

Report of Algorithm

109550031

李旻融

Sorting

- Introduction -p. 2
- Implement Details -p. 2
- Results -p. 6
- Discussion -p. 10
- Conclusion -p. 10

Minimum Spanning Tree

- Introduction -p. 11
- Implement Details -p. 11
- Results -p. 16
- Discussion -p. 18
- Conclusion -p. 19

Shortest Path

- Introduction -p. 20
- Implement Details -p. 20
- Results -p. 23
- Discussion -p. 25
- Conclusion -p. 26

PART 1: sorting

[Introduction]

- **Quick Sort:** 在數列中選最右邊的數當 pivot，比 pivot 小的數往 pivot 左邊放，比 pivot 大則往右邊放，在 pivot 左右兩邊各自成為一個新的數列，一直重複此步驟，直到分不出新的數列。
- **Merge Sort:** 將數列對半拆解，分成很多個小數列，直到每個數列都只剩下一個元素，接著比較兩兩數列裡頭元素的大小，由小到大合併成一個新數列，直到恢復成一個數列即完成排序。

[Implement Details]

➤ Quick Sort

在這個排序法裡，我總共寫了三個 function，並且利用 array 傳入各個 function 來完成 Quick Sort。

1. **Swap:** 這個 function 會用在 Partition(下個 function)裡面，用來交換比 pivot 小的數，讓小的數字確實在 pivot 前面，因為必須是一個 pointer 傳入，所以在 Partition 那邊，必須用 &傳入位置。在 function 裡，設一個新的 variable temp，先讓他等於 a 的值，再讓 a 的值等於 b 的值，接著讓 b 的值等於 temp，便可成功交換兩者位置，並順利更改 main 裡的 array，而不是只在此 function 裡頭更改。

```
void Swap(int *a,int *b){  
    int temp;  
    temp = *a;  
    *a = *b;  
    *b = temp;  
}
```

2. **Partition:** 這個 function 用來找出 pivot 和分割 array，傳入 array 和頭尾的 index，並可以決定要取 array 的哪一段，我讓每次傳進來的最右邊當 pivot，並且設一個 for 迴圈，讓他從頭到尾檢查一遍，每當遇到比 pivot 小的數值便利用上一個 swap function 將其往前放，並且設一個 a 當作頭，一開始要減一，這樣才能確保第一個比 pivot 小的會放到 front 那個位置，因為 if 裡頭是小於等於 pivot，所以 pivot 必定會往前放，而且 pivot 又是最後一個檢查的數值，此時 pivot 的左邊必定都小於等於 pivot，右邊都大於 pivot，最後還要 return pivot，在 Quicksort function(第三個 function)裡會用到。

```
int Partition(int *arr,int front,int end){
    int pivot,a;
    pivot = end;
    a = front-1;
    for(int i=front;i<=end;i++){
        if (arr[i] <= arr[pivot]){
            a++;
            Swap(&arr[a],&arr[i]);
        }
    }
    return a;
}
```

3. **Quicksort:** 在這個 function 裡，首先利用 if(front<end)判斷 array 是否有超過一個數值，如果是才繼續分割下去，前一個 Partition 會將數列分割好並且 return pivot 值，此時便可以利用遞迴關係，分別讓 pivot 前後的兩個數列繼續進行分割，直到不能再分割為止。

```
void Quicksort(int *arr,int front,int end){
    int pivot;
    if (front < end){
        pivot = Partition(arr,front,end);
        Quicksort(arr,front,pivot-1);
        Quicksort(arr,pivot+1,end);
    }
}
```

4. **Main Function:** 利用一個 while 迴圈，確定可以重複輸入，但只要輸入 0 便結束程式，根據題目，一開始會先給定 array 的大小，因此我們可以直接定義 array 大小，並將 array 的各個數值 cin 進 array 的對應位置，接著便執行 Quicksort 這個 function，再將排好的 array 印出。

```
int main(){
    int size;
    while(1){
        cin >> size;
        if(size==0)
            break;
        int arr[size];
        for(int i=0;i<size;i++){
            cin >> arr[i];
        }

        Quicksort(arr,0,size-1);

        for(int i=0;i<size;i++){
            cout << arr[i] << " ";
        }
        cout << endl;
    }

    return 0;
}
```

➤ Merge Sort

在這個排序法裡，共有兩個 function，並且利用 array 來完成。

1. **Mergesort:** 首先會先將 array 和頭尾傳入 function，在 if 裡是要判斷是否超過一個數值，如果是的話，設一個 middle 代表中間的數，將 array 分成頭到 middle 和 middla+1 到尾，就這樣一直進行下去，直到全部都只剩自己，最後則要 merge(下一個 function)。

```

void Mergesort(int *arr,int left,int right){
    int middle;
    if(right>left){
        middle = (left+right)/2;
        Mergesort(arr,left,middle);
        Mergesort(arr,middle+1,right);
        Merge(arr,left,middle,right);
    }
}

```

2. **Merge:** 在上個 function 裡頭有 call Merge function，就是執行以下的程式，要將數個 array 從小到大合併，首先我們必須找到 L 和 R 兩個 array 的大小並且設置他，下一步將小的依序存入原先的 array(從傳入的第一個，也就是 left 開始存)，為了避免 L 和 R 有其中一個先存完，造成不能繼續比較，我設置了 if 條件，只要有一方先存完，另一方便可接著存入。

```

void Merge(int *arr,int left,int middle,int right){
    int lsize = middle-left+1;
    int rsize = right-middle;
    int L[lsize],R[rsize];

    for(int i=0;i<lsize;i++){
        L[i] = arr[left+i];
    }
    for(int i=0;i<rsize;i++){
        R[i] = arr[middle+1+i];
    }

    int i=0,j=0;
    for(int k=left;k<=right;k++){
        if(i<lsize && j<rsize){
            if(L[i]<=R[j]){
                arr[k] = L[i];
                i++;
            }
            else{
                arr[k] = R[j];
                j++;
            }
        }
        else if(i<lsize){
            arr[k] = L[i];
            i++;
        }
        else if(j<rsize){
            arr[k] = R[j];
            j++;
        }
    }
}

```

3. **Main Function:** 和 Quick Sort 相像，利用一個 while 迴圈，確定可以重複輸入，但只要輸入 0 便結束程式，根據題目，一開始會先給定 array 的大小，因此我們可以直接定義 array 大小，並將 array 的各個數值 cin 進 array 的對應位置，接著便執行 Mergesort 這個 function，再將排好的 array 印出。

```
int main(){
    int size;
    while(1){
        cin >> size;
        if(size==0){
            break;
        }
        int arr[size];
        for(int i=0;i<size;i++){
            cin >> arr[i];
        }

        Mergesort(arr,0,size-1);

        for(int i=0;i<size;i++){
            cout << arr[i] << " ";
        }
        cout << endl;
    }
    return 0;
}
```

[Results]

➤ Quick Sort

1. **Judging** (助教測資的案太長貼不上來，因此放 hackerrank)

Q1 - Quick Sort Algorithm

| Problem | Submissions | Leaderboard | Discussions |
|---|--------------|-------------|--------------|
| Submitted a few seconds ago • Score: 100.00 | | | |
| Status: Accepted | | | |
| ✓ | Test Case #0 | ✓ | Test Case #1 |
| ✓ | Test Case #3 | ✓ | Test Case #4 |
| | | ✓ | Test Case #2 |
| | | ✓ | Test Case #5 |

2. **Time Complexity:** 利用只有排列順序不同但同樣數值的資測進行測試最佳及最糟的情況，由下面可發現最糟的情況在這短短測資中便有差異，由此可知，在排列更多數值時，此差異將變得更明顯，但根據時間複雜度，最好的情況可能到時依然差異不大。

- ✓ **Average Time:** 運用 $T(n)$ 的算式，我們能夠遞推出平均所需要的時間應為 $O(n \log n)$ 。

```
int main(){
    int size;
    clock_t start,end;
    while(1){
        cin >> size;
        if(size==0)
            break;
        int arr[size];
        for(int i=0;i<size;i++){
            cin >> arr[i];
        }

        start = clock();
        Quicksort(arr,0,size-1);
        end = clock();

        cout << double(start-end)/CLOCKS_PER_SEC << endl;
    }
}
```

Input (stdin)

```
7
-10 5 8 -6 4 3 4
0
```

Your Output

```
-2e-06
```

- ✓ **Fastest Time:** 在分割的時候，如果每次都恰好將列表成兩個幾乎相等的 array，這樣只要進行 $\log n$ 次，但因為左右兩個 array 是不同的，所以需要時間 $O(n \log n)$ 。但從 a 圖可發現照這樣邏輯去設計的數據，確實花較少時間。

Input (stdin)

```
7
-10 3 -6 4 8 5 4
0
```

Your Output

```
-1e-06
```

(a)

Input (stdin)

```
7
5 4 4 3 -6 -10 8
0
```

Your Output

```
-3e-06
```

(b)

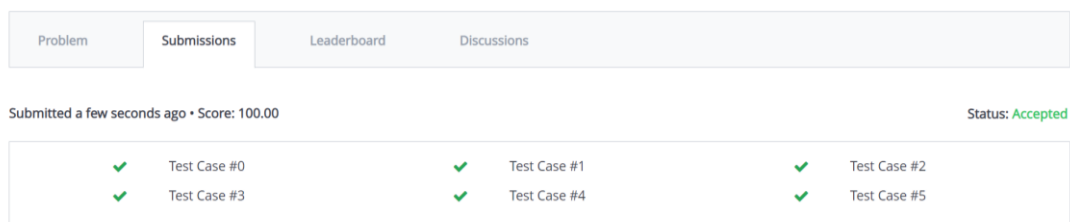
- ✓ **Slowest Time:** 如果 pivot 是數列(n 個數值)裡最大或最小，會發生一邊沒有數值，一邊有 $n-1$ 個數值的情況，如果這種狀況重複發生，便要進行 $n-1$ 次，在這種情況下，Quick Sort 需要 $O(n^2)$ 的時間。從 b 圖可發現照這樣邏輯去設計的數據，確實花較多時間。

3. **Stability:** 如果輸入裡面有兩個相同的數值，經過排列之後，相對順序保持不變，則算是 stable，在 Quick Sort 中，發生了大量交換，導致它的 unstable。舉例來說，假設現在有個數列是 5(1)、1、5(2)、4、3，那經過第一次變化後會變成 1、3、5(2)、4、5(1)，再一次 1、3、4、5(2)、5(1)，此時已排列完畢，我們也可以發現兩個五的位置對調了，因此我們可以證明 Quick Sort is unstable.

➤ Merge Sort

1. **Judging** (助教測資的案太長貼不上來，因此放 hackerrank)

Q1 - Merge Sort Algorithm



| Problem | Submissions | Leaderboard | Discussions | | |
|---|--------------|-------------|--------------|---|--------------|
| Submitted a few seconds ago • Score: 100.00 Status: Accepted | | | | | |
| ✓ | Test Case #0 | ✓ | Test Case #1 | ✓ | Test Case #2 |
| ✓ | Test Case #3 | ✓ | Test Case #4 | ✓ | Test Case #5 |

2. **Time Complexity:** 和 Quick Sort 一樣，利用只有排列順序不同但同樣數值的資測進行測試最佳及最糟的情況，同時，我也用和 Quick Sort 相同的測資，這樣就可以順便比較兩者的時間。根據時間複雜度，最好和最糟的情況或許仍有些許差異，但在排列更多數值時，此差異可能小到不足以掛齒。

- ✓ **Average Time:** 重點在於合併，每回合的合併需要花 $O(n)$ ，總共需要 $\log n$ 回合，因此所需時間為 $O(n \cdot \log n)$ ，也可以運用 $T(n)$ 的算式，我們能夠遞推出平均所需的時間和 Quick Sort 相同，應為 $O(n \cdot \log n)$ 。

```
int main(){
    int size;
    clock_t start,end;
    while(1){
        cin >> size;
        if(size==0){
            break;
        }
        int arr[size];
        for(int i=0;i<size;i++){
            cin >> arr[i];
        }
        start = clock();
        Mergesort(arr,0,size-1);
        end = clock();
        cout << double(start-end)/CLOCKS_PER_SEC << endl;
```

Input (stdin)

```
7
-10 5 8 -6 4 3 4
0
```

Your Output

```
-2e-06
```

- ✓ **Fastest Time:** 因為重點在於合併，每回合的合併依然需要花 $O(n)$ ，依然需要 $\log n$ 回合，所以和 average time 一樣需要 $O(n \log n)$ 。從 a 圖可發現照這樣邏輯去設計的數據，確實花相同時間。

Input (stdin)

```
7
-10 3 -6 4 3 5 4
0
```

Your Output

```
-2e-06
```

(a)

Input (stdin)

```
7
5 4 4 3 -6 -10 8
0
```

Your Output

```
-2e-06
```

(b)

- ✓ **Slowest Time:** 因為重點在於合併，每回合的合併依然需要花 $O(n)$ ，依然需要 $\log n$ 回合，和 average time、fastest time 都一樣需要 $O(n \log n)$ 。從 b 圖可發現照這樣邏輯去設計的數據，確實花相同時間。
- ✓ **Comparison with Quick Sort:** 和 Quick Sort 測試相同測資，Merge Sort 最佳、最壞情形和平均執行時間幾乎和 Quick Sort 的平均執行時間一樣，計算出所需時間也都是 $O(n \log n)$ ，這更加驗證了時間複雜度的準確性。

3. **Stability:** 因為 Merge Sort 不像 Quick Sort 一直隨意交換位置，假設在 array 中有兩個五，傳進 Merge function 的時候是按照順序的，如果今天兩個都在 L array 或 R array 的話，原先在前面的一定會先傳進去，如果今天一個在 L array 一個在 R array 的話，原先在前面的一定在左邊，後者一定會在右邊，輪到這兩個比大小時，雖然一樣大，但 L array 會先輸入，所以前面的依然在前面，經過排列之後，相對順序依然保持不變，所以 Merge Sort is stable.

[Discussion]

1. **My Discover:** 一開始不知道 array 就有 swap 的 function 可以直接用，還自己寫了一個，經過這次的作業，我知道有好多工具其實都在身旁，正等待我們去發現。
2. **Situation Analysis:** Merge Sort 和 Quick Sort 在執行時間上差不多，一般而言，選哪一個都差不多，但如果不想改變同數值的排列順序，則要選擇 Merge Sort，因為它是 stable。
3. **Challenges:** 因為上學期在上計概的時候，有稍微交一下 sorting，那時候我們寫的是 bubble sort 和 insertion sort，因此在 sorting 的部分，並沒有遇到太多困難，唯一的困難是我在寫 Merge Sort 的時候，一開始有 bug，有些輸出會變成零，但我一直找不出來在哪，後來才突然想到，如果 L 填完但 R 還沒，他們依然會繼續比下去，我才加了那些 if，只要其中一個 array 填到底了，另一個可以直接填進去，不用經過比較。

[Conclusion]

1. **Learning:** 我了解到了 stable 是什麼意思，還有它有一定的重要性，還有學會分析 Time Complexity，最重要的當然是徹底熟悉了 Merge Sort 和 Quick Sort 的方法，並且能夠將他們寫出來。
2. **Spending Time:** coding 3 小時 / report 4 小時
3. **Feedback:** 我認為這次的作業很有意義，讓我學到了很多有關 sorting 的知識，不知道助教為什麼選擇 Quick Sort 和 Merge Sort，或許是因為比較常用，但也許之後可以多分析幾種 sorting，可能一次的作業就是所有練習 sorting。

PART 2: minimum spanning tree

[Introduction]

- **Prim Algorithm:** 一開始設所有 vertex 的值都為無限大，除了起點設為零，每次從最小的 vertex A 開始找，其連接的所有 edge 的 weight，只要小於對應 vertex B 的值，便將 vertex B 的值更新為上連接 AB 的 edge 的 weight，並將 vertex B 的 parent 登記為 vertex A，等所有 vertex 都檢查過，便能得到最小生成樹。
- **Kruskal Algorithm:** 此演算法則是從 edge 下手，一開始有幾個 vertex 則有幾個 set，從最小的 edge 開始下手，只要此 edge 不會造成 cycle 的話，就新增此 edge，並將那兩個 vertex 的 set 合在一起，一直新增 edge 直到所有 vertex 都在同個 set，這也代表所有 vertex 都被連接起來了，這樣便能得到最小生成樹。

[Implement Details]

➤ Prim Algorithm

創造了一個 class，並且寫了兩個 function 在裡頭。

1. **Class / Constructor:** class 有兩個 function 在 public，private 則有一個 variable vnum，是指 vertex 個數，並且有個 list，型態則是<int, int>的 pair，constructor 傳入 vertex 的個數，接著創造一個 graph，裡頭的 vnum 是傳入的 vertex 個數，list 的大小也是依照 vnum 來決定。

```
typedef pair<int,int> pai;

class Graph{
public:
    Graph(int V);
    void AddEdge(int s,int e,int w);
    void Prim();
private:
    int vnum;
    list<pai> *lis;
};

Graph::Graph(int vnum){
    this->vnum = vnum;
    lis = new list<pai>[this->vnum];
}
```

2. **AddEdge:** input 會傳入 edge 頭尾的 vertex 值和 edge 的 weight，因為題目說沒有方向性，所以雙向都要存，將尾巴 vertex 值和 weight 創造 pair 並存進頭的 list，也將頭 vertex 值和 weight 創造 pair 並存進尾巴的 list。

```
void Graph::AddEdge(int s,int e,int w){  
    lis[s].push_back(make_pair(e,w));  
    lis[e].push_back(make_pair(s,w));  
}
```

3. **Prim:** 首先創造一個由小排到大的 priority queue 叫 pq，因為 priority queue 是根據 first 排序的，要按照 edge 的 weight 的大小依序輸入，所以 pq 的 first 必須是 weight，second 則是相對應 vertex，除此之外，還要三個大小都是 vnum 的 vector，第一個 parent 用來記錄他們的 parent 是誰，初始化所有值為 -1，第二個 vector 用來存 key，初始化起點為 0，其他則為無限大(前面有 define)，第三個 visited 則是用來確認是否檢查過這個 vertex，一個 vertex 檢查一遍就好，初始化所有值為 false。因為題目規定從零開始，所以將起點和其值(0,0)創成 pair 加進 pq，接著執行 while 迴圈，只要不是空的便持續執行(空的代表檢查完了)，設一個 variable p 等於 pq 的 second 也就是 vertex A，設完之後將此 pair pop 掉，接著根據 vertex A，找出所有和它相連的 vertex B，如果 vertex B 在 visited 已經標記為 true，代表已經檢查過了可以跳過，如果是 false，便去檢查 weight 是否小於 vertex B 的 key，如果是的話，更新 vertex 的 parent 為 vertex A，並將 vertex B 的 key 更新為 weight，並將新的 key 和 B 的 index 新增至 pq，如果檢查完所有和 vertex A 相連的 vertex B，將 vertex A 的 visited 標記為 true，等 pq 空了之後，整個 while 也將停止，將 vector key 裡頭所有值加起來便是最小 cost 了。

```

void Graph::Prim(){
    priority_queue<pai,vector<pai>,greater<pai> > pq;
    vector<int> parent(vnum,-1);
    vector<int> key(vnum,INF);
    vector<bool> visited(vnum,false);

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

    while(!pq.empty()){
        int p = pq.top().second;
        pq.pop();

        list<pair<int,int>>::iterator it;
        for(it=lis[p].begin();it!=lis[p].end();it++){
            int vertex = (*it).first;
            int weight = (*it).second;
            if(visited[vertex]==1){
                continue;
            }
            if(weight<key[vertex]){
                parent[vertex] = p;
                key[vertex] = weight;
                pq.push(make_pair(key[vertex],vertex));
            }
        }
        visited[p] = true;
    }
    int cost=0;
    for(int i=0;i<vnum;i++){
        cost += key[i];
    }
    cout << cost;
}

```

4. **Main Function:** 依照題目，會先給定 vertex 和 edge 個數，利用 constructor 創造 Graph g，在將 edge 利用 AddEdge 這個 function 傳入 list，最後執行 Prim 便能得到想要的結果。

```

int main(){
    int vnum,ednum,start,end,weight;
    cin >> vnum >> ednum;
    Graph g(vnum);
    for(int i=0;i<ednum;i++){
        cin >> start >> end >> weight;
        g.AddEdge(start,end,weight);
    }
    g.Prim();
    return 0;
}

```


➤ Kruskal Algorithm

我用了一個 struct 裡面有 start、end、weight 三個變數，加上四個 function。

1. **Compare:** 在 Kruskal function 裡面要 sort，但因為 Edge 是一個 struct，而我要根據裡頭的 weight 來排，所以要寫個 Compare function。

```
bool Compare(Edge a, Edge b){  
    return a.weight < b.weight;  
}
```

2. **Findset:** 傳入一個 i，只要 subset[i] 不等於 i，就讓 i 等於 subset[i] 一直找下去，直到發現源頭為止，找到後回傳 i。

```
int Findset(int *subset, int i){  
    while(subset[i] != i){  
        i = subset[i];  
    }  
    return i;  
}
```

3. **Union:** 傳入的 a 和 b 會有不同的 parent，分別找出他們的 parent，然後讓 a 的 parent 變成 b，這樣所有和 a 或 b 同個集合的 parent 都變成 b 了。

```
void Union(int *subset, int a, int b){  
    int aroot, broot;  
    aroot = Findset(subset, a);  
    broot = Findset(subset, b);  
    subset[aroot] = broot;  
}
```

4. **Kruskal**: subset[vnum]代表每個 vertex 的 parent，一開始先設定為自己，接著將每條 edge 的 start、end、weight 都存進 edge[ednum]，然後依照 weight 進行 sort，進入 for 迴圈後，只要 start 和 end 已經在同個集合裡(parent 相同)，就繼續往下找，不然會導致 cycle，如果找到不一樣的就執行 Union function，將 start 和 end 放進同個集合，然後他們之間的連線就是 weight，將它加進 cost 裡面，等到每個都檢查過，確定大家都在同個集合裡時，印出 cost。

```
void Kruskal(int *subset,int vnum,int ednum){
    int cost=0;
    struct Edge edge[ednum];
    for(int i=0;i<vnum;i++){
        subset[i] = i;
    }

    for(int i=0;i<ednum;i++){
        cin >> edge[i].start >> edge[i].end >> edge[i].weight;
    }

    sort(edge,edge+ednum,Compare);

    int i,j;
    for(i=0,j=0;i<vnum-1 && j<ednum ;i++){
        while(Findset(subset,edge[j].start)==Findset(subset,edge[j].end)){
            j++;
        }
        Union(subset,edge[j].start,edge[j].end);
        cost += edge[j].weight;
    }
    cout << cost;
}
```

5. **Main Function**: 建立 subset[vnum]，接著執行 Kruskal。

```
int main(){
    int vnum,ednum;
    cin >> vnum >> ednum;
    int subset[vnum];
    Kruskal(subset,vnum,ednum);
    return 0;
}
```


[Results]

➤ Prim Algorithm

1. Judging (hackerrank / 助教測資)

Q2 - Prim's Algorithm

Problem

Submissions

Leaderboard

Discussions

Submitted a few seconds ago • Score: 100.00 Status: Accepted

| | | | | | |
|---|--------------|---|--------------|---|--------------|
| ✓ | Test Case #0 | ✓ | Test Case #1 | ✓ | Test Case #2 |
| ✓ | Test Case #3 | ✓ | Test Case #4 | ✓ | Test Case #5 |

```
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
-----
```

```
899 215 781
899 153 7399
899 377 8097
899 202 1864
899 345 2982
899 60 4441
899 45 6655
899 365 3705
899 210 6541
899 92 7171
899 238 3106
899 151 4402
899 277 8474
27570
-----
Process exited
```

```
999 432 4890
999 323 2274
999 491 5325
999 716 6756
999 87 7395
999 602 1762
999 144 3702
999 214 5012
999 141 6763
999 969 8142
999 533 5165
999 784 2981
145978
-----
Process exited
請按任意鍵繼續
```

2. **Time Complexity:** 創造 pq 的時間複雜度為 $O(1)$ ，設定 vector 值和 while 迴圈皆是 $O(V)$ ，dequeue 則是 $O(\log V)$ ，比較每個 vertex 和 weight 的時間為 $O(V)$ ，由此我們可知，完整時間是 $O(1)+O(V)+O(V^2 \log V) = O(E \log V)$

```
int main(){
    int vnum,ednum,start,end,weight;
    clock_t startc,endc;
    cin >> vnum >> ednum;
    Graph g(vnum);
    for(int i=0;i<ednum;i++){
        cin >> start >> end >> weight;
        g.AddEdge(start,end,weight);
    }
    startc = clock();
    g.Prim();
    endc = clock();
    cout << double(start-end)/CLOCKS_PER_SEC;
    return 0;
}
```

```
5 7
0 1 10
0 2 20
1 2 30
1 3 5
2 3 15
2 4 6
3 4 8
```

Your Output (stdout)

```
29
-1e-06
```

Expected Output

➤ Kruskal Algorithm

1. Judging (hackerrank / 助教測資)

Q2 - Prim's Algorithm

Problem

Submissions

Leaderboard

Discussions

Submitted a few seconds ago • Score: 100.00

Status: Accepted

| | | | | | |
|---|--------------|---|--------------|---|--------------|
| ✓ | Test Case #0 | ✓ | Test Case #1 | ✓ | Test Case #2 |
| ✓ | Test Case #3 | ✓ | Test Case #4 | ✓ | Test Case #5 |

```
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 a
請按任意鍵繼續
```

```
399 60 4441
399 45 6655
399 365 3705
399 210 6541
399 92 7171
399 238 3106
399 151 4402
399 277 8474
27570
-----
Process exited af
請按任意鍵繼續
```

```
999 87 7395
999 602 1762
999 144 3702
999 214 5012
999 141 6763
999 969 8142
999 533 5165
999 784 2981
145978
-----
Process exited
請按任意鍵繼續
```

2. **Time Complexity:** 創造 subset 的時間複雜度為 $O(V)$ ，sort weight 則是 $O(E \log E)$ ，將全數 vertex 移到同個 subset 花了 $O(E \cdot V)$ ，由此我們可知，完整時間是 $O(V + E \log E + EV)$ ，也就是 $O(E \log E)$ 。

```
int main(){
    int vnum,ednum;
    clock_t start,end;
    cin >> vnum >> ednum;
    int subset[vnum];
    start = clock();
    Kruskal(subset,vnum,ednum);
    end = clock();
    cout << double(start-end)/CLOCKS_PER_SEC;
    return 0;
}

bool Compare(Edge a,Edge b){
    return a.weight < b.weight;
}

int Findset(int *subset,int i){
    while(subset[i]!=i){
        i = subset[i];
    }
    return i;
}
```

Input (stdin)

```
5 7
0 1 10
0 2 20
1 2 30
1 3 5
2 3 15
2 4 6
3 4 8
```

Your Output (stdout)

```
29
-5e-05
```

[Discussion]

1. **My Discover:** Kruskal Algorithm 的想法好酷喔，竟然想到用集合來做，老師在上課的時候就覺得這想法很新穎了，等到自己實作的時候，我甚至想不到怎麼創集合，上網查才發現可以用 parent 的方法，找到源頭並將它紀錄下來，其中 Findset 和 Union 那些 function 也讓我大開眼界呢！
2. **Comparison of Time Complexity:** Kruskal 的時間複雜度為 $O(E \log E)$ ，Prim 的則是 $O(E \log V)$ ，因此我們可以藉由 E 和 V 的大小來選擇我們要使用何者演算法。上圖執行結果顯示 Kruskal 所需時間大於 Prim，我想是因為 edge 數量超過 vertex，才導致這種結果。
3. **Situation Analysis:** Kruskal 的重點在 edge 上，每次都以 edge 的 weight 為主，直接將他們相加，而 Prim 則在 vertex 上，它會給定 vertex 值，過程中也一直更新 vertex 值，這就是為什麼 Kruskal 的 Time Complexity 為 $O(E \log E)$ ，Prim 的則是 $O(E \log V)$ ，因此我們可以藉由 E 和 V 的大小來選擇我們要使用何者演算法。
4. **Challenges:** MST 和 sorting 比起來難了些，在這部分遇到蠻多困難的，我在打 Prim 的時候，一開始不知道有 priority queue 可以用的時候，一直在想 edge 要怎麼存，要怎麼把它和 node 綁在一起，一開始用了 struct，但過程中還是不知道如何連接，後來知道可以用 priority queue 時候，整個人都順暢了，好開心。在打 Kruskal 的時候，一開始沒花很多時間就打出來了，在 Dev C++ 都很順利，但一丟到 hackerrank 上面的時候，他就說什麼編譯器錯誤，上網查也都沒辦法解決問題，只好直接換一種寫法，結果第二次想超久的，幸好後來也順利通過測試了。

[Conclusion]

1. **Learning:** 在這次作業裡，我對 minimum spanning tree 有了更多認識，知道可以怎麼利用集合在演算法上做些事，也學到如何使用 pair、list、priority queue，他們真的是很厲害的工具，甚至可以結合在一起來用，我想我以後可能會很常用到它，真的好好用喔。
2. **Spending Time:** coding 6-8 小時 / report 3 小時
3. **Feedback:** 我認為 part 2 是我學到最多知識的地方，或許是因為它最難，或是它有很多新奇的工具讓我去學習，反正我個人很喜歡這個 part，它讓我增廣見聞許多。

PART 3: shortest path

[Introduction]

- **Dijkstra's Algorithm:** 以其中一個vertex A為出發點，計算vertex A到另一個vertex B的最短路徑，檢查和vertex A連接且未被選過的vertex，如果vertex A的值加上連接AB的edge值小於vertex B，更改vertex B的值為vertex A的值加上連接AB的edge值，並將vertex B的parent改為vertex A，接著選擇距離vertex A有最短距離的當成下個檢查的對象，重複此動作直到所有vertex都被檢查過為止。
- **Bellman-Ford Algorithm:** 此演算法是檢查每條edge，如果vertex A的值加上連接AB的edge值小於vertex B，更改vertex B的值為vertex A的值加上連接AB的edge值，並將vertex B的parent改為vertex A，執行這個動作(vnum-1)次，便可以得到最短路徑，再多執行一次的話，可以檢查有沒有negative loop。

[Implement Details]

- **Dijkstra's Algorithm:** 這個演算法和Prim極為相似，一樣運用了class和裏頭的兩個function。
 1. **Class / Constructor:** class有兩個function在public，private 則有一個variable vnum，是指vertex個數，並且有個list，型態則是<int,int>的pair，constructor傳入vertex的個數，接著創造一個graph，裡頭的vnum是傳入的vertex個數，list的大小也是依照vnum來決定。

```
class Graph{
public:
    Graph(int vnum);
    void AddEdge(int s,int e,int w);
    void Dijkstra(int start,int target);
private:
    int vnum;
    list<pai> *lis;
};

Graph::Graph(int vnum){
    this->vnum = vnum;
    lis = new list<pai>[this->vnum];
}
```

2. **AddEdge:** input 會傳入 edge 頭尾的 vertex 值和 edge 的 weight，因為題目說有方向性，所以不像 Prim 雙向都要存，只要存單向就好，將 end 和 weight 創造 pair 並存進 start 的 list。

```
void Graph::AddEdge(int s,int e,int w){
    lis[s].push_back(make_pair(e,w));
}
```

3. **Dijkstra:** 首先一樣先創造一個由小排到大的priority queue叫pq，，除此之外，還要兩個大小都是vnum的vector，第一個vector用來存distance，初始化起點為0，其他則為無限大(前面有define)，parent則用來記錄他們的parent是誰，初始化所有值為-1。因為題目規定從零開始，所以將起點和其值(0,0)創成pair加進pq，接著執行while迴圈，只要不是空的便持續執行(空的代表檢查完了)，設一個variable p等於pq的second也就是vertex A，設定完之後將此pair pop掉，接著根據vertex A，找出所有和它相連的vertex B，去檢查vertex A加上weight是否小於vertex B，如果是，更新vertex B的parent為vertex A，並將vertex B的distance更新為weight，並將新的distance和B的index新增至pq，等pq空了之後，整個while也將停止，將vector distance裡頭所有值加起來便是最小cost了。

```
void Graph::Dijkstra(int start,int target){
    priority_queue<pai,vector<pai>,greater<pai>> pq;
    vector<int> distance(vnum,INF);
    vector<int> parent(vnum,-1);
    distance[start] = 0;
    pq.push(make_pair(0,start));

    while(!pq.empty()){
        int p = pq.top().second;
        pq.pop();

        list<pai>::iterator it;
        for(it=lis[p].begin();it!=lis[p].end();it++){
            int vertex = (*it).first;
            int weight = (*it).second;

            if(distance[vertex]>distance[p]+weight){
                parent[vertex] = p;
                distance[vertex] = distance[p]+weight;
                pq.push(make_pair(distance[vertex],vertex));
            }
        }
    }
    cout << distance[target] << endl;
}
```

4. **Main Function:** 因為這有方向性的問題，所以除了vnum和ednum，還要cin起點和終點，接著創造一個Graph g，將所有edge利用AddEdge()存進list，接著執行Dijkstra()。

```
int main(){
    int vnum,ednum,start,target,from,to,weight;
    cin >> vnum >> ednum >> start >> target;
    Graph g(vnum);
    for(int i=0;i<ednum;i++){
        cin >> from >> to >> weight;
        g.AddEdge(from,to,weight);
    }
    g.Dijkstra(start,target);
    return 0;
}
```

- **Bellman-Ford Algorithm:** 利用struct來輸入edge，使用Graph和public裡頭的兩個function。

1. **Class / Constructor:** class有兩個function在public，private有兩個variable，vnum和ednum，還有個型態是Edge的vector，constructor傳入vertex和edge的個數，接著創造一個graph。

```
class Graph{
public:
    Graph(int vnum,int ednum);
    void AddEdge(int s,int e,int w);
    void BellmanFord(int start,int target);
private:
    int vnum,ednum;
    vector<Edge> edge;
};

Graph::Graph(int vnum,int ednum){
    this->vnum = vnum;
    this->ednum = ednum;
    vector<Edge> edge;
}
```

2. **AddEdge:** input 會傳入 edge 頭尾的 vertex 值和 edge 的 weight，將他們加入 vector 中。

```
void Graph::AddEdge(int s,int e,int w){
    Edge ed = {s,e,w};
    edge.push_back(ed);
}
```


3. **BellmanFord**: 創造 parent 和 distance 兩個 vector，一樣初始化 distance 為無限大(起點為 0)，parent 則初始化為 -1，接著檢查每條 edge，只要 end 的值大於 start 加上 weight 的值，更新 end 的值為 start 加上 weight 的值，並將 end 的 parent 改為 start，此步驟須執行 vnum-1 次，就可以得到最短路徑，最後再檢查一次，如果有發現 end 的值大於 start 加上 weight 的值，代表有 negative loop，就印出 Negative loop detected!，如果沒有就印出 target 的 distance 值。

```
void Graph::BellmanFord(int start,int target){
    vector<int> distance(vnum,INF);
    vector<int> parent(vnum,-1);
    distance[start] = 0;

    for(int i=0;i<vnum-1;i++){
        for(int j=0;j<ednum;j++){
            if(distance[edge[j].end]>distance[edge[j].start]+edge[j].weight){
                distance[edge[j].end] = distance[edge[j].start]+edge[j].weight;
                parent[edge[j].end] = edge[j].start;
            }
        }
    }

    for(int j=0;j<ednum;j++){
        if(distance[edge[j].end]>distance[edge[j].start]+edge[j].weight){
            cout << "Negative loop detected!";
            return;
        }
    }
    cout << distance[target];
}
```

[Results]

➤ Dijkstra's Algorithm

1. Judging (hackerrank / 助教測資)

Q3 - Dijkstra's Algorithm

Problem

Submissions

Leaderboard

Discussions

Submitted a few seconds ago • Score: 100.00

Status: Accepted

✓

Test Case #0

✓

Test Case #1

✓

Test Case #2

✓

Test Case #3

✓

Test Case #4

✓

Test Case #5


```

156 402 5888
206 158 622
61 302 4906
296 378 2947
221 171 3266
466 481 8869
160 295 515
42656
-----
Process exited
請按任意鍵繼續

```

```

885 228 9200
245 657 4988
420 733 7072
95 232 4268
449 339 4582
884 498 628
460 77 6813
584 633 5487
166 459 2535
54364
-----
Process exited
請按任意鍵繼續

```

```

717 700 5170
152 35 1399
323 963 6802
89 594 5213
676 168 8956
325 0 5164
618 876 2304
39995
-----
Process exited
請按任意鍵繼續

```

2. **Time Complexity:** 設vector值和while的時間都是 $O(V)$ ，dequeue則是 $O(\log V)$ ，比較每個vertex和weight的時間為 $O(V)$ ，由此我們一共需要的時間是 $O(V^2)$ 。

```

int main(){
    int vnum,ednum,start,target,from,to,weight;
    clock_t a,b;
    cin >> vnum >> ednum >> start >> target;
    Graph g(vnum);
    for(int i=0;i<ednum;i++){
        cin >> from >> to >> weight;
        g.AddEdge(from,to,weight);
    }
    a = clock();
    g.Dijkstra(start,target);
    b = clock();
    cout << double(a-b)/CLOCKS_PER_SEC;
    return 0;
}

```

Input (stdin)

```

5 8
4 0
1 0 3
2 3 2
4 3 2
3 1 1
0 2 6
4 1 4
3 2 1
0 3 6

```

Your Output (stdout)

```

-6e-06

```

➤ Bellman-Ford Algorithm

1. Judging (hackerrank / 助教測資)

Q3 - Bellman-Ford Algorithm

| Problem | Submissions | Leaderboard | Discussions |
|---|----------------|----------------|-------------|
| Submitted a few seconds ago • Score: 100.00 | | | |
| Status: Accepted | | | |
| ✓ Test Case #0 | ✓ Test Case #1 | ✓ Test Case #2 | |
| ✓ Test Case #3 | ✓ Test Case #4 | ✓ Test Case #5 | |

```

408 33 8980
193 456 9108
246 213 -147
457 43 4878
190 388 5487
137 25 6235
478 134 4589
220 483 6744
20157
-----
Process exited
請按任意鍵繼續

```

```

312 294 9295
358 396 9448
949 667 5937
579 135 8605
880 419 8243
477 736 7652
315 287 5625
126 167 3228
24585
-----
Process exited
請按任意鍵繼續

```

```

706 943 9527
74 605 565
276 708 2560
99 42 3471
60 418 2686
248 566 9396
88 152 732
Negative loop detected!
-----
Process exited after 7.3
請按任意鍵繼續 . . .

```

2. **Time Complexity:** 設vector值是 $O(V)$ ，比較每個edge中end和start + weight的時間為 $O(E)$ ，檢查negative loop的時間為 $O(E)$ ，由此我們一共需要的時間是 $O(VE+E)=O(VE)$ 。

```

int main(){
    int vnum,ednum,start,target,from,to,weight;
    clock_t a,b;
    cin >> vnum >> ednum >> start >> target;
    Graph g(vnum,ednum);
    for(int i=0;i<ednum;i++){
        cin >> from >> to >> weight;
        g.AddEdge(from,to,weight);
    }
    a = clock();
    g.BellmanFord(start,target);
    b = clock();
    cout << double(a-b)/CLOCKS_PER_SEC;
    return 0;
}

```

Input (stdin)

```

5 8
4 0
1 0 3
2 3 2
4 3 2
3 1 1
0 2 6
4 1 4
3 2 1
0 3 -6

```

Your Output (stdout)

```

-3e-06

```

[Discussion]

1. **My Discover:** minimum spanning tree 和 shortest path 其實蠻相像的，只是 shortest path 會有一個要走到的目標，而 minimum spanning tree 是要將所有 vertex 值相加。
2. **Negative Loop Detection:** 一開始執行檢查(vnum-1)次，可以得到最短路徑，如果此時再多執行一次的話，所有 vertex 值不應該改變，如果發現 end 的值大於 start 加上 weight 的值，代表有 negative loop，如果有 negative loop，會一直越來越小，而沒有最短路徑存在。

3. **Print the Path:** 起點的 parent 永遠是 -1，過程中有將所有的 parent 記下來，根據這個 vector，可以一直往下找，直到找到 -1，也就是起點，但因為會從終點往回找，於是我將箭頭反過來印。無論是 Dijkstra's Algorithm 或是 Bellman-Ford Algorithm 都有個 vector 叫 parent，要印出 path 的步驟也一模一樣。

```
int i = target;
cout << i;
while(parent[i]!=-1){
    i = parent[i];
    cout << " <- " << i;
}
```

```
5 8
4 0
1 0 3
2 3 2
4 3 2
3 1 1
0 2 6
4 1 4
3 2 1
0 3 6
6
0 <- 1 <- 3 <- 4
```

4. **Comparison of Time Complexity:** Dijkstra 的時間複雜度為 $O(V^2)$ ，Bellman-Ford 的則是 $O(EV)$ ，Time Complexity 那邊的圖顯示 Bellman-Ford 所需時間大於 Dijkstra，我想是因為 edge 數量超過 vertex，才導致這種結果。
5. **Challenges:** 因為前面先寫過 minimum spanning tree 了，所以這部份就變得沒那麼困難了，一下就完成了。

[Conclusion]

1. **Learning:** 前面忘記講了，我在這次作業還學到用 clock 測時間，在查資料的同時，甚至還查到了另外兩種方法，以前都不知道還有這種功能。在 part 3 則學到了如何利用 parent 找出那一整條路徑。
2. **Spending Time:** coding 2-3 小時 / report 2 小時
3. **Feedback:** 我依然認為這是一個很好的作業，難度不會太難，學到很多新東西，相信在以後對打扣一定能更加得心應手，