

Part I: Sort Algorithm

I-1. Introduction

以下將介紹兩種熱門的 sorting algorithm : merge sort 與 quick sort，這兩者在所有排序演算法中，擁有優秀的平均時間複雜度(average time complexity)為 $O(n\log n)$ ，然而這兩者各有優缺，下面進行詳細介紹。

I-2. Implement Details

I-2-1. Quick Sort

圖 1 為 quick sort 的程式碼，此演算法採用了分治法(divide-and-conquer algorithm)，會將傳入 array 拆成兩個 subarray，再分別對兩個 subarray 進行遞迴排序；quicksort function 中可分成四大部分：遞迴出口、選擇基準(pivot)、根據基準劃分兩個 subarray、將 subarray 進行遞迴。

function 傳入值中，被排序的 array 以 pass-by-reference 的方式傳入，兩個 int 參數 left, right 分別代表被排序的 array index 範圍，最小值是 left，最大值是 right，可想像成被排序 array 的最左端與最右端，因此在最初呼叫 quicksort function 時， $left = 0$ 、 $right = n - 1$ 、 n 為 array size；當 $left$ 等於 $right$ 時，代表排序範圍只有 1 個 element，無需進行排序，若 $left > right$ 則是無效的 index 範圍，因此以 $left \geq right$ 作為遞迴出口條件；選擇 pivot 上，這裡是以排序範圍 index 的中間值所對應的 element 作為 pivot；partition 上，會將排序範圍內 $element \geq pivot$ 放在 array 右半部， $element \leq pivot$ 放在左半部，操作上使用了指針 i 與指針 j 分別從排序範圍的最左與最右端往中心掃描，在兩指針相遇之前，左指針遇到 $array[i] \geq pivot$ 則停下來，右指針遇到 $array[j] \leq$

```
void quicksort(vector<int> &array, int left, int right) {
    if (left >= right)
        return; // Exit of recursive function

    int i = left, j = right;
    int pivot = array[(i + j) / 2]; // Select pivot

    while (i <= j) {
        while (i <= j && array[i] < pivot)
            i++;
        while (i <= j && array[j] > pivot)
            j--;
        if (i <= j) {
            int temp = array[i];
            array[i] = array[j];
            array[j] = temp;
            i++; j--;
        }
    }

    quicksort(array, left, j); // Partition array into two subarrays
    quicksort(array, i, right); // Sort two subarray recursively
}
```

圖 1. Quick sort 程式碼

pivot 則停下來，接著交換兩指針對應的 element，交換完後指針繼續前進，重複以上動作直到兩指針超過彼此；partition 結束後，(left, j) 和 (i, right) 分別代表兩個 subarray 的 index 範圍，再將這兩個 index 範圍傳入 quicksort function。

I-2-2. Merge Sort

圖 2 為 merge sort 的程式碼，與 quick sort 同樣採用了分治法，會將傳入 array 拆成兩個 subarray 並分別進行排序，最後再將兩個排序好的 subarray 合併為一排序 array；程式碼第一部分為遞迴出口，第二部分將 array 等分成兩個 subarray 進行遞迴，第三部分將兩個 subarray 所有 element 合併排序成一個 array。

傳入 mergesort function 的參數比起 quicksort function 多了一個 temp array，用在合併排序 subarray；由於 merge sort 最後會將所有 subarray 合併回一開始傳入的 array，因此 temp array 的 size 要和一開始傳入的 array 的 size 一樣大；temp array 以 pass-by-reference 的方式傳入 mergesort function，就不用每次遞迴 mergesort function 時都要重新生成 temp array；參數 left, right 為本次 mergesort function 需排序 array 的 index 範圍，在最初呼叫 mergesort function 時，left = 0、right = n - 1、n 為 array size；第一部分遞迴出口的條件與 quicksort function 相同，使用 left >= right 判斷本次 index 範圍是否只含有一個 element，或是傳入了無效的 index 範圍；第二部分會將本次傳入 array 對切成兩個 subarray 再遞迴傳入 mergesort function，遞迴完成後兩個 subarray 也排序完成；第三部份中，我們以三個指針 i, j, index 用於掃描定位兩個 subarray 和 temp array，指針均由 array 左端往右端移動；第一個 while 迴圈會將當前 i, j 指到的 element 較小者存放至 temp array，並將 index 指針往右移動一格，若為 i 指針的 element 被選取則 i 向右移動一格，

```
void mergesort(vector<int> &array, vector<int> &temp, int left, int right) {
    if (left >= right)
        return; Exit of recursive function

    int mid = (left + right) / 2;
    mergesort(array, temp, left, mid); Sort two subarray recursively
    mergesort(array, temp, mid + 1, right);

    int i = left, j = mid + 1;
    int index = left;
    while (i <= mid && j <= right) {
        if (array[i] <= array[j])
            temp[index++] = array[i++];
        else
            temp[index++] = array[j++];
    }
    while (i <= mid)
        temp[index++] = array[i++];
    while (j <= right)
        temp[index++] = array[j++];

    for (int k = left; k <= right; k++)
        array[k] = temp[k];
}
```

Merge two sorted subarrays
into one sorted array

圖 2. Merge sort 程式碼

若為 j 則 j 向右移動一格；第二和第三個 while 迴圈均處理有一個 subarray 中所有 element 均被挑選至 temp array，而另一個 subarray 有剩餘 element 尚未被挑選，則將這些 element 依序排至 temp array；最後，將 temp array 中 index 範圍從 left 至 right 所有 element 依序取出，填入 input array，index 範圍同樣對應 left 至 right，即完成 merge sort。

I-3. Results & Discussion

I-3-1. Time & Space Complexity Analysis

➤ Quick sort

Quick sort 的排序核心是將指定 index 範圍中所有 array element 與 pivot 進行大小比較，分別放至兩個 array 兩側，從圖 1 可以看到，element 分群是以 in-place swap 進行，僅需一個 int 變數作為 swap buffer，在不考慮 recursive call 所使用的記憶體情況下，可以說 quick sort 的 space complexity 為 $O(1)$ ；在 time complexity 上，pivot 的選擇會影響分割後兩個 subarray 的 size，而兩個 subarray 的 size 是否平衡則會影響 quick sort 的 time complexity。

Time complexity 的最差情況發生於每次 pivot 剛好挑到傳入 array 的最大值或最小值，會使傳入 array 在分割後得到的兩個 subarray 大小分別為 $n - 1$ 與 1 ， n 為傳入 array 的大小；這裡假設 subarray A 大小為 $n - 1$ ，subarray B 大小為 1 ，兩者進行遞迴，subarray B 達到遞迴出口，subarray A 被分割成 subarray C 與 subarray D，大小分別為 $n - 2$ 與 1 ，subarray C 與 subarray D 再繼續遞迴，subarray D 達到遞迴出口，subarray C 再分割成 subarray E 與 subarray F，大小分別為 $n - 3$ 與 1 ...不段重複上述遞迴直到兩個 subarray 大小均為 1 ；分析上述 time complexity 我們可以得到為 $O(n) + O(n - 1) + O(n - 2) + \dots + O(1) = O(n * (n + 1) / 2) = O(n^2)$ ，因此可得 quick sort 最差情況下 time complexity 為 $O(n^2)$ 。

Time complexity 的最佳情況則發生於在每次遞迴 quicksort function 時，所挑選之 pivot 均能讓傳入 array 分割成兩個相同大小的 subarray，因此我們可以得到最大遞迴深度會是 $\log_2 n$ ；綜合所有擁有相同遞迴深度的 function call，可以發現這些 function call 所涵蓋的 index 範圍剛好是最一開始傳入 array 的整個 index 範圍，因此總和各別 function call 進行分割所花時間複雜度剛好等於 $O(n)$ ，因此總共時間複雜度 = 遞迴深度 * 相同遞迴深度之 function calls 執行總時間複雜度 = $O(\log n) * O(n) = O(n \log n)$ ，得證 quick sort 在最佳情況下 time complexity 為 $O(n \log n)$ 。

至於 Time complexity 的平均情況，由於 array 中每個 element 被挑選為 pivot 的機率應該都是相同的，因此我們可以說，有 50% 機率 pivot 挑選了所有 array element 在大小排序上第 25 百分位數到第 75 百分位數之間，如圖 3 中的綠色區域，另外 50% 機率則挑選了圖 3 中的藍色區域；我們先假設每次 pivot 都挑選了圖 3 中的綠色區域，在最差的情況下，pivot 都挑中了第 25 或 75 百分位

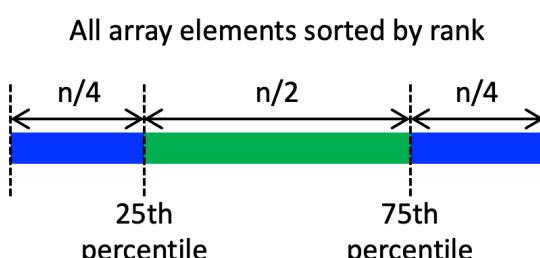


圖 3. Input array element 就大小排序分佈示意圖

數，如此會造成每次分割所產生兩個 subarray 的 size 比例為 3:1，我們將 quicksort function 遞迴執行的情形以圖 4 表示，遞迴出口與 pivot 挑選都可視作 $O(1)$ 時間，left child 的 array size 為 root 的 $1/4$ ，right child 的 array size 為 root 的 $3/4$ ；前面提到遞迴的終點為 subarray size = 1，因此左子樹會較快達到遞迴終點，所對應的遞迴次數為 $\log_4 n$ ，右子樹較慢，對應次數為 $\log_{3/4} n$ ；從圖 4 我們可以得到，起初 $\log_4 n$ 的遞迴深度每層均掃描了 n 個 element 來進行 partition，假設掃描一個 element 所花時間為 c ，一層總共掃描時間為 $c * n$ ，而當遞迴深度超過 $\log_4 n$ 時，有些 subarray 達到了遞迴終點，每層掃描總 element 數會少於 n 個，因此我們能說每層 partition 所使用時間不會超過 $c * n$ ；總結上述，在 pivot 每次挑了 array 中第 25 到 75 百分位數之間的 element，最差的時間複雜度為 $O(n * \log_{3/4} n) = O(n * \log_2 n / \log_2(3/4)) = O(n \log n)$ 。

接著我們考慮，實際上每次 pivot 挑選不會只挑到圖 3 綠色區域，也會挑到圖 3 藍色區域，且兩區域被挑選機率均為 50%，因此，若綠色區域被挑選了 k 次，平均來說藍色區域也會被挑選 k 次，可推得執行時間不會超過上段計算時間的兩倍，即 $O(2 * n * \log_{3/4} n)$ ，綜合可得 [quick sort 在平均情況下 time complexity 為 \$O\(n \log n\)\$](#) 。

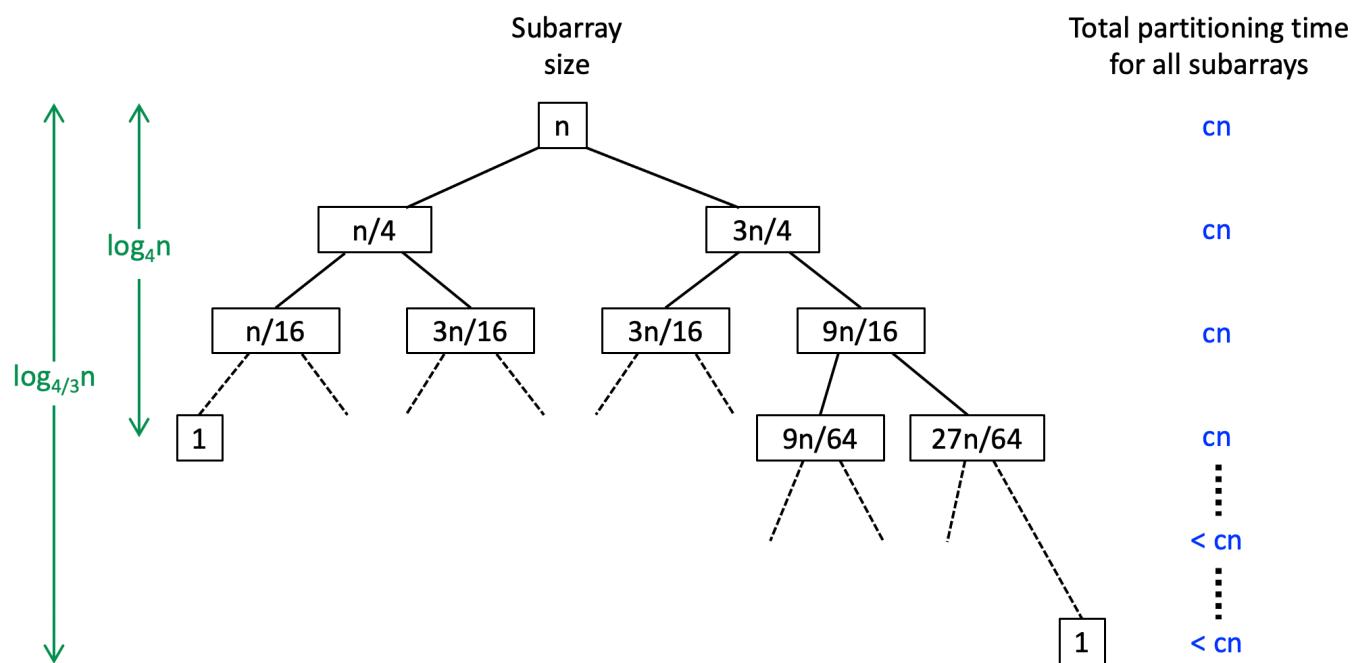


圖 4. 在每次 pivot 都挑選了 array element 第 25 或第 75 百分位數下遞迴執行示意圖

➤ [Merge sort](#)

Merge sort 的排序核心是將兩個等 size(絕大多數情況下的) subarray 合併排序成一個 array，需要準備一個與 input array 同樣大小的暫存空間，因此同樣在不考慮 recursive call 所佔用的記憶體空間下，[merge sort 的 space complexity 為 \$O\(n\)\$](#) ；merge sort 相對於 quick sort，不會有因挑選 pivot 不好而影響時間效率，mergesort function 在傳入 array 的 size 大於 1 的情況下，必先將 array 進行等分並 recursively sort，最後將兩個 subarray 進行合併，因此[其最差、最佳與平均時間效率都是一樣的](#)；圖 5 呈現了 mergesort function 遞迴執行的情形，遞迴出口與選擇對切 array 的 index 位置都當 $O(1)$ ；由於遞迴出口是在 array size = 1 的時候，因此可以算出遞迴深度為 $\log_2 n$ ；當 array size 大於 1 時，mergesort function 最後會進行合併排序，這裡假設每個 element 在合併排序時所有 copy & assignment 所消耗時間為 c ，從圖 5 我們可以觀察到，不同遞迴深度所消耗之

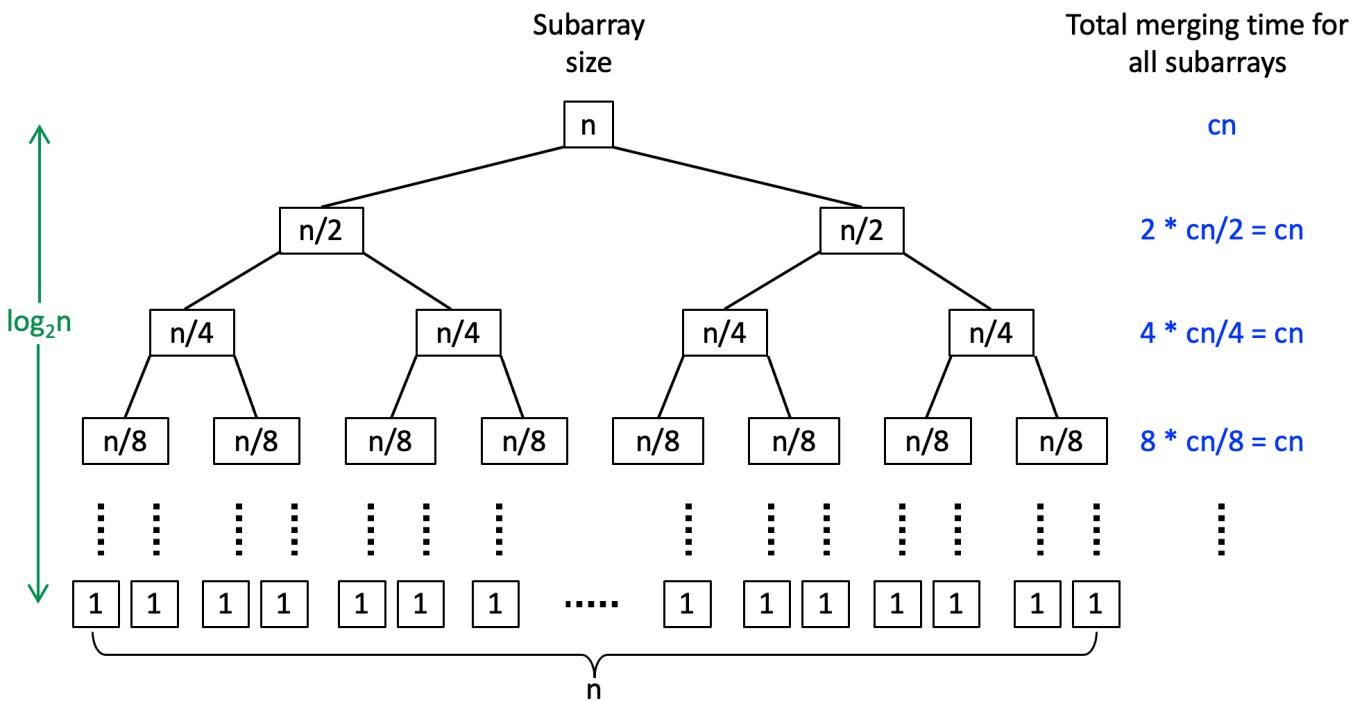


圖 5. Merge sort 遞迴執行示意圖

總共合併排序時間均為 $c * n$ ，因此 Merge sort 總執行時間為 $O(cn * \log_2 n) = O(n \log n)$ ，得到 merge sort 的 time complexity 為 $O(n \log n)$ 。

I-3-2. Stability

若輸入 array 有值重複的 element，在經過某排序演算法後，這些值重複的 element 的排序相對順序與輸入時的排序相對順序相同時，我們稱此演算法能進行 stable sort。不相同則稱為 unstable sort。stability 的影響可用以下例子說明：我們在玩大老二的時候，可能希望自己手上的撲克牌能先以花色排序，相同花色再以數字排序。假設某演算法能達到 stable sort，我們就可以先用它先對牌的

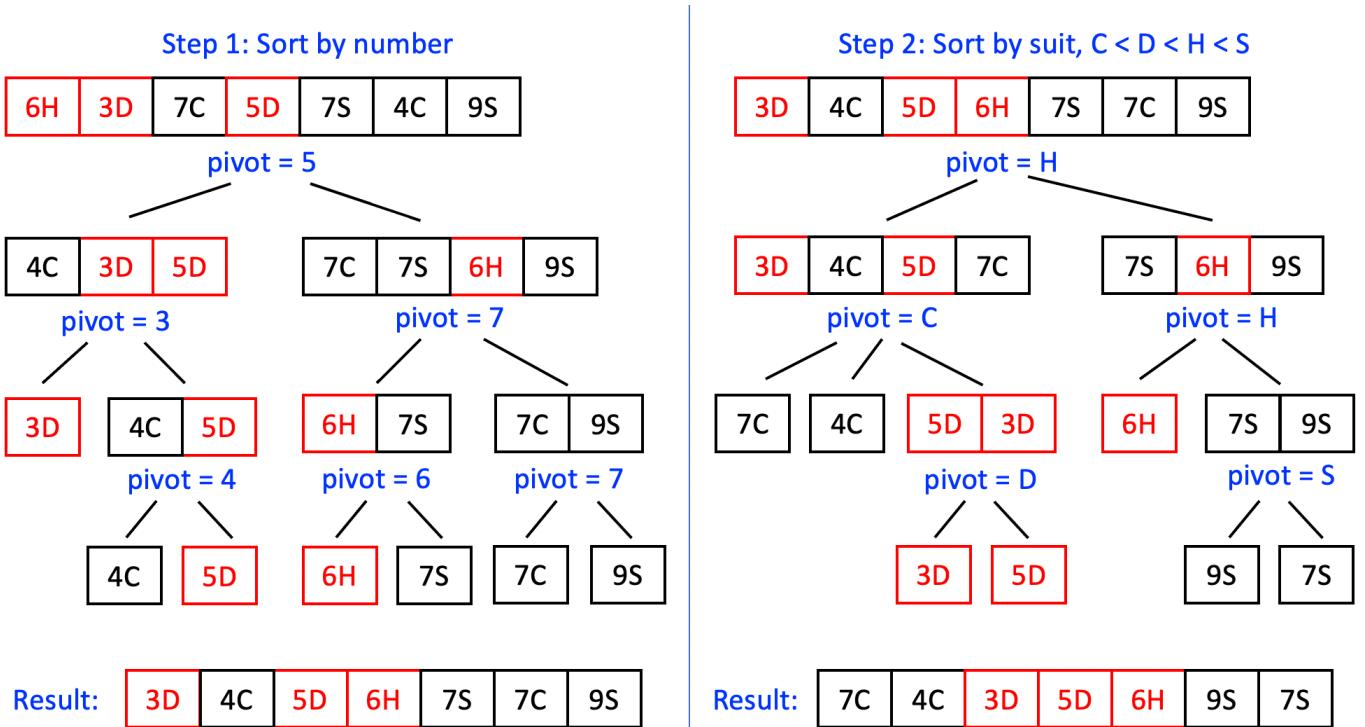


圖 6. 以 quick sort 對撲克牌先進行數值排序再進行花色排序

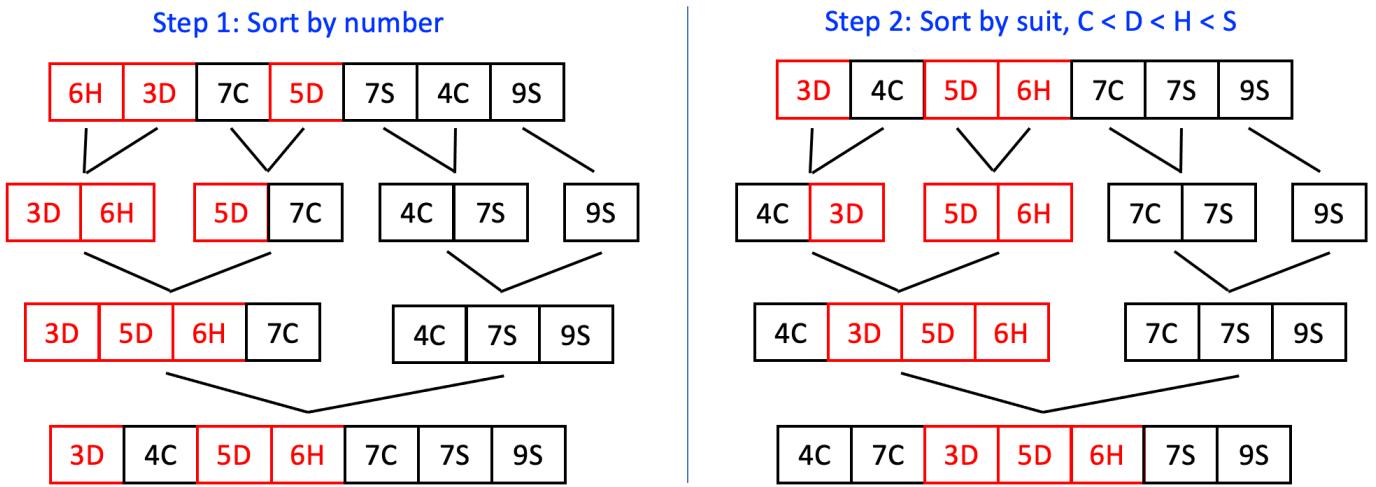


圖 7. 以 merge sort 對撲克牌先進行數字排序再進行花色排序

數字排序，再使用同一個演算法對花色進行排序，就能達到我們的目標；圖 6 與圖 7 分別展示了使用 quick sort 和 merge sort 對撲克牌先進行數值排序再進行花色排序，從結果可以看出 merge sort 可以提供 stable sort 而 quick Sort 不行，從排序過程我們能簡易看出原因：quick sort 在進行 partition 時對 array element 進行 swap 會改變 element 相對順序，而 merge sort 在進行合併排序時一直保持由左至右取出 element，因此能保持相對順序。

I-3-3. Execution time

這裡使用了<ctime>量測了 quick sort 和 merge sort 的程式執行時間，使用的 source code 分別於圖 1 與圖 2 所示，不同 input array size 與不同 sorting algorithm 執行時間呈現於圖 8；在選定一個 input array size n 後，會產生一個 0 到 n-1 組成的隨機數列，且數列中每個數字均不重複，來作為 input array，接著會去量測 quick sort 和 merge sort 對這個 input array 排序時間，每個 input array size 會做 100 次並取平均，因此圖 8 中呈現的是兩種演算法的平均執行時間；從圖 8 中可以看到兩種演算法均趨近於線性增加，但看實際數據，當 input array size 倍增時，執行時間成長了比 2 倍多一點點，由此可粗略驗證兩種演算法的平均時間複雜度為 O(n log n)；最後可以看到在相同 input array size 下，merge sort 的平均執行時間比 quick sort 的還大，這裡推測是 merge sort 在相同遞迴深度下，所有 input array element 都會被複製到 temp array 再複製回 input array，而

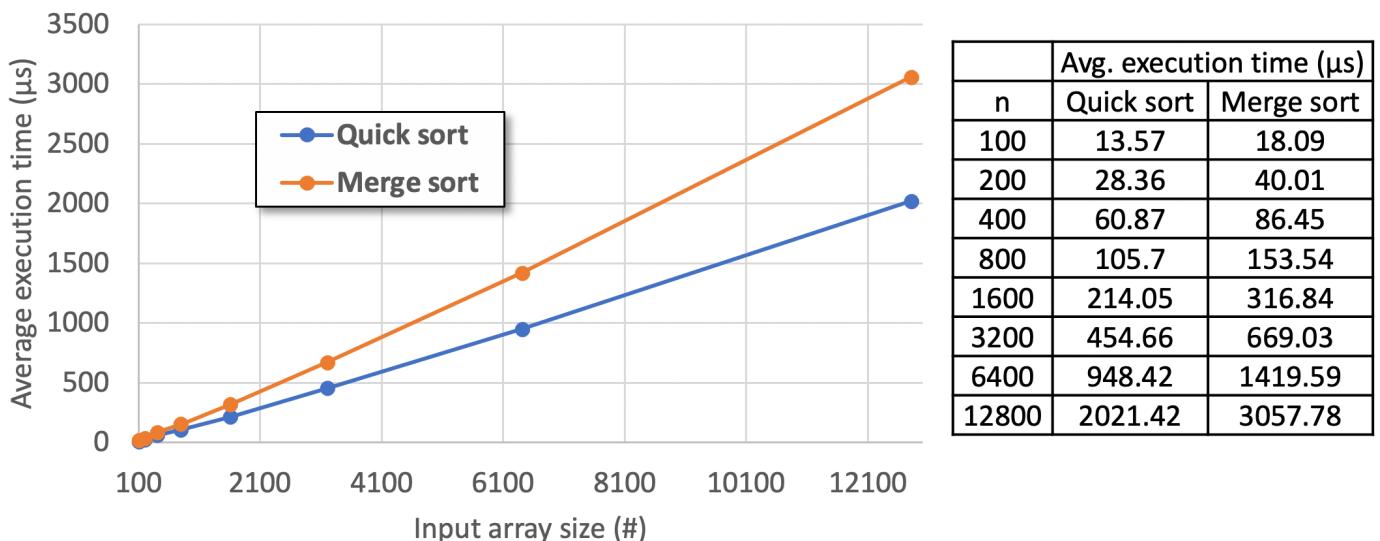


圖 8. Quick sort 和 merge sort 在不同 input array size 下的平均執行時間比較

quick sort 的 element swap 並不一定會對所有 input array element 執行，因此 merge sort 在操作 array element 所花費的時間比 quick sort 還多。

I-4. Conclusion

quick sort 與 merge sort 擁有優秀的平均時間複雜度 $O(n\log n)$ ，quick sort 的時間複雜度會取決於 pivot 的挑選，在每次 pivot 挑選都能將 array 平衡分割下可達到最佳時間複雜度 $O(n\log n)$ ，而每次都是最不平衡分割下時間複雜度會是 $O(n^2)$ 。Merge sort 的最佳與最差時間複雜度都是 $O(n\log n)$ ；空間複雜度上，不考慮 recursive call 所使用的空間下，quick sort 的 in-place swap 特性使空間複雜度為 $O(1)$ ，merge sort 則需要與 input array 等大額外空間提供 subarray 合併排序，因此空間複雜度為 $O(n)$ ，因此若希望節省空間資源，應使用 quick sort；Stability 上，quick sort 不能提供 stable sort，而 merge sort 可以提供 stable sort，因此若希望保持 input array 重複 element 之間的相對順序，應使用 merge sort；而實際程式執行結果顯示，在相同 input array 下，quick sort 比起 merge 花費更少執行時間，因此若希望用最少執行時間應使用 quick sort。

Part II: Minimum Spanning Tree Algorithm

II-1. Introduction

最小生成樹演算法可將一個無向圖(undirected graph)產生一組無向邊(edge)集合，這組邊集合能將所有圖中節點(vertex)串連在一起，不會生成環(cycle)而且邊集合的權重(weight)總和會是最小；最小生成樹可應用在許多地方，舉例來說，電信公司想以最低成本在一個社區架設電信網路，此時能把每一棟房子當成節點，房子間或許能以電纜連結但每個連結路線有其成本，因此電信公司可以計算最小生成樹來決定如何架設電纜來達到最低成本；下面將介紹兩種經典最小生成樹演算法 Kruskal's Algorithm 和 Prim's Algorithm。

II-2. Implement Details

II-2-1. Prim's Algorithm

Prim's Algorithm 使用的是貪心法(greedy algorithm)，會從起始點並向鄰近點不段擴張，擴張路線會選擇能讓點的權重達到最小，流程如下：1. 選擇起始點並讓其權重為 0，剩下節點權重均設定為無限大；2. 選擇當前最小權重節點為目前節點，遍歷與其連結之相鄰節點；3. 相鄰節點未拜訪且相鄰節點權重大於邊的權重，更新相鄰點的權重為邊的權重，相鄰點之父節點為目前節點；4. 將目前節點設為已拜訪；重複步驟 2 到 4，直到所有節點均拜訪。

Prim's Algorithm 的程式碼如圖 8 所示；節點使用了 struct 建構，在生成時會讓 weight 等於 INT_MAX 來使節點一開始權重為無限大，節點還加入了變數 visited 來記錄是否訪問過，如此可以確保不會生成環，所有節點存放在 vector，index 即對應節點編號；邊的資訊使用 nested vector 存放，例如(start vertex, end vertex, edge cost)就會以 vertices[start vertex] = (end vertex, edge cost)方式存下，由於是無向圖，還要多存一組 vertices[end vertex] = (start vertex, edge cost)；為了達到步驟 2 中選擇當前權重最小節點，這裡使用了 priority queue，並使用了 greater function 使 queue 的 top 能得到最小值，而步驟 3 中更新的節點以(vertex weight, vertex index)的格式存入

queue 以對應 greater function 的排序；圖 8 中我們將起點節點設定在 index = 0；演算法執行完成後，若我們想知道最小生成樹所有邊總和權重，將所有節點權重加起來就能得到，而節點間的連結關係可以透過遍歷每個節點的 parent 得到。

```
struct Vertex {
    int index, weight;
    bool visited;
    Vertex* parent;
    Vertex (int i): index(i), weight(INT_MAX), visited(false), parent(nullptr) {}
};

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

```

vector<vector<iPair>> edges;           ----- Input graph
vector<Vertex> vertices;               ----- Set the weight of all vertices as infinite O(V)
priority_queue<iPair, vector<iPair>, greater<iPair>> pq; ----- Create min priority queue O(1)
vertices[0].weight = 0;                  ----- Set vertex 0 as start vertex, weight = 0 O(1)
pq.push(make_pair(vertices[0].weight, 0)); ----- & push into queue O(1)

while (!pq.empty()) {                   ----- O(V)
    v1 = pq.top().second;              ----- Pop min vertex from queue O(log V)
    pq.pop();                         ----- O(log V)

    if (vertices[v1].visited) continue; ----- Check current vertex is visited or not, O(1)
    vertices[v1].visited = true;

    for (iPair &edge : edges[v1]) {      ----- Get every adjacent vertex of current vertex O(V)
        v2 = edge.first; cost = edge.second; ----- O(VlogV)
        if (!vertices[v2].visited && vertices[v2].weight > cost) { ----- O(1)
            vertices[v2].weight = cost;          ----- O(1)
            vertices[v2].parent = &vertices[v1];   ----- O(log V)
            pq.push(make_pair(vertices[v2].weight, v2)); ----- Update adjacent vertex & push into queue O(1 + log V)
        }
    }
}
}                                         ----- Time complexity: O(V + ElogV) → O(ElogV)

```

圖 8. Prim's Algorithm 程式碼與各行對應之時間複雜度

Kruskal's Algorithm 同樣是使用貪心法，其核心是不斷尋找最小權重的邊且不會造成環，流程如下：1. 將每個節點各自形成一個併查集(disjoint set)；2. 將每條邊依權重由小到大排序；3. 將邊依序挑出，若邊的兩個節點均屬於不同併查集，則將兩節點所屬併查集進行合併，並將邊的權重算入最小生成樹邊總權重。

Kruskal's Algorithm 的程式碼如圖 9 所示；edge 使用 struct 建構存取節點與權重資訊，由於會對 edge 以權重由小到大排序，因此會將所有 edge 都存放在 vector 中並使用 std::sort()，排序設定使用了 lambda 運算式並設定以邊的權重排序；節點併查集使用了 UnionFind 這個 class 實作，每個節點以 struct 構成，變數 parent 決定了此節點與哪個節點組成併查集森林(disjoint-set forest)，變數 rank 會用來使併查集森林的構成高度平衡，優化查詢時間；節點們存放在 vector，以 index 對應不同節點編號；在 int UnionFind::Find(int i) 中，傳入參數為節點編號，回傳此節點所在併查集森

```

struct Edge {
    int vertex1, vertex2, cost;
    Edge(int a, int b, int c)
        : vertex1(a), vertex2(b), cost(c) {}
};

struct Vertex {
    int parent, rank;
    Vertex(int i): parent(i), rank(0) {}
};

class UnionFind {
private:
    vector<Vertex> vertices;
public:
    UnionFind(int n) {
        for (int i = 0; i < n; i++)
            vertices.push_back(Vertex(i));
    }
    int Find(int);
    void Union(int, int);
};

UnionFind uf(numberOfVertices);           Make set for each vertex O(V)
vector<Edge> edges;                     Put all edges into container O(E)
sort(edges.begin(), edges.end(), [](Edge& e1, Edge& e2) { return e1.cost < e2.cost; }); Sort edges by edge's cost O(ElogE)

int totalCost = 0, i_edges = 0;
while (i_res < numberOfVertices - 1 && i_edges < numberOfEdges) {           Check every edge until
    Edge& cur = edges[i_edges++];
    int root1 = uf.Find(cur.vertex1), root2 = uf.Find(cur.vertex2);           V - 1 edges picked up O(E)
    if (root1 != root2) {           Check two vertices are at
        totalCost += cur.cost;           the same set or not O(1)
        i_res++;
        uf.Union(root1, root2);           Add edge's cost to total cost O(1)
    }
}
}                                     Merge two sets O(1)

```

Time complexity: $O(V + E + E\log E + E) \rightarrow O(E\log E)$

圖 9. Kruskal's Algorithm 程式碼與各行對應之時間複雜度

林之根節點編號，其中加入了路徑壓縮最佳化，在查找同時將路徑上每個節點直接與根節點連接，使之後每次查找能直接取得根節點；在 `void UnionFind::Union(int x, int y)` 中，傳入參數為兩個要進行合併的併查集森林之根節點編號，其中使用了按秩合併最佳化，使合併後各子森林高度平衡，以優化查詢時間；若我們想知道最小生成樹所有邊總和權重和節點連結情形，可以在符合條件的邊被挑選時，將邊的資訊額外儲存在一個 `vector`，以便後續查看。

II-3. Results & Discussion

I-3-1. Time & Space Complexity Analysis

➤ Prim's Algorithm

空間上主要用了 min priority queue 來存放所有節點，以及使用了一個 `vector` 追蹤所有節點權重，因此 Prim's Algorithm 的 space complexity 為 $O(V)$ ；演算法執行時各行時間複雜度如圖 8 所示，其中主要部分為：1. 設定所有節點初始權重，對應時間複雜度 $O(V)$ ；2. 所有節點一定會從 priority

queue 中取出，對應時間複雜度為 $O(V)$ ，而每次取出對應時間複雜度為從 min binary heap 中取出最小值的時間複雜度 $O(\log V)$ ；3. 對每次所取出的節點，會遍歷與其相接的鄰近節點，對應時間複雜度為 $O(V)$ ；4. 而鄰近節點有被更新權重者，會被放入 queue 中，對應時間複雜度為插入 min binary heap 的時間複雜度 $O(\log V)$ ；總合以上 Prim's Algorithm 的時間複雜度為 $O(V + V * (\log V + V * \log V)) \rightarrow O(V * (1 + V * \log V)) \rightarrow O(V^2 \log V)$ ，而我們再仔細去看演算法運作的過程，「取出在 queue 中節點 → 遍歷鄰近節點」這個動作其實將每條邊走了兩次，即節點 A 到節點 B 再節點 B 到節點 A，因此 $O(V^2 \log V) \rightarrow O(2E \log V)$ ，可得 Prim's Algorithm 的 time complexity 為 $O(E \log V)$ 。

➤ Kruskal's Algorithm

空間使用上，會將每個節點生成一個併查集，因此空間複雜度為 $O(V)$ ，再來為了將所有邊依照權重排序，需要額外空間存放邊的資訊，對應空間複雜度 $O(E)$ ，綜合上述 Kruskal's Algorithm 的 space complexity 為 $O(V+E)$ ；演算法各段的時間複雜度如圖 9 所示，其中主要部分為：1. 各節點生成併查集對應時間複雜度 $O(V)$ ；2. 將所有邊放入 vector 並進行排序，對應時間複雜度 $O(E + E \log E)$ ；3. 迴圈檢查了每條在 vector 的邊，對應時間複雜度 $O(E)$ ；4. 檢查每條邊的兩端節點是否在同一併查集，以及將併查集的合併的時間複雜度會因不同 UnionFind 而不同，若我們將同一集合內每個點單純以 Linked list 的形式接起來，以及在合併集合時不考慮平衡併查集樹深度，最差情況下會有 $O(V)$ 的深度，使在執行 Find 時會有 $O(V)$ 時間複雜度；若在 struct Vertex 中加入了變數 rank，達到紀錄樹深度的功能，可讓 Union 中進行集合合併時樹的深度最小，如此能優化 Find 的時間複雜度至 $O(\log V)$ ；最後，在執行 Find 時讓每個節點直接與集合根節點連結，便可讓 Find 的平均時間複雜度再優化至 $O(1)$ ；因此本實作中，執行 Union 和 Find 的時間複雜度為 $O(1)$ ；綜合以上，此演算法時間複雜度為 $O(V+E+E \log E+E)$ ，而 $V-1 \leq E \leq V^2$ ，可得 Kruskal's Algorithm 的 time complexity 為 $O(E \log E)$ 。

II-4. Conclusion

由以上討論可以得到，將一個無向圖產生最小生成樹時，Prim's Algorithm 主要是以點的權重來出發，而 Kruskal's Algorithm 則是以邊的權重出發；Prim's Algorithm 的空間複雜度為 $O(V)$ ，時間複雜度為 $O(E \log V)$ ，此情況是使用了 priority queue 存取節點；Kruskal's Algorithm 的空間複雜度為 $O(V+E)$ ，時間複雜度為 $O(E \log E)$ ，本實作使用了優化後的 disjoint-set data structure；由於 $V-1 \leq E \leq V^2$ ，可以得出在稠密圖(dense graph)的情況下，邊比較多使用 Prim's Algorithm 執行時間效率較好，而在稀疏圖(sparse graph)的情況下，邊比較少使用 Kruskal's Algorithm 執行時間效率較好。

Part III: Shortest Path Algorithm

III-1. Introduction

在一個圖中，若我們想得出從給定的一個起點抵達給定的一個終點所需之最短距離路徑，可以使用 single-source shortest path algorithm 來解出答案；舉一個實際生活例子，在擁有所有台北捷運各站之間的費用與班車時間，我們就可以使用此演算法計算在最短時間或最少費用下，從某站到某

站的最佳路線；本門課所提到演算法有 Breadth-first search algorithm(BFS)、Dijkstra's Algorithm 和 Bellman-Ford Algorithm，其中 BFS 僅僅是搜尋挑選節點之相連鄰近節點，無法處理當連接邊擁有權重，對應上面的例子，使用 BFS 我們只能得到從出發站到終點站最少站數，無法得到最短時間或最少費用；本作業需考慮邊有權重的情況，因此下面主要討論 Dijkstra's Algorithm 和 Bellman-Ford Algorithm 兩種演算法。

III-2. Implement Details

III-2-1. Dijkstra's Algorithm

此演算法的運算模式與 BFS 類似，差別在於會優先從權重較小的節點向周邊搜索；課堂中介紹的演算法版本可以計算給定一個起點到所有連結點的最短路徑，本作業則是計算給定一個起點與終點的最短路徑，流程如下：1. 設定起始點權重為 0，剩下節點權重均為無限大；2. 取出當前權重最小節點，檢查是否為終點，若是則印出終點權重，結束演算法；3. 若取出節點並非終點，遍歷與其連接之相鄰節點；4. 若取出節點權重加上邊權重小於連接節點之權重，更新連接節點之權重；5. 重複步驟 2 到 4 直到抵達終點。

Dijkstra's Algorithm 的程式碼如圖 10 所示；各節點的權重使用了 vector 儲存，不同 index 則對應不同節點的編號，這裡使用了 INT_MAX 來對應權重無限大；各邊的資訊使用了 nested vector 儲存，由於題目給的是有向圖，假設某一邊為節點 v1 單向連到節點 v2，權重為 c，資訊儲存格式為 edges[v1].push_back(make_pair(v2, c))；為了達到步驟 2 中取出當前權重最小節點，這裡使用了 priority queue，並使用了 greater function 使 queue 的 top 能得到最小值，而起始節點與被更新的節點以(vertex weight, vertex index)的格式存入 queue 以對應 greater function 的排序。

III-2-2. Bellman-Ford Algorithm

相對於前一個演算法，此演算法以貪心法不段尋找節點與節點之間最佳的連接邊，流程如下：1. 將起始點權重設定為 0，其餘點設定為無限大；2. 遍歷每條邊，若邊出發節點權重加上邊的權重小於邊目的節點的權重，更新目的節點的權重；3. 步驟二重複 $V - 1$ 次， V 為節點數量；4. 再重複一次步驟二，但此時發現有節點權重可以更新，印出「圖有負權重環」，結束演算法；5. 在沒檢測出負權重環後，印出終點權重。

Bellman-Ford Algorithm 的程式碼如圖 11 所示；各節點的權重使用了 vector 儲存，不同 index 則對應不同節點的編號，這裡使用了 INT_MAX 來對應權重無限大；各邊資訊使用 nested vector 儲存，由於每條邊都會被遍歷到，不需要 random access 特定邊，因此在給定節點 v1 單向連到節點 v2，權重為 c 時，資訊儲存格式為 edges.push_back({v1, v2, c})；每當進行一輪所有邊的遍歷與節點權重更新，可以理解為至少有一條邊已被挑選為最短路徑上的邊，而在有 V 個節點的情況下，最短路徑最多只有 $V-1$ 條邊，因此步驟二只要進行 $V-1$ 次就能確保得到最短路徑，這也表示可能用少於 $V-1$ 次的情況下就得到最短路徑；由於步驟三已確定要得到最短路徑，即各節點權重不會再更新，此時若多做一次步驟二還能得到節點更新，表示圖中有負權重環，讓路徑加總權重走過還後變小，這樣的圖無法計算最短路徑；由於我們設定起始點權重為 0，因此演算法的最後，終點權重就能代表從起點到終點之最短路徑之總權重。

```

typedef pair<int, int> iPair;
vector<vector<iPair>> edges ----- Input graph

vector<int> vertexWeight(numberOfVertices, INT_MAX); ----- Set all weight of all vertices as
vertexWeight[startVertex] = 0; ----- infinite and starting vertex as 0 O(V)
priority_queue<iPair, vector<iPair>, greater<iPair>> pq; ----- Create min priority queue &
pq.push(make_pair(vertexWeight[startVertex], startVertex)); ----- push starting vertex into queue O(1)

while (!pq.empty()) { ----- O(V)
    v1 = pq.top().second; ----- Pop min vertex from queue O(logV)
    pq.pop(); ----- O(logV)

    if (v1 == endVertex) { ----- O(1)
        cout << vertexWeight[v1] << endl;
        break;
    }

    for (iPair& edge : edges[v1]) { ----- O(V)
        v2 = edge.first; cost = edge.second; ----- Get every adjacent vertex
        if (vertexWeight[v1] + cost < vertexWeight[v2]) { ----- of current vertex O(1)
            vertexWeight[v2] = vertexWeight[v1] + cost; ----- Update adjacent vertex
            pq.push(make_pair(vertexWeight[v2], v2)); ----- & push into queue O(1 + logV)
        }
    }
}
}

Time complexity:  $O(V + E \log V) \rightarrow O(E \log V)$ 

```

圖 10. Dijkstra's Algorithm 程式碼與各行對應之時間複雜度

```

vector<vector<int>> edges ----- Input graph

vector<int> vertexWeight(numberOfVertices, INT_MAX); ----- Set all weight of all vertices as
vertexWeight[startVertex] = 0; ----- infinite and starting vertex as 0 O(V)

for (int i = 0; i < numberOfVertices - 1; i++) { ----- O(V)
    for (vector<int>& edge : edges) { ----- O(E)
        v1 = edge[0]; v2 = edge[1]; cost = edge[2]; ----- O(1)
        if (vertexWeight[v1] != INT_MAX && ----- O(1)
            vertexWeight[v1] + cost < vertexWeight[v2]) { ----- O(1)
                vertexWeight[v2] = vertexWeight[v1] + cost; ----- O(1)
            }
        }
    }
}

for (vector<int>& edge : edges) { ----- O(E)
    v1 = edge[0]; v2 = edge[1]; cost = edge[2]; ----- O(1)
    if (vertexWeight[v1] != INT_MAX && ----- O(1)
        vertexWeight[v1] + cost < vertexWeight[v2]) { ----- O(1)
            cout << "Negative loop detected!" << endl; ----- O(1)
            return 0;
        }
    }
}

cout << vertexWeight[endVertex] << endl; --- result O(1) Time complexity:  $O(V + VE + E + 1) \rightarrow O(VE)$ 

```

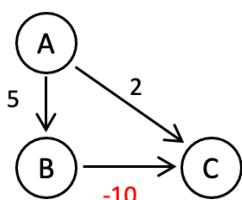
圖 11. Bellman-Ford Algorithm 程式碼與各行對應之時間複雜度

III-3. Results & Discussion

III-3-1. Dijkstra's Algorithm

空間複雜度上，此演算法需追蹤所有節點的權重變化，以及使用 priority queue 提供每次 pop 能取出當前最小權重節點，因此 **Dijkstra's Algorithm 的 space complexity 為 $O(V)$** ；演算法各段時間複雜度如圖 10 所示，其中主要部分為：1. 設定所有節點初始權重，對應時間複雜度 $O(V)$ ；2. 所有節點一定會從 priority queue 中取出，對應時間複雜度為 $O(V)$ ，而每次取出最小權重節點對應時間複雜度為 $O(\log V)$ ；3. 檢查當前節點是否為終點，時間複雜度為 $O(1)$ ；3. 對當前節點遍歷與其相接之鄰近節點，對應時間複雜度為 $O(V)$ ；4. 鄰近節點有被更新權重者，會被放入 queue 中，對應時間複雜度為 $O(\log V)$ ；總合以上 Dijkstra's Algorithm 的時間複雜度為 $O(V + V * (\log V + V * \log V)) \rightarrow O(V * (1 + V * \log V)) \rightarrow O(V^2 \log V)$ ；觀察從 priority queue 取出節點與遍歷相鄰節點的動作，等同於將所有邊都遍歷了一遍，因此 $O(V^2 \log V) \rightarrow O(E \log V)$ 。可得 **Dijkstra's Algorithm 的 time complexity 為 $O(E \log V)$** 。

Search shortest path from A to C:

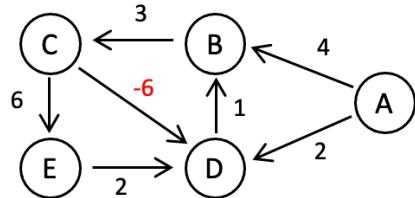


Current vertex	Vertices in queue
A	C(2, A), B(5, A)
C	B(5, A)

Result path: A → C, total cost = 2
!=

Shortest path: A → B → C, total cost = -5

Search shortest path from A to E:



Current vertex	Vertices in queue
A	D(2, A), B(4, A)
D	B(3, D), B(3, D)
B	C(6, B), C(6, B)
C	D(0, C), C(6, B), E(12, C)
D	B(1, D), C(6, B), E(12, C)
B	C(4, B), C(4, B), E(12, C)
C	D(-2, C), C(4, B), E(10, C), E(10, C)

No exit!

圖 12. 使用 Dijkstra's Algorithm 在圖含有負權重邊時可能之錯誤結果

complexity 為 $O(E \log V)$ 。

當邊的權重有負值時使用此演算法可能會產生錯誤的結果，如圖 12 所示，我們在演算法中追加了當節點更新時同時更新其父節點，在 queue 中所有節點以「節點編號(節點權重, 父節點編號)」的格式表示；圖 12 中左邊的例子可以看到，我們得到了最小路徑為 A→C，總權重為 2，然而圖中可以看出答案應該是 A→B→C，總權重為 -5，原因來自於自 priority queue 中取出之節點為終點時，立即結束演算法，可能會忽略後續操作所能產生的最短路徑結果；在圖 12 中右邊的例子，當路徑走過 B→C→D→B 一遍就會讓總權重變小，且在 priority queue 的作用下會進入無限迴圈；由此可知 Dijkstra's Algorithm 只要在圖有負權重的邊時應避免使用。

III-3-2. Bellman-Ford Algorithm

空間複雜度上，此演算法同樣會追蹤所有節點的權重變化，因此 **Bellman-Ford Algorithm 的 space complexity 為 $O(V)$** ；演算法各段時間複雜度如圖 11 所示，主要部分為：1. 設定所有節點初

始權重，對應時間複雜度 $O(V)$ ；2. 所有邊的遍歷對應時間複雜度 $O(E)$ ，需要做 $V-1$ 次才能確保得到最短路徑，因此時間複雜度為 $O(VE)$ ；3. 檢查是否負權重環需做一次所有邊的遍歷，對應時間複雜度 $O(E)$ ；綜合以上時間複雜度為 $O(V + VE + E) \rightarrow O(VE)$ ，可得 **Bellman-Ford Algorithm 的 time complexity 為 $O(VE)$** 。

圖 13 演示了使用 Bellman-Ford Algorithm 在圖含有負權重邊和圖含有負權重環的情形，我們同樣在節點權重被更新時也更新父節點，case A 在上一個演算法中有路徑與總權重不匹配的結果，在此演算法則呈現了匹配的結果，可歸因於這裡所有邊都會被等次數的遍歷並更新節點，不會有用 priority queue 造成某些節點與邊優先處理的問題；case B 中可以看到在最後一輪檢查時仍有節點權重能被更新，因此觸發顯示錯誤訊息並結束演算法，解決了上個演算法中無限迴圈的問題。

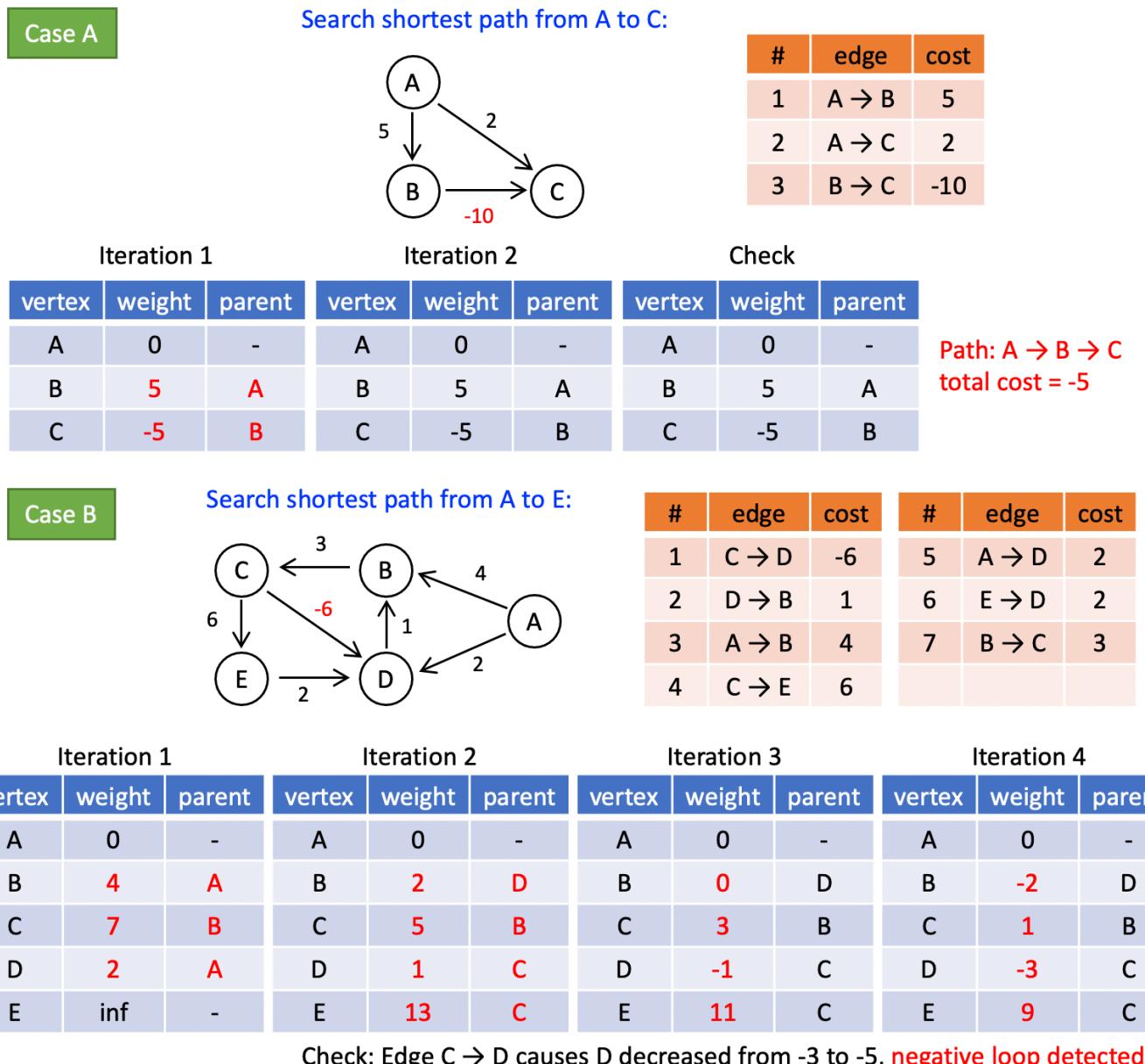


圖 13. 使用 Bellman-Ford Algorithm 在 case A: 圖含有負權重邊與 Case B: 圖含有負權重環之結果

III-3-3. Print the shortest path

圖 14 呈現了若需要印出最短路徑上所有節點時的修改 source code，這裡我們將最短路徑上所有節點使用 Linked list 串接，因此每個節點使用 struct 建構，並以 index, weight, parent 分別紀錄節點編號、節點權重與相連之父節點；在兩個演算法中當節點權重被更新時，同時更新該節點的

parent，此方法下演算法結束時得到的 Linked list 頭會是終點、尾會是起點；若想要最短路徑是從起點到終點印出，可以從頭到尾依序遍歷 Linked list 並將節點編號存入 stack，再從 stack 中依序取出印出編號即可達成。

```
struct Vertex {
    int index, weight;
    Vertex* parent;
    Vertex (int i): index(i), weight(INT_MAX), parent(nullptr) {}
};

vector<Vertex> vertices;
for (int i = 0; i < numberofVertex; i++)
    vertices.push_back(Vertex(i));
```

```
for (iPair& edge : edges[v1]) {
    v2 = edge.first; cost = edge.second;
    if (vertices[v1] + cost < vertices[v2]) {
        vertices[v2].weight = vertices[v1].weight + cost;
        vertices[v2].parent = &vertices[v1];
        pq.push(make_pair(vertices[v2].weight, v2));
    }
}
```

Dijkstra's Algorithm

```
for (vector<int>& edge : edges) {
    v1 = edge[0]; v2 = edge[1]; cost = edge[2];
    if (vertices[v1].weight != INT_MAX &&
        vertices[v1].weight + cost < vertices[v2].weight) {
        vertices[v2].weight = vertices[v1].weight + cost;
        vertices[v2].parent = &vertices[v1];
    }
}
```

Bellman-Ford Algorithm

```
stack<int> stk;
Vertex* cur = &vertices[endVertex];

while (!cur) {
    stk.push(cur->index);
    cur = cur->parent;
}

while (!stk.empty()) {
    cout << stk.top() << ' ';
    stk.pop();
}
```

圖 14. 印出最短路徑所有節點所需之 source code 修改

III-4. Conclusion

從上述討論中，Dijkstra's Algorithm 的時間複雜度為 $O(E \log V)$ ，Bellman-Ford Algorithm 的時間複雜度為 $O(VE)$ ，兩者的空間複雜度均為 $O(V)$ ，因此在圖沒有負權重邊時，可選擇 Dijkstra's Algorithm 來達到較好的執行時間效率，然而 Dijkstra's Algorithm 在處理圖含有負權重邊時可能產生錯誤結果或無限迴圈，因此圖有負權重邊時就要選擇使用 Bellman-Ford Algorithm；若我們不僅需要最短路徑的總權重，也需要最短路徑上經過的所有節點資訊，可使用 Linked list 的方式在更新節點時彼此串結，之後依序遍歷 Linked list 上所有節點編號即可得到。