# Homework 2

109550178 黃昱翰

## Part1 Sorting

### ●Introduction

◯ Quick Sort:

•一個分部解決的排序方法。

•每次選擇一個數字當作”Pivot”，讓所有比”Pivot”小的在”Pivot”左邊，大的則在右邊，分成兩部分後再各自取”Pivot”，以此完成排序，而我是取陣列中最後一個當作”Pivot”。

•不需要額外儲存空間。

◯ Merge Sort

•一個分部解決的排序方法。

•每次把陣列分成一半，分到每個陣列只有一個元素，然後再兩兩岸照順序合併，持續合併就可以得到排序好的陣列。

•需要額外儲存空間

### ● Implement Details

◯Quick sort:

•partition: 首先讓 pivot = array[last element], i = first element -1，再從頭到尾檢查檢查有沒有元素比 pivot 小，有的話則和 array[i]換，迴圈跑完就讓 array[i]和 pivot 換位置，讓 pivot 到正確的位置，回傳 pivot 的 index。

•quickSort: 先用 partition 找到 pivot，再用遞迴的方式，把陣列以 pivot 為分隔再分別做一次 quickSort。

◯Merge Sort:

•merge: 以中間為界線，創造兩個陣列把資料複製近來，讓兩個陣列元素比較大小，讓小的值存進陣列中。

•mergeSort: 先用遞迴的方式，以中間為界，分別做一次 mergeSort，使得所有元素自成一個集合，再 merge 起來。

### ● Results

◯ Time complexity

- Quick sort: O(nlogn)

```
int partition(int a[],int start,int end){
    //i is use to count the position where pivot should be placed
    int pivot = a[end];         O(1)
    int i = start-1;            O(1)
    for(int j=start; j <= end-1; j++){
        if(a[j]<pivot){
            i++;                                O(n)
            swap(&a[i],&a[j]);
        }
    }

    //let pivot swap to right position
    swap(&a[i+1],&a[end]);         O(1)
    return i+1;
}

void quickSort(int a[],int start,int end){   T(n)
    int pi;      //pi           O(1)
    if(start<end){                   O(1)
        pi = partition(a,start,end);    O(n)
        quickSort(a,start,pi-1);               T(n/2)
        quickSort(a,pi+1,end);       T(n/2)
    }
}
```

T(n) = O(1) + O(1) + T(n/2) + T(n/2) + O(n)

= 2T(n/2) + O(n)

= 2(2T(n/4) + O(n/2)) + O(n)

= 4(T(n/4)) + 2*O(n) + O(n)   // O(n/2) can be represented as O(n)

= 2kT(n/2k) + k*O(n)   // now replace k with logn to meet the base condition

= 2logn*1+logn*O(n)

=O(nlogn)

- Merge sort:  O(nlogn)

```
void merge(int arr[], int left, int mid, int right)
{
    int n1 = mid - left + 1;
    int n2 = right - mid;                       O(1)
    // Create temp arrays
    int L[n1], R[n2];
    // Copy data to temp arrays L[] and R[]
    for (int i = 0; i < n1; i++)
        L[i] = arr[left + i];                   O(n)
    for (int j = 0; j < n2; j++)
        R[j] = arr[mid + 1 + j];
    // Merge the temp arrays back into arr[left..right]
    // Initial index of first subarray
    int i = 0;                                   O(1)
    // Initial index of second subarray
    int j = 0;                                   O(1)
    // Initial index of merged subarray
    int k = left;
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k] = L[i];
            i++;
        }
        else {
            arr[k] = R[j];                              Θ(n)
            j++;
        }
        k++;
    }
    // Copy the remaining elements of
    // L[], if there are any
    while (i < n1) {
        arr[k] = L[i];
        i++;
        k++;
    }
    // Copy the remaining elements of
    // R[], if there are any
    while (j < n2) {
        arr[k] = R[j];
        j++;
        k++;
    }
}
```

```
void mergeSort(int arr[],int left ,int right){   T(n)
    int mid;        O(1)
    if(left<right){   O(1)
        mid = (left+right)/2;   O(1)
        mergeSort(arr,left,mid);     T(n/2)
        mergeSort(arr,mid+1,right);     T(n/2)
        merge(arr,left,mid,right);    O(n)
    }
}
```

T(n) = O(1) + O(1) + T(n/2) + T(n/2) + O(n)

= 2T(n/2) + O(n)

= 2(2T(n/4) + O(n/2)) + O(n)

= 4(T(n/4)) + 2*O(n) + O(n)

= 2kT(n/2k) + k*O(n)

= 2logn*1+logn*O(n)

=O(nlogn)

## ● Discussion

◯ Execution time & Stability

•下圖是極端例子 sorted array 下使用兩個 sort 的情況，可以看到 quick sort 會比 merge sort 慢很多，因為在這個情況下 merge sort 是 O(nlogn)但是 quick sort 則是 O(n^2)，下圖我不確定是不是資料量太大，無法負荷，但可以看出兩者相差很大。

```
Active code page: 950
Number of random numbers        Merge sort time         Quick sort time
10000                           0.001s                  0.221s
20000                           0.002s                  0.851s
30000                           0.002s                  1.862s
Press any key to continue . . .
```

•下圖是我以亂數生成-10000~10000 範圍的數字產生的結果

測試很多遍以後大部分平均時間是一樣的，最快時間是 quicksort 比較常小於 mergesort，最慢時間則反之。

| Number of random numbers | Merge sort time | Quick sort time |
|---|---|---|
| 10000 | 0.004s | 0.003s |
| 20000 | 0.009s | 0.006s |
| 30000 | 0.009s | 0.006s |
| 40000 | 0.011s | 0.01s |
| 50000 | 0.013s | 0.009s |
| 60000 | 0.014s | 0.01s |
| 70000 | 0.014s | 0.013s |
| 80000 | 0.017s | 0.014s |
| 90000 | 0.023s | 0.027s |
| 100000 | 0.037s | 0.018s |
| 110000 | 0.025s | 0.018s |
| 120000 | 0.025s | 0.022s |
| 130000 | 0.028s | 0.023s |
| 140000 | 0.029s | 0.025s |
| 150000 | 0.036s | 0.04s |
| 160000 | 0.044s | 0.029s |
| 170000 | 0.039s | 0.03s |
| 180000 | 0.04s | 0.033s |
| 190000 | 0.043s | 0.035s |
| 200000 | 0.047s | 0.037s |
| 210000 | 0.046s | 0.038s |
| 220000 | 0.05s | 0.039s |
| 230000 | 0.069s | 0.04s |
| 240000 | 0.054s | 0.045s |
| 250000 | 0.055s | 0.046s |
| 260000 | 0.057s | 0.049s |
| 270000 | 0.06s | 0.05s |
| 280000 | 0.062s | 0.054s |
| 290000 | 0.066s | 0.054s |
| 300000 | 0.069s | 0.064s |
| 310000 | 0.076s | 0.06s |
| 320000 | 0.086s | 0.058s |
| 330000 | 0.092s | 0.073s |
| 340000 | 0.093s | 0.066s |
| 350000 | 0.079s | 0.068s |
| 360000 | 0.081s | 0.069s |
| 370000 | 0.1s | 0.07s |
| 380000 | 0.116s | 0.071s |
| 390000 | 0.093s | 0.083s |
| 400000 | 0.098s | 0.084s |
| 410000 | 0.092s | 0.084s |
| 420000 | 0.098s | 0.086s |
| 430000 | 0.103s | 0.096s |
| 440000 | 0.119s | 0.145s |
| 450000 | 0.122s | 0.09s |
| 460000 | 0.103s | 0.091s |
| 470000 | 0.103s | 0.092s |
| 480000 | 0.121s | 0.099s |
| 490000 | 0.11s | 0.1s |
| 500000 | 0.114s | 0.1s |

| | Merge sort | Quick sort |
|---|---|---|
| Average time = | 0.06188s | 0.06188s |
| Fastest time = | 0.004s | 0.003s |
| Slowest time = | 0.122s | 0.145s |

•Quick sort 是 unstable，因為他是靠選取 pivot 來分割陣列，假如舉例有兩個一樣的數字分別放在陣列最後兩個位置，那他們順序就有可能會改變。

Merge sort 則是 stable，因為他是靠一小塊集合慢慢組起來的，若不需要變動遍不會變動。

◯ Which is better algorithm in which condition

Quick sort 適合用於小陣列且不需要額外儲存空間，但在大陣列中就比較沒有效率。偏好使用 Array

Merge sort 適合用於任何大小的陣列，但與用於較小任務的其他排序算法相比，速度較慢。偏好使用 Linked List

◯ Challenges you encountered

在比較執行時間這部分花了我很長時間，因為資料量太小，所以我也不知道到底是哪裡有問題，後來助教給測資後，我又遇到了一個小問題，不知道為甚麼好像超

過特定數字(800 左右)就沒有辦法複製，所以我不能直接全選複製貼上，要看到最後複製到哪接著複製，所以後來我就亂數來比較這兩個排序方法。

## ● Conclusion

○ What did you learn from this homework

Quick sort 和 Merge sort 的差異和 Stability 的重要性以及如何計算執行時間。

○ How many time did you spend on this homework

應該有 6~7 小時

# Part 2 Minimum Spanning Tree

## ● Introduction

○Prim Algorithm

- a greedy algorithm
- 除了 source 設為 0 以外其他的 vertex 都設為無限大，可以用 min-heap 或 priority_queue 來實作，每次取出最小的 vertex，標註為 visited，檢查鄰近的 vertex 有沒有拜訪過，若沒有拜訪過就檢查從 source 到鄰近 vertex 是否有更近的路並 update distance，計算 MST cost 時把所有 vertex 的 distance 加起來就可以了。
- focus on vertex

○ Kruskal Algorithm

- a greedy algorithm
- 讓每個 vertex 自成一個集合，把所有 edges 由小到大排序，每次取最小的 edge，檢查兩端的 vertex 是否在同個集合，是則紀錄下這個 edge，若在同一個集合則略過，持續下去就可以找到 MST 的所有 edges，以此計算 cost。
- focus on vertex

## ● Implement Details

○ Prim Algorithm

• addEdge : 用 adjacency list 來建造 Graph，index 是 source，value 是 destination 和 weight 的 pair。

• primmest : 創造一個 priority queue，三個 vector，key 用來儲存從 source 到 vertex 的 cost；parent 紀錄路線；inMST 確認有無拜訪過。用迴圈確認 adjacency list 所有原色的 key 值是否大於 weight 並 update，最後把所有 key 值相加就可以得到 cost。

○ Kruskal Algorithm.

•DisjointSets: initialize parent 和 rank，parent 用來記錄上一個 vertex，rank 則是用來記錄階層關係。

•find : 找尋 i 所在的集合。

•merge : 合併 x 和 y 的集合，確認 rank[x]和 rank[y]的大小關係並記錄 parent 關係。

•addEdge : 用 vertex 紀錄所有 edges 的 weight,source 和 destination

•kruskalMST: 把所有 edges 按照由小到大排序，創造 DisjointSets，用迴圈確認所有 edges 的 source 和 destination 是否在同個集合，若不同便紀錄 weight 並 merge 兩個 vertex 和計算 cost。

# ● Results

○ Time complexity

• Prim Algorithm

```
void Graph::primMST()
{

    priority_queue< iPair, vector <iPair> , greater<iPair> > pq;          O(1)

    int src = 0; // Taking vertex 0 as source

    vector<int> key(V, INF);
    vector<int> parent(V, -1);
    vector<bool> inMST(V, false);

    pq.push(make_pair(0, src));          O(1)
    key[src] = 0;

    /* Looping till priority queue becomes empty */
    while (!pq.empty())          O(V)
    {
        int u = pq.top().second;
          //Different key values for same vertex may exist in the priority queue.
        pq.pop();          O(log V)
          //The one with the least key value is always processed first.
          //Therefore, ignore the rest.
        if(inMST[u] == true){
            continue;
        }

        inMST[u] = true;  // Include vertex in MST          O(V²logV)
        // 'i' is used to get all adjacent vertices of a vertex
        vector< pair<int, int> >::iterator i;
        for (i = adj[u].begin(); i != adj[u].end(); ++i)          O(V)
        {
            // Get vertex label and weight of current adjacent
            // of u.
            int v = (*i).first;          O(1)
            int weight = (*i).second;

            //  If v is not in MST and weight of (u,v) is smaller
            // than current key of v
            if (inMST[v] == false && key[v] > weight)          O(1)   O(logV)   O(VlogV)
            {
                // Updating key of v          O(1)
                key[v] = weight;
                pq.push(make_pair(key[v], v));          O(logV)
                parent[v] = u;          O(1)
            }
        }

    }
    int cost=0;
    // Print edges of MST using parent array
    for (int i = 1; i < V; ++i)
        cost += key[i];          O(V)
    cout <<cost <<endl;
}
```

Time complexity = O(1)+O(V)+O(V2logV)

= O(V2logV) = O(E logV)

- Kruskal Algorithm.

```
int Graph::kruskalMST()
{
    int mst_wt = 0; // Initialize result

    // Sort edges in increasing order on basis of cost
    sort(edges.begin(), edges.end());    O(elog e)

    // Create disjoint sets
    DisjointSets ds(V);    O(V)
    // Iterate through all sorted edges
    vector< pair<int, iPair> >::iterator it;    O(1)
    for (it=edges.begin(); it!=edges.end(); it++)    O(e)    O(ev)
    {
        int u = it->second.first;
        int v = it->second.second;    O(1)

        int set_u = ds.find(u);    O(1)
        int set_v = ds.find(v);

        // Check if the selected edge is creating
        // a cycle or not (Cycle is created if u    O(v)
        // and v belong to same set)
        if (set_u != set_v)    O(1)
        {
            // Current edge will be in the MST
            // so print it
            //cout << u << " - " << v << endl;

            // Update MST weight
            mst_wt += it->first;    O(1)

            // Merge two sets
            ds.merge(set_u, set_v);    O(V)
        }
    }
```

Time complexity: O(V+ElogE + EV)

= O(E logE)

# ● Discussion

○ Your discover

上課的時候我認為 kruskal 應該比 prim 簡單很多，但實際寫 code 下來我反而覺得 prim 比較清楚，因為 kruskal 的集合我花了好久才想通。

○ Which is better algorithm in which condition

- Prim Algorithm

Prim 算法的時間複雜度為 $O(V^2)$，V 是頂點的數量，可以使用斐波那契堆提高到 O(E + log V)。

Prim 只能作用在 connected graph。

Prim 在密集的圖形中跑得更快

• Kruskal Algorithm.

Kruskal 算法的時間複雜度是 O(E log V)，V 是頂點的數量。

Kruskal 的算法可以在任何時刻生成 forest（斷開的組件），也可以處理斷開的組件

Kruskal 在稀疏的圖形中跑得更快

○ Challenges you encountered

我覺得理解原理和實際實作起來真的差很多，研究了好久才理解應該要怎麼做，尤其是 kruskal 紀錄的方式，我覺得這 Part 是 HW2 中最難的。

## ● Conclusion

○ What did you learn from this homework

兩個演算法的差異以及實作的方法。

○ How many time did you spend on this homework

應該有 7~8 個小時

# Part 3 Shortest Path

## ● Introduction

○ Dijkstra's Algorithm

大致想法和 prim 差不多，讓所有除了 source 以外的 vertex 設為 INF，用 Min heap 來找尋最短的路程，每次找尋最小 distance 的 vertex，讓所有鄰近的 vertex 的 distance 比較 source distance+wieght 的大小，若是小於就 update，以此得到 shortest path。

○ Bellman Ford’s Algorithm

選擇一個 vertex 當 source，做 v 次 iteration，每次檢查每個 edge ，若有 source distance + weight < destination distance 就 update，做到第 V 次時，若還可以 update 就代表有 negative-weight cycle。

# ● Implement Details

○ Dijkstra's Algorithm

- dijkstraDist:首先 initialize vertex 的 distance，visited 紀錄各個 vertex 有無拜訪過，path 紀錄最短路徑，使用 unordered_set 去防止重複拜訪，比較 source distance+wieght 和 destination distance 的大小並決定是否 update，回傳 vector 儲存 source 到各個 vertex 的 distance。

○ Bellman Ford’s Algorithm

- BellmanFord: 首先 initialize vertex 的 distance，做 V-1 次 iteration，每圈跑再 E 次迴圈比較 distance 並 update，跑完之後再跑一個 E 次的迴圈確認是否還有 destination distance > source distance + weight，有的話就代表有 negative loop 就沒有 shortest path 了。

# ● Results

○ Time complexity

- Dijkstra's Algorithm

```
vector<int> dijkstraDist(vector<Node*> g, int s, vector<int>& path)
{

    vector<int> dist(g.size());

    // Boolean array that shows
    // whether the vertex 'i'
    // is visited or not
    bool visited[g.size()];
    for (int i = 0; i < g.size(); i++) {
        visited[i] = false;
        path[i] = -1;                        O(V)
        dist[i] = infi;
    }
    dist[s] = 0;
    path[s] = -1;
    int current = s;

    unordered_set<int> sett;             O(V)
    while (true) {

        // Mark current as visited
        visited[current] = true;
        for (int i = 0; i < g[current]->children.size();i++) {  O(V)
            int v = g[current]->children[i].first;
            if (visited[v])
                continue;                      O(1)

            sett.insert(v);            O(1)                     O(V)     O(V^2)
            int alt = dist[current] + g[current]->children[i].second;

            if (alt < dist[v]) {
                dist[v] = alt;                O(1)
                path[v] = current;
            }
        }
        sett.erase(current);        O(logV)
        if (sett.empty())
            break;              O(1)

        // The new current
        int minDist = infi;            O(1)
        int index = 0;

        // Loop to update the distance
        // of the vertices of the graph
        for (int a: sett) {                O(V)
            if (dist[a] < minDist) {
                minDist = dist[a];
                index = a;
            }
        }
        current = index;
    }
    return dist;
}
```

Time complexity: O(V^2)

- Bellman Ford’ s Algorithm

```
void BellmanFord(Graph* graph,int source, int target){
    int V = graph->vernum;
    int E = graph->edgenum;        O(1)
    int dist[V];

//Initialize
    for(int i=0; i<V; i++){        O(V)
        dist[i] = INF;
    }
    dist[source] = 0;

//Iteration
    for(int i=0; i<V-1; i++){      O(V)
        for(int j=0; j<E; j++){        O(E)
            int u = graph->edge[j].start;
            int v = graph->edge[j].end;                     O(EV)
            int weight = graph->edge[j].weight;    O(1)

            if(dist[u] != INF && dist[u] + weight < dist[v]){
                dist[v] = dist[u] + weight;             O(1)
            }
        }
    }

//Check negative cycle
    for(int i=0; i<E; i++){
        int u = graph->edge[i].start;
        int v = graph->edge[i].end;            O(1)        O(E)
        int weight = graph->edge[i].weight;

        if(dist[u] != INF && dist[u] + weight < dist[v]){
            cout << "Negative loop detected!" <<endl;    O(1)
            return;
        }
    }

    cout << dist[target] << endl;
}
```

Time complexity: O(VE)

# ● Discussion

○ Describe how you detect the negative loop in the Bellman-Ford Algorithm.

做了 v-1 次後，所有 vertex 的 distance 應該要是最小的不會再有變動，因此如果還可以繼續 update 的話就代表有 negative loop。

○ If you need to print out the path of the shortest path, describe how it can be done?

可以用遞迴的方式，寫一個函式 printPath，由於原本就有紀錄 parent 或是 child，可以由此關係去印出 child 再進入遞迴，最終印出完整路徑。

○ Which is better algorithm in which condition

Dijkstra 沒有辦法用在有負權重的 edge 圖形上，但 Bellman-Ford 可以偵測是否有負權重循環。Bellman-Ford 算法比 Dijkstra 更耗時。

○ Challenges you encountered

一開始我照著網路上別人的作法打，但不知道為甚麼就是有些測資沒有過，後來才發現那是 undirected 的，花了好長時間修修改改才終於寫出來。

# ● Conclusion

## ○ What did you learn from this homework

兩個演算法的差異和實作方式，有方向性和沒有方向性的圖之間的差異和實作方法。

## ○ How many time did you spend on this homework

應該也是 7~8 小時。

# ● Feedback to TA

雖然這學期都是 online judge，後面又都遠距上課，基本上都沒什麼看過助教，但還是很謝謝助教很認真回答我們的問題，我自己是覺得和其他同學比起來，自己寫程式真的滿爛的，每次的 Lab 都至少要花比別人多的時間才可以寫完，這次的作業也是幾乎都要先看網路上別人是怎麼做的，要不然只看 psuedo code，我根本不知道要從何下手，雖然我沒有主動問過助教問題，但討論區的那些問題都對我幫助滿大的，例如這次 sorting 的 execution time 的問題，其實我也還不是很確定到底對不對，但至少應該是有做出個樣子，總之，感謝助教們這一年來的幫助!