**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(nlogn)，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 >= 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 pivot放在array右半部，element 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] pivot則停下來，右指針遇到array[j]

一張含有 文字 的圖片

自動產生的描述

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，

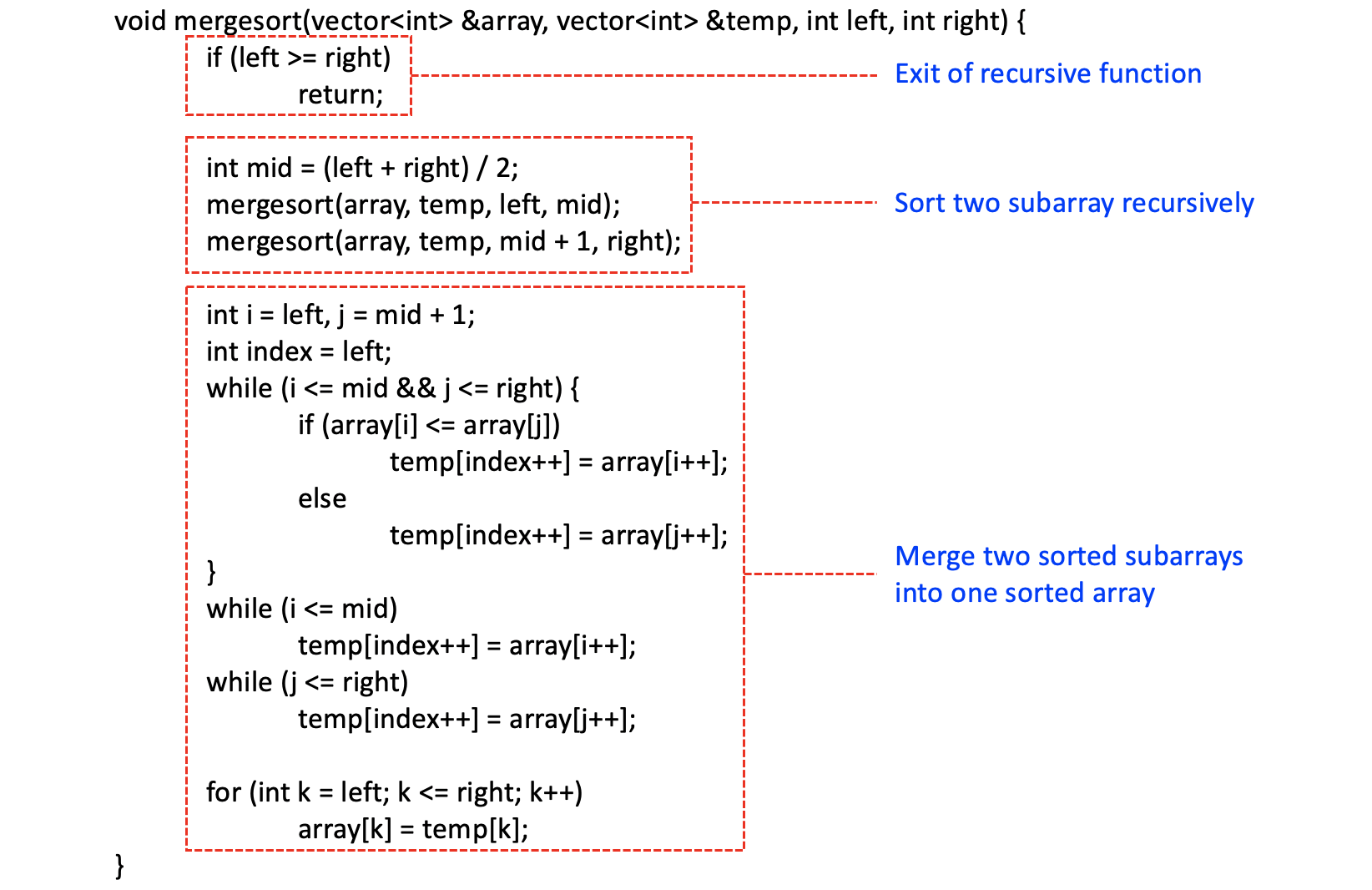


圖2. Merge sort程式碼

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**

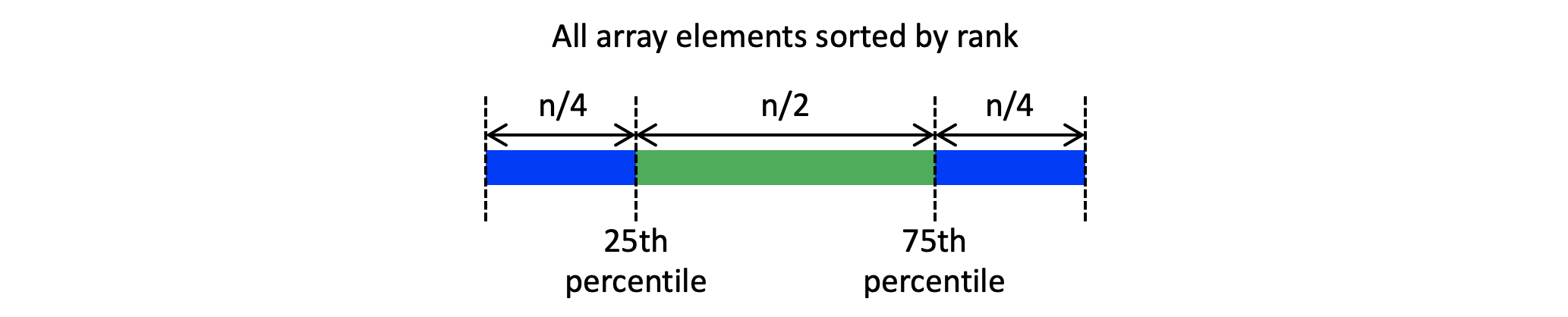
* **Quick sort**

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) + … + O(1) = O(n \* (n + 1) / 2) = O(n2)，因此可得**quick sort最差情況下time complexity為O(n２)**。

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log2n ；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(logn) \* O(n) = O(nlogn)，得證**quick sort在最佳情況下time complexity為O(nlogn)**。

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log4n，右子樹較慢，對應次數為log3/4n；從圖4我們可以得到，起初log4n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log4n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 \* log3/4 n) = O(n \* log2 n / log2(3/4)) = O(nlogn)。

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 \* log3/4 n)，綜合可得**quick sort在平均情況下time complexity為O(nlogn)**。

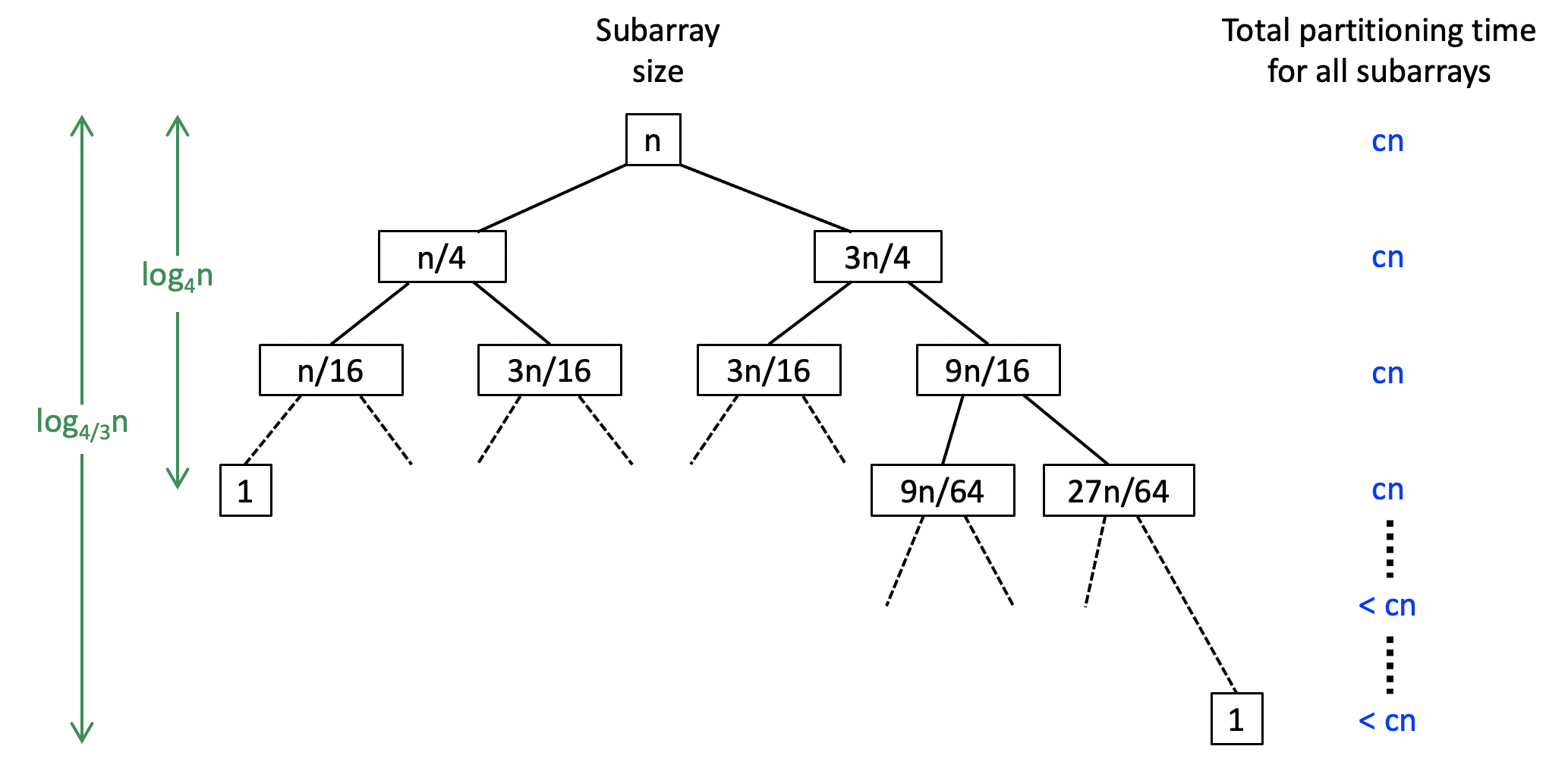
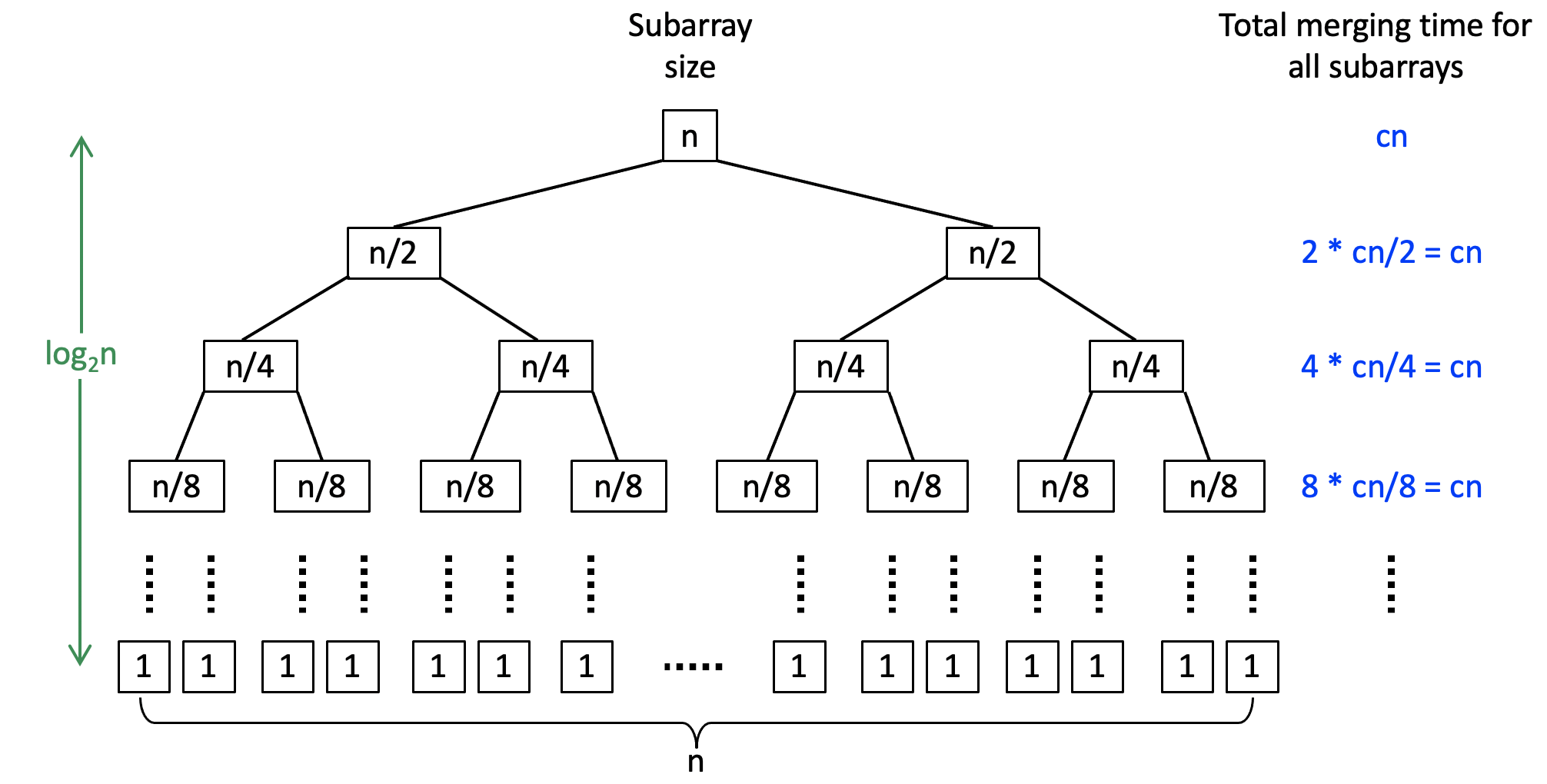


圖4. 在每次pivot都挑選了array element第25或第75百分位數下遞迴執行示意圖

* **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log2n；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

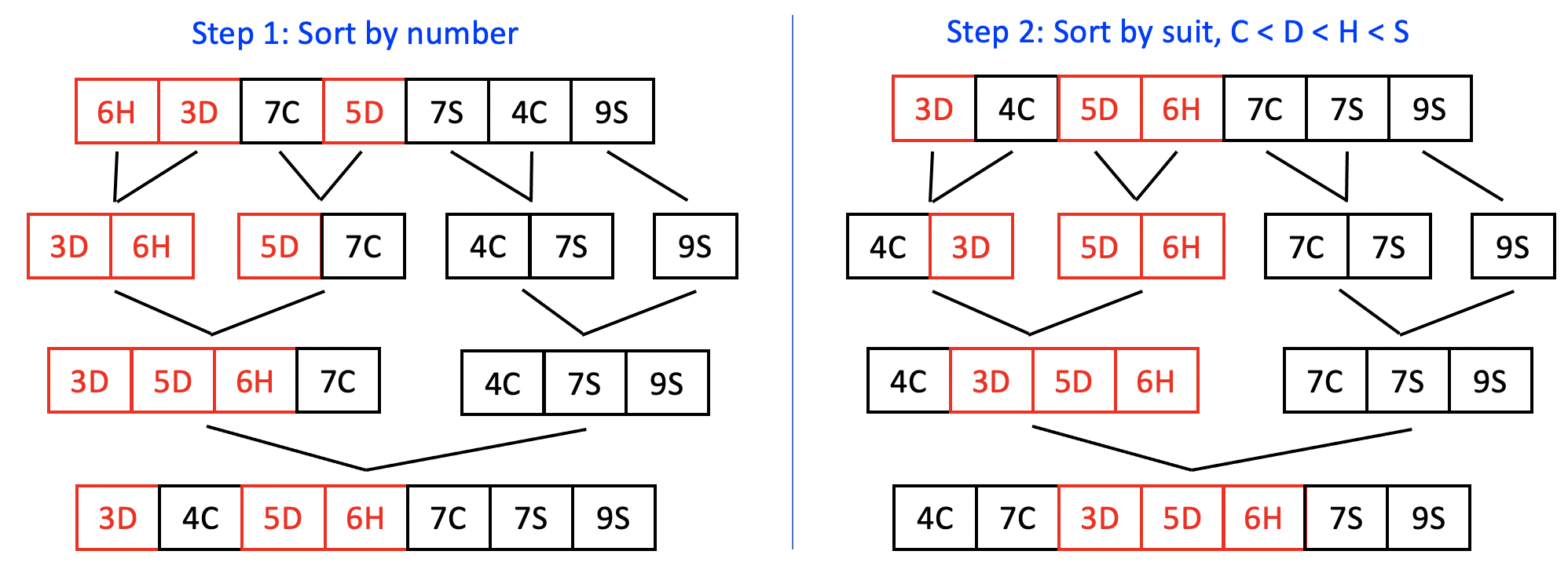
總共合併排序時間均為c \* n，因此Merge sort總執行時間為O(cn \* log2n) = O(nlogn)，得到**merge sort的time complexity為O(nlogn)**。

**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

一張含有 文字, 標誌, 差異, 許多 的圖片

自動產生的描述圖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(nlogn)；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

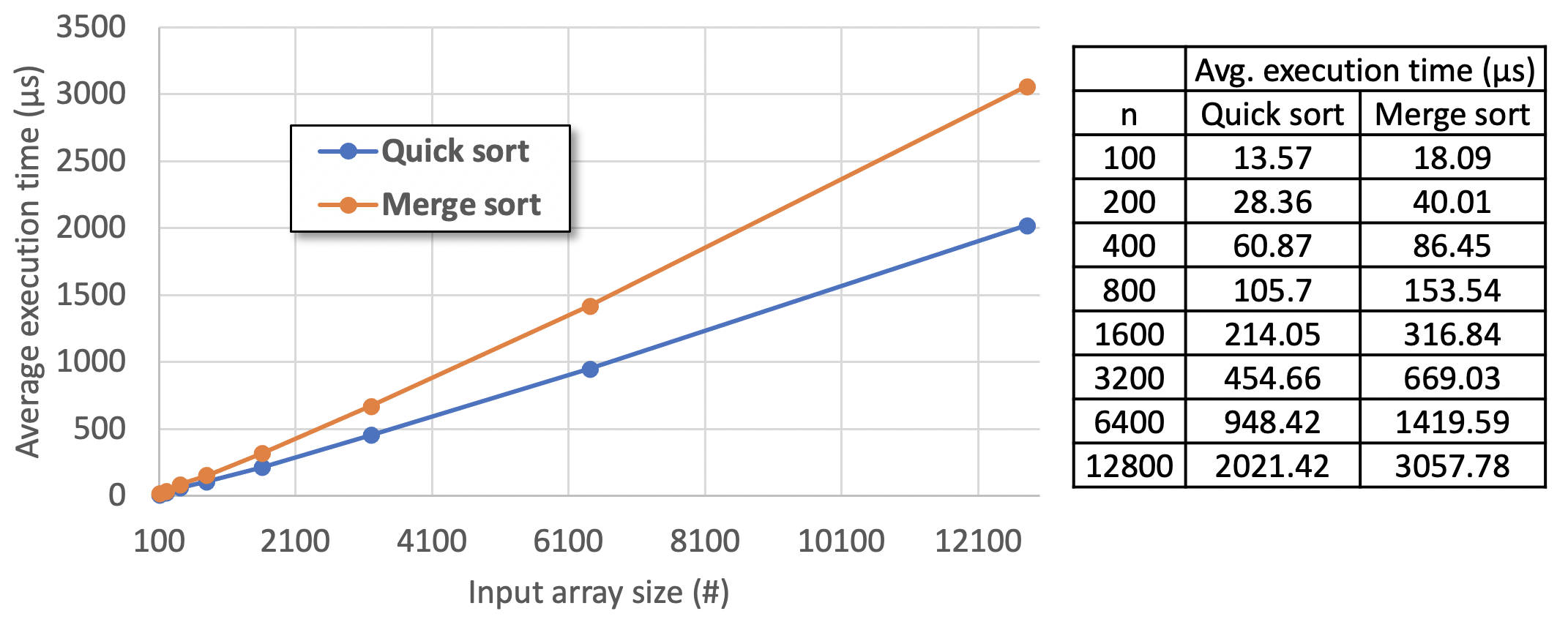


圖8. Quick sort和merge sort在不同input array size下的平均執行時間比較

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(nlogn)，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(nlogn)，而每次都是最不平衡分割下時間複雜度會是O(n2)，Merge sort的最佳與最差時間複雜度都是O(nlogn)；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

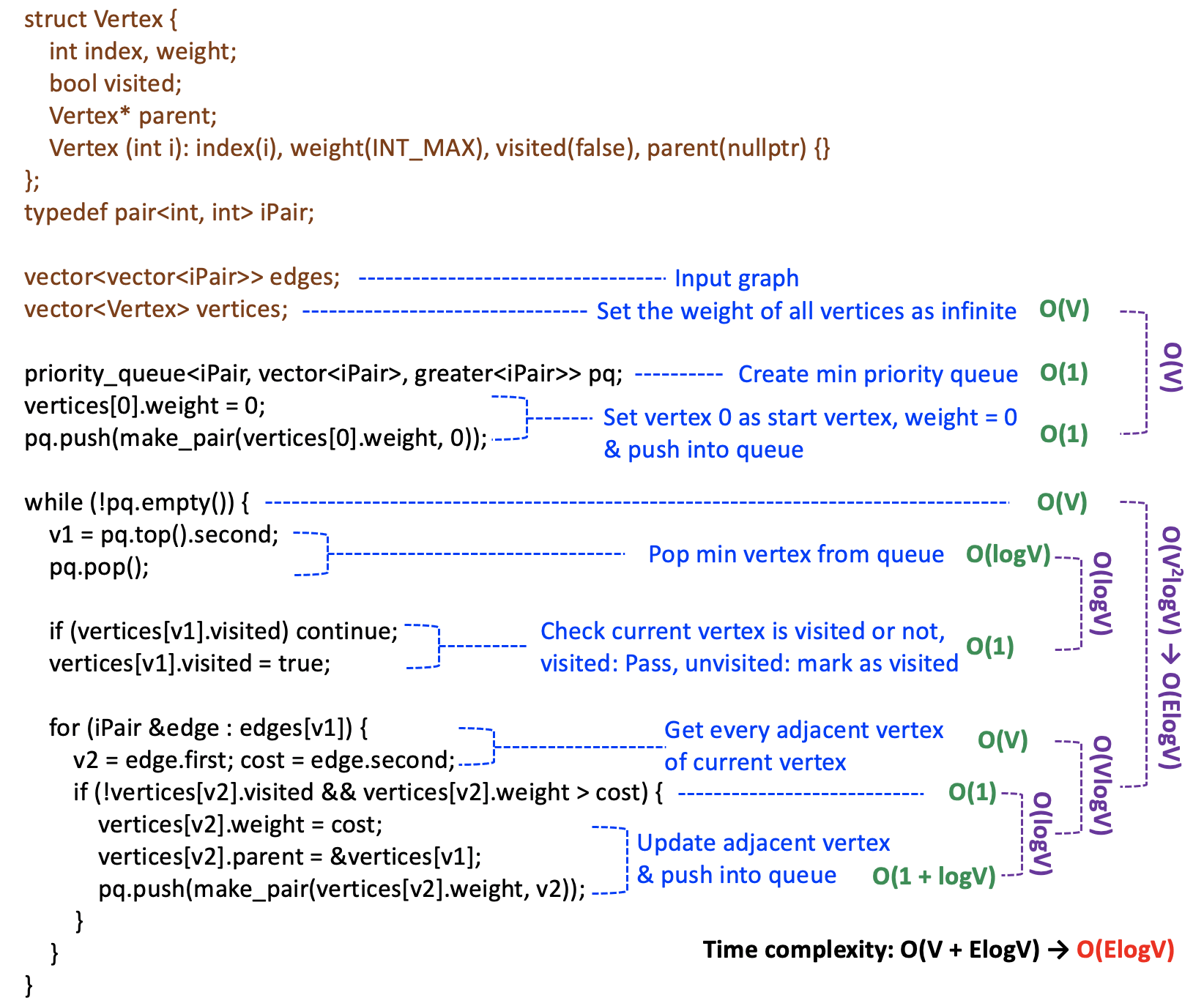


圖8. Prim’s Algorithm程式碼與各行對應之時間複雜度

係可以透過遍歷每個節點的parent得到。

**II-2-2. Kruskal’s Algorithm**

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

一張含有 桌 的圖片

自動產生的描述

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(logV)；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(logV)；總合以上Prim’s Algorithm的時間複雜度為O(V + V \*( logV + V \* logV)) → O(V \*(1 + V \* logV)) → O(V2logV)，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(V2logV) → O(2ElogV)，可得**Prim’s Algorithm的time complexity為O(ElogV)**。

* **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 + ElogE)；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(logV)；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+ElogE+E)，而V – 1 E V2，可得**Kruskal’s Algorithm 的time complexity為O(ElogE)**。

**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(ElogV)，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(ElogE)，本實作使用了優化後的disjoint-set data structure；由於V-1 E V2，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點距離路徑，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。

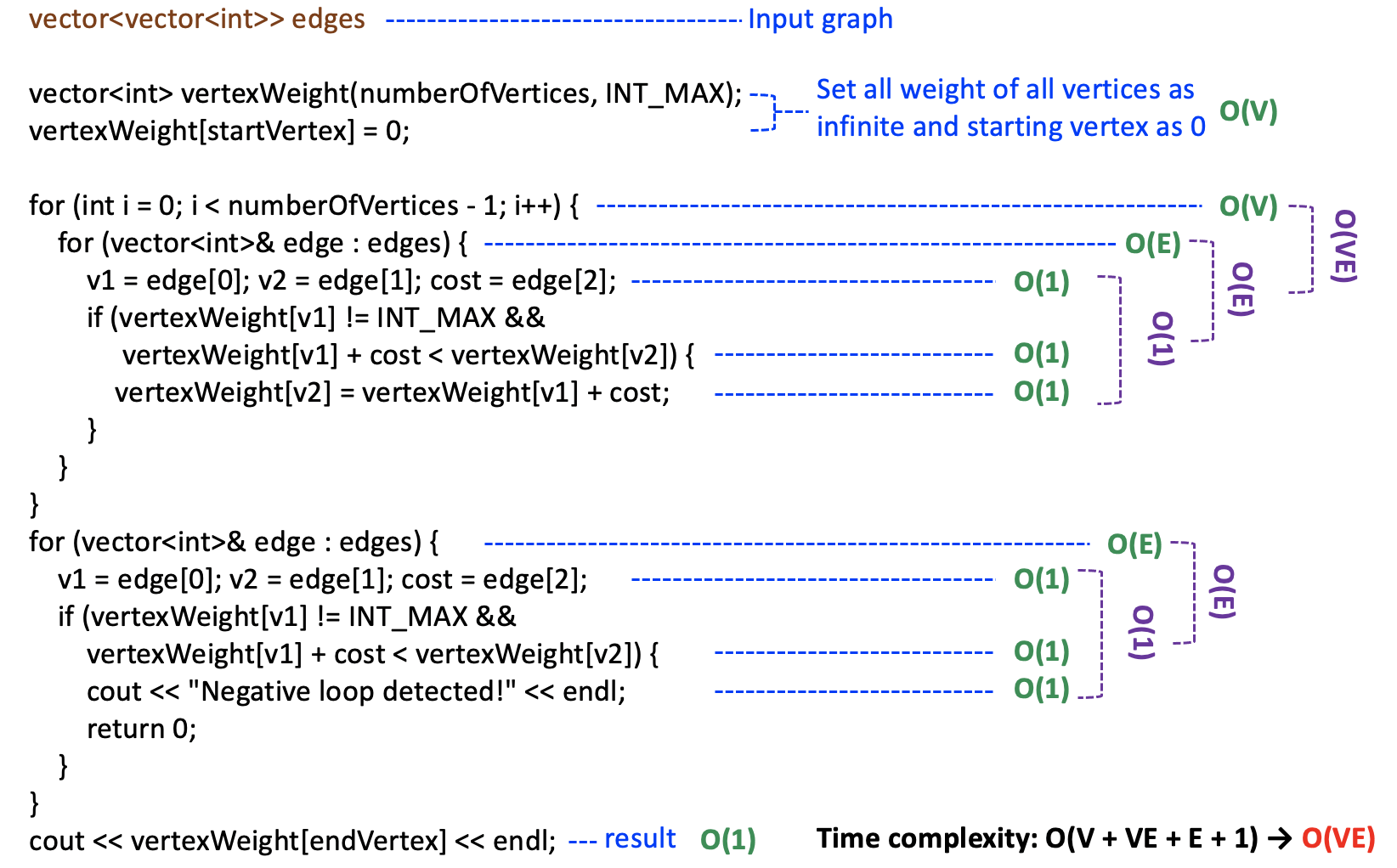
**III-2-2. Bellman-Ford Algorithm**

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。

一張含有 桌 的圖片

自動產生的描述圖10. Dijkstra’s Algorithm程式碼與各行對應之時間複雜度

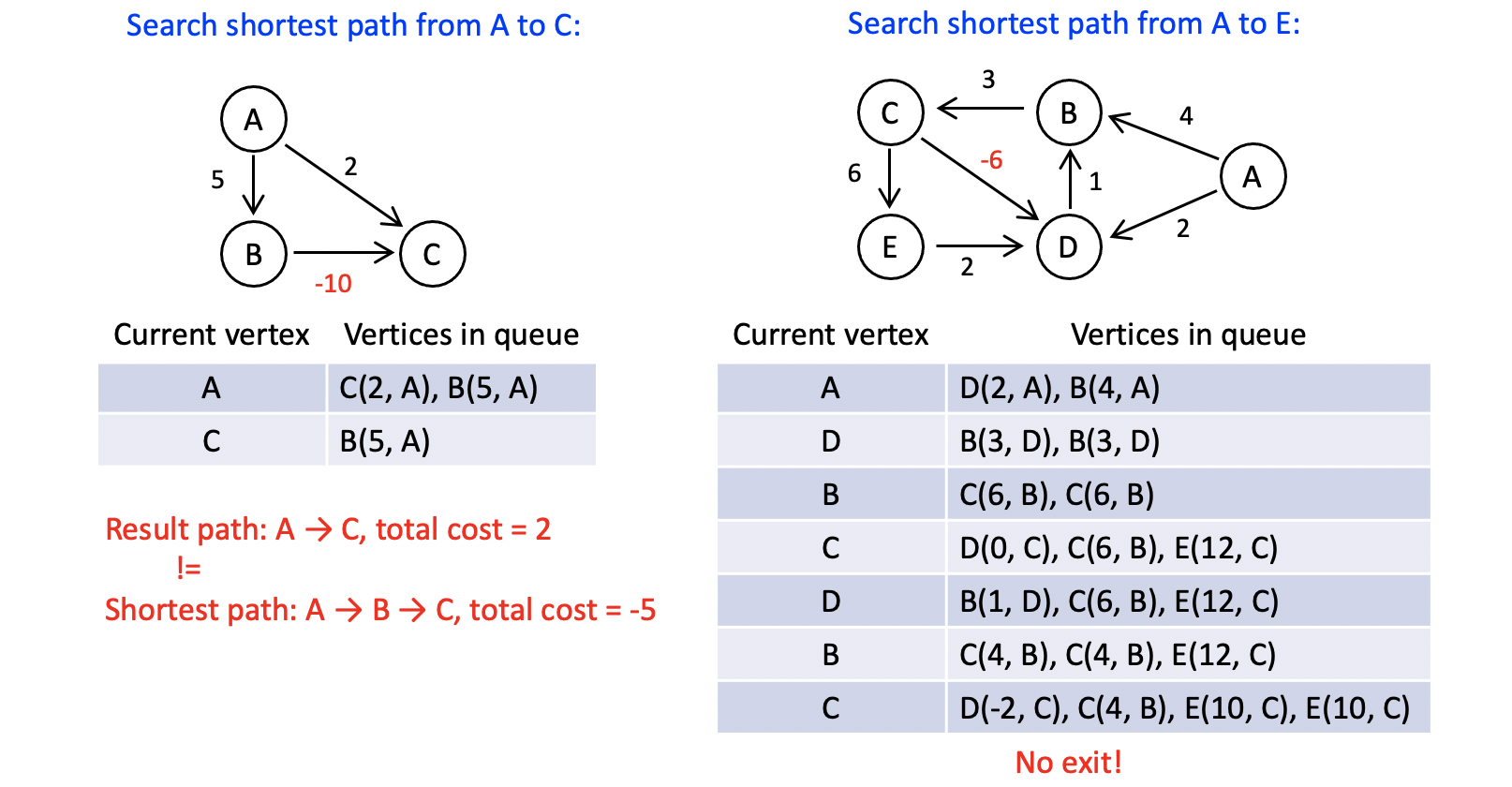
圖11. Bellman-Ford Algorithm程式碼與各行對應之時間複雜度

**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(logV)；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(logV)；總合以上Dijkstra’s Algorithm的時間複雜度為O(V + V \*( logV + V \* logV)) → O(V \*(1 + V \* logV)) → O(V2logV)；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(V2logV) → O(ElogV)，可得**Dijkstra’s Algorithm 的time**

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(ElogV)**。

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) → 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。

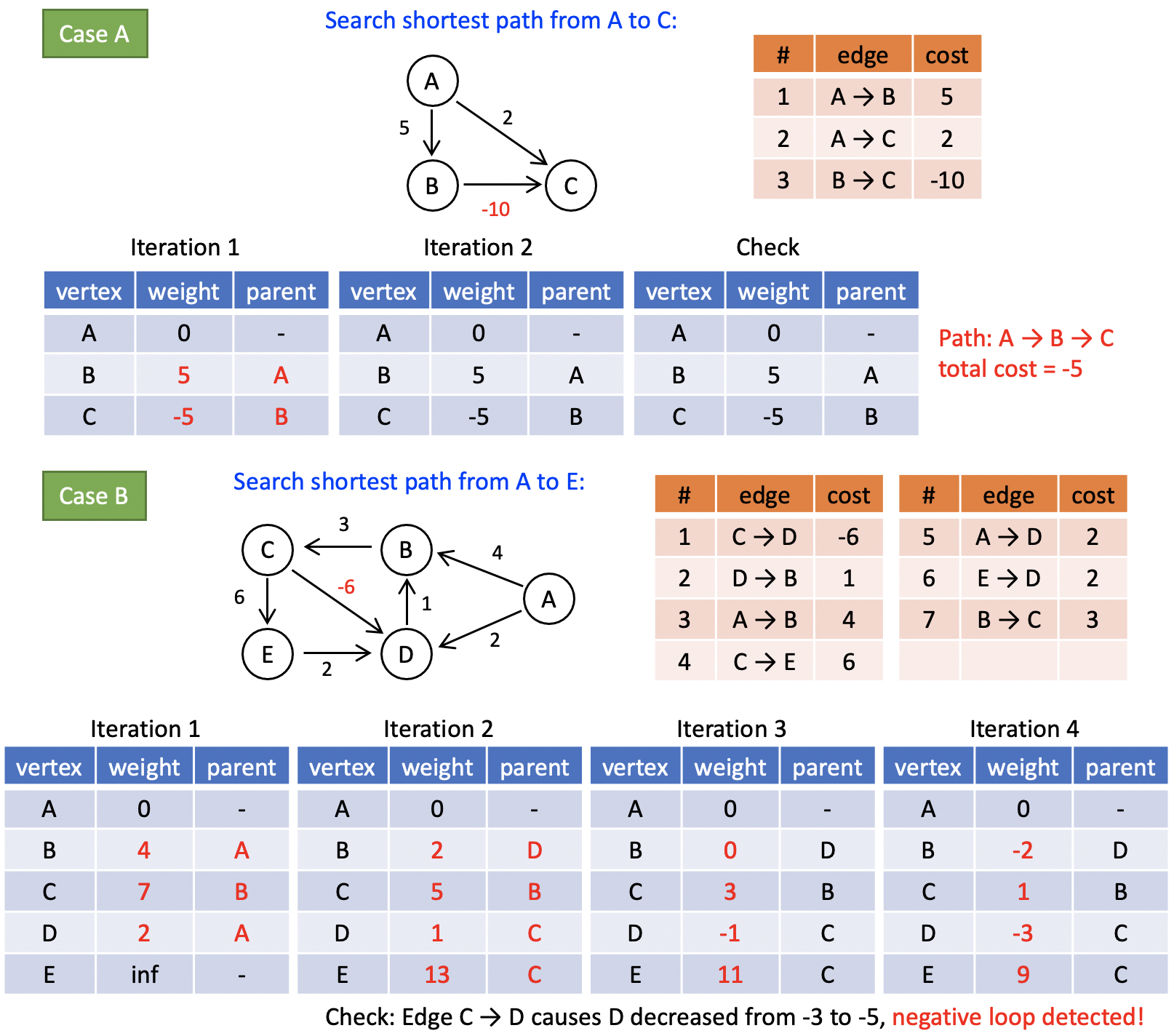


圖13. 使用Bellman-Ford Algorithm在case A: 圖含有負權重邊與Case B: 圖含有負權重環之結果

**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。

一張含有 文字 的圖片

自動產生的描述圖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(ElogV)，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。