return false。
   c) root 不同的話就可以加，兩個 root 比較之後，child 數目比較多的就可以當 parent（這樣比較不會亂掉）。
6. void kruskal(vector<vector<int>> matrix, int n, int m)
   a) 前面主要都是初始化，還有把矩陣裡的長度們存進 edges 這個 array，再用 sorting 把它們由小到大排序。
   b) 最後用一個 for 迴圈，如果用 checkCycle 確認過不會形成 cycle 就加在 totalCost 裡面，最後輸出。

C. Discussion

1. 我最原始的寫法是設定如果這個 edge 的頭跟尾有其中一個還沒 visited 就會把它存進去 minimum spanning tree，但這個寫法用 hakerrank 上的測資就是會少一條 edge，才發現這個寫法不可行的原因是可能有其中一條 edge 連起來之後不會形成 cycle，但他的兩個 vertex 剛好都已經被遇到過了。最後換了一個寫法是專注在 cycle 上。
2. 這個演算法我寫了比較多 function，主要是怕都擠在一起會看起來很亂，此外用了一個之前不知道（或是忘了）用法的 sort()，才知道要寫他要另外寫排序的 function，有學到新東西。

III. Results and Discussion between Prim's and Kruskal's Algorithm

A. Results

1. Time Complexity (*宣告、印出、設值都是 O(1)，不另外標出)

a) Prim's - O(m log m)

```
26      void prim()
27      {
28          int inEdge[m];
29          bool visited[m];
30          for (int i=0; i<m; i++) {
31              visited[i] = false;
32              inEdge[i] = 200001;
33          }
34          int start=0;
35          priority_queue<kav,vector<kav>,greater<kav>> queue;
36          queue.push(make_pair(0, start));
37          inEdge[start] = 0;
38
39          while (queue.empty()!=true) {
40              int current = queue.top().second;
41              queue.pop();
42
43              if (visited[current]==true) {
44                  continue;
45              }
46              visited[current] = true;
47              vector<kav>::iterator next;
48
49              for (next = matrix[current].begin(); next != matrix[current].end(); ++next) {
50                  int vertex = (*next).first;
51                  int weight = (*next).second;
52
49              for (next = matrix[current].begin(); next != matrix[current].end(); ++next) {
50                  int vertex = (*next).first;
51                  int weight = (*next).second;
52
53                  if (visited[vertex]==false && weight < inEdge[vertex]) {
54                      inEdge[vertex] = weight;
55                      queue.push(make_pair(inEdge[vertex], vertex));
56                  }
57              }
58          }
59          int totalCost = 0;
60          for (int i=0; i<m; i++) {
61              totalCost += inEdge[i];
62          }
63          cout << totalCost << endl;
64      }
65  };
```

O(m) — lines 30–37

O(m) — line 39

O(1) — lines 43–46

O(log m) — lines 53–56

O(m) — lines 49–58

O(m) — lines 60–62

O(m log m) — lines 26–64

b) Kruskal's - O(n log n)

```
56  void kruskal(vector<vector<int>> matrix, int n, int m)
57  {
58      int totalCost = 0;
59      Edge Edges[n];
60      bool visited[m];
61      for (int i=0; i<m; i++) {
62          visited[i] = false;
63      }
64      int k=0;
65      for (int i=0; i<m; i++) {
66          for (int j=0; j<m; j++) {
67              while (matrix[i][j] != 0 && k<n) {
68                  Edges[k].length = matrix[i][j];
69                  Edges[k].start = i;
70                  Edges[k].end = j;
71                  k++;
72                  break;
73                  }
74          }
75      }
76      sort(Edges, Edges+n, compare);
```

O(m) — lines 61–63

O(m^2) — lines 65–74

O(n log n) — line 76

O(n log n) — lines 56–76

```
77
78      for (int i=0; i<m; i++) {
79          parent[i] = i;
80          child[i] = 0;
81      }
82      for (int i=0; i<n; i++) {
83          if (checkCycle(Edges[i].start, Edges[i].end)) {
84              totalCost += Edges[i].length;
85          }
86      }
87
88      cout << totalCost << endl;
89
90  }
```

O(m)

O(n)

B. Discussion

1. 如果說邊的數目很多的話，用 Prim 會比用 Kruskal 快，因為 Kruskal 是專注在邊上，Prim 是專注在點上

IV. Conclusion

A. What I've learned & How many time I spent

1. 多學了蠻多 function 的用法，像是 priority_queue, iterator, pair，尤其是 pair，我原本都覺得要用 class 這種東西才能把我要的元素們包起來，雖然我還是更偏好用 class，但也是多了一個選擇。
2. 也有學到利用一些 vector 本身的功能，像是 resize，透過這次作業我算是真正熟悉了 vector 的用法，不然之前都覺得用 array 就好了。
3. Prim 真的花了我最久時間，所以這兩個演算法加起來花了我四天，其中三天是 Prim，沒日沒夜地改結果還是跑出多 14，後來才果斷跑去學這些新用法，雖然很痛苦但也是學了新的東西。

Shortest Path

I. Dijkstra's Algorithm

A. Introduction

1. Dijkstra 是選定一個開始的 vertex 之後把鄰近它的點的距離從無限大改成他們之間的 edge 長度，然後選到達的距離最短的點重複動作，最後得到最短路徑。

B. Implement Details

1. int main()
   a) 這部分的寫法跟 prim 很像，先分別輸入有幾個 vertex 和幾條 edge 以及起點終點，然後用 graph 的 constructor 建立一個矩陣來根據兩個端點放 edge 的長度（下一點詳細說明），最後呼叫 graph 裡面的 djikstra。

2. bool compare(vertex* left, vertex* right)
   a) 後面建立 heap 的時候會用到的，把最小的數字放到最後。
3. class vertex
   a) 先把每個 vertex 的距離設成無限大，然後記錄一個可以存 vertex 和邊長的 vector。
4. class graph
   a) 裡面有一個 constructor 和一個函數。
5. graph :: void dijkstra(graph* newGraph, int s, int t)
   a) 先把起點的距離設成 0，然後建立一個 minheap，透過主程式把 vertex們塞進去之後根據距離由大排到小，最後取出最小的那個設為 minVertex，如果它周邊還有路可以走就繼續。
   b) 進入 for 迴圈，如果這個點離原點的距離加上往它下一個點 next 的邊長比 next 到原點的距離短，而且這個點離原點的距離也已經不是無限大了的話，就更新 next 到原點的距離。
   c) 跳出迴圈後，這個點就不用了就可以刪掉，一直重複執行到 heap 裡面沒有東西，就印出到終點的最短距離。

C. Discussion

1. 寫 Dijkstra 因為是在寫完 Prim，所以對利用 pair 比較上手，另外因為這兩個演算法都可以用 minheap 的概念執行，已經在 prim 掙扎過的東西就可以直接套用，不過因為這邊的寫法比較多指標，容易寫一寫亂掉誰要指到誰，所以也花了不少時間。
2. 這邊特別提一下 make_heap 的用法，我原本以為 compare 的寫法應該是要由小排到大，所以自己糾結了一陣子，後來才想到因為我要取出的時候是從後面取，所以要由大排到小。

II. Bellman-Ford Algorithm

A. Introduction

1. Bellman-Ford 也是透過判斷點之間的 edge 長度來變更到起點的距離，跟 Dijkstra 最大的差別在他可以判斷有沒有 negative loop。

B. Implement Details

1. int main()
   a) 這部分跟 dijkstra 幾乎一樣，只有存的方式不是靠 pair，是分開存起點終點和邊長，最後呼叫 graph 裡面的 bellmanFord。
2. class vertex、class edge

a) 這邊會把 vertex 存成 class 是因為原本它還有一個內容物是要存 parent，後來發現其實用不到我就刪掉了，只留下距離。
b) edge 裡面就包含它的兩個端點和長度。

3. class graph
   a) 裡面有一個 constructor 和 一個函數
4. graph :: void bellmanFord(graph* newGraph, int s, int t)
   a) 先把起點距離設成 0，進入 for 迴圈，跑的次數是 (vertex - 1)，然後進第二層迴圈，設一個 current 表示現在在檢查的 edge，如果這條邊的起點的距離加上這條 edge 的長度比這條邊的終點的距離短，就更新它的終點的距離。
   b) 所有迴圈跑完就檢查，這時候如果還是有邊的起點的距離加上這條 edge 的長度比這條邊的終點的距離短的問題就表示這個圖形有 negative loop，沒有的話就印出到所求終點的最短距離。

C. Discussion

1. 寫完 Bellman-Ford 之後覺得其實跟 Dijkstra 的想法很像，就是沒有用 minHeap 取而是順順的一個點一個點找下去，還有最後會檢查有沒有 negative loop。
2. 原本其實也要用 pair 寫，但是寫的時候想說分開設不知道會不會比較清楚一點，結果寫完我還是比較偏好這種寫法，對我來說更直觀一點。
3. 不知道為什麼我第一個版本照老師講義上的 psuedo code 很快就寫出來了，結果不知道為什麼 hackerrank 後面的六個測資有四個一直過不了，請同學幫我除錯也找不出原因，只好作罷，個人覺得可能是哪個地方指標指錯，我真的用太多指標了。

III. Results and Discussion between Dijkstra's & Bellman-Ford Algorithm

A. Results

1. Time Complexity (*宣告、印出、設值都是 O(1)，不另外標出)
   a) Dijkstra's - O(m^2)

```
32  void dijkstra(graph* newGraph, int s, int t)
33  {
34      (newGraph->v + s)->d = 0;
35
36      vector<vertex*> minheap;
37      for (int i=0; i<numVertex; i++)
38      {                                          O(m)
39          minheap.push_back(v+i);
40      }
41      while (minheap.empty() == false) {  ————————————  O(m)
42          make_heap(minheap.begin(), minheap.end(), compare);
43          pop_heap(minheap.begin(), minheap.end(), compare);
44          vertex* minVertex = minheap.back();  ————————  O(log m)
```

O(m^2)

```
45
46          if (minVertex->matrix.empty() == true) {
47              continue;
48          }
49
50          for (int i=0; i<minVertex->matrix.size(); i++) {
51              pair<vertex*, int> next = minVertex->matrix[i];
52              if(minVertex->d + next.second < next.first->d && minVertex->d!=INF)
53              {
54                  next.first->d = minVertex->d + next.second;
55              }
56          }
57          minheap.pop_back();
58      }
59      cout << (newGraph->v + t)->d << endl;
60  }
```

O(1)

O(m)

b) Bellman-Ford - O(m^2)

```
37  void bellmanFord(graph* newGraph, int s, int t)
38  {
39      (newGraph->v + s)->d = 0;
40
41      for (int i=0; i<(numVertex-1); i++) {
42          for (int j=0; j<(newGraph->numEdge); j++) {
43              edge* current = (newGraph->e + j); // current edge
44              if (current->start->d + current->weight < current->end->d &&
                    current->start->d!=INF) {
45                  current->end->d = current->start->d + current->weight;
46              }
47          }
48      }
49
50      bool negativeLoop = false;
51      for (int i=0; i<numEdge; i++) {
52          edge* check = (newGraph->e + i); // check current edge
53          if (check->start->d + check->weight < check->end->d &&
                check->start->d!=INF) {
54              negativeLoop = true;
55              break;
56          }
57      }
58
59      if (negativeLoop == true) {
60          cout << "Negative loop detected!" << endl;
61      }
62      else cout << (newGraph->v + t)->d <<endl;
63  }
```

O(m)

O(n)

O(1)

O(n)

O(mn)

B. Discussion

1. 考慮時間複雜度的話，Dijkstra 會比較有效率，但如果要有 negative loop 偵測的功能則要選擇 Bellman-Ford。
2. 如果要印出最短路徑怎麼走的話，就幫兩個演算法各自加上 parent 的 array，每次更新距離的時候把這個點更新成下個點的 parent。

IV. Conclusion

A. What I've learned & How many time I spent

1. 寫 Dijkstra 的時候讓我順便複習了一下 heap 的用法，也好好地複習了一下指標，不然一直指錯東西。
2. 這兩個花的時間也蠻久，加起來大概三天，因為都是錯 hackerrank 後面的六個測資，不知道輸入了什麼就比較難除錯 。