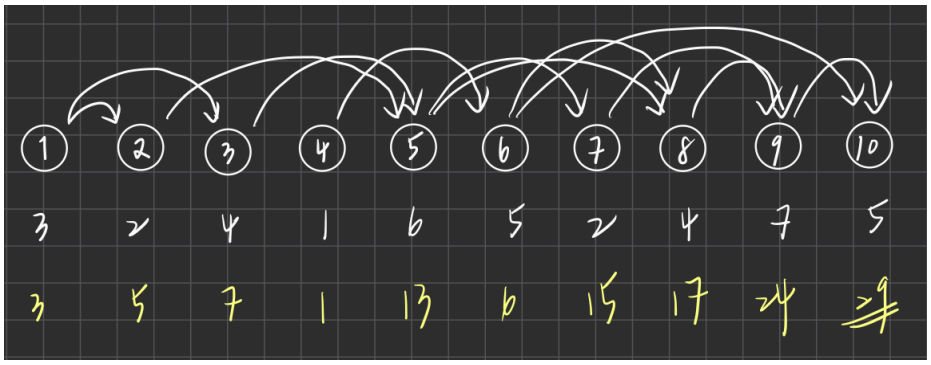
2024 OOP&DS Homework 2

112550015, 劉家琪

1. Implementation
   - Steps



白：process time, 黃：finish time

演算法：
1.把沒有被任何其他vertex指向的vertex(source)的finish time都設定為process time，因為它們不需要任何prerequisites
2.處理finish time已經確定的vertex (假設是u) 所指向的vertex(假設是v)的finish time
if (v還有被其他vertex指向（假設有u1, u2, u3….）)
    v的finish_time = max(u1, u2, u3…的finish_time) + v 的process time
    // 如果有任何一個的finish_time還沒確定都不能做
else //即v只有被u一個指向
    finish_time = u的finish_time + v 的process time
3.持續做2.，直到所有的finish_time都被決定
4.output最大的finish_time

   - The way you deal with the data in the problem
adjacent list：要處理「要找到同樣指向某個vertex的其他vertex」如果每次都要遞迴不如一開始就建表，之後查表就是用O(1)的速度找到
inDegree：存所有vertex分支度（入支度）的array，可以快速知道還有沒有要先處理的vertex

```
vector<int> finish_time(n, -1);        //-1表示還沒被決定
vector<int> inDegree(n, 0);
vector<vector<int> > adj(n);
for(int i = 0; i < m; i++) {    // iterate all items in relations
    //在index部分的-1是因為array會從0開始數，但vertec的編號是1開始
    inDegree[relations[i][1]-1]++;
    //relations[i][1]就是relation當中被指入（後做）的vertex, 將其入支度+1
    adj[relations[i][0]-1].push_back(relations[i][1]-1);
    //在adj[往外指出的vertex]中加入「被指入的vertex」
}
```

- Explain your code (do not paste all the code without any explanation)

```
queue<int> q;
for (int i = 0; i < n; i++) {
     if(inDegree[i] == 0) {
            finish_time[i] = time[i];
           q.push(i);
      }
}
```

queue q：存應該要處理其分支vertex的vertex，用queue是因為FIFO比較符合直覺，但其實裡面的都是處理好的vertex所以要用stack其實也不影響結果。
先處理掉入支度0的vertex，並把這些vertex都放入q，因為它們的finish_time都確定了，所以計算其他vertex的時候就可以使用他們了。

```
while(!q.empty()) {
    int u = q.front();
    q.pop();
    for(int v : adj[u]) {
        finish_time[v] =
               max(finish_time[v], time[v] + finish_time[u]);
        if(--inDegree[v] == 0) {
            q.push(v);
        }
    }
}
```

一直執行到所有vertex的adjacent list都被處理過。u是q的其中一個element，並在取出u之後從q中pop掉，再根據u的adjacent list去update每個v的finish_time（最後會停在最大值），每處理一個v就會把v的入支度-1，當處理的是最後一個指向v的vertex時，會將v也放入q（意即v的finish_time已經被確定了，v指向的vertex開始有可能被處理）。執行到最後一個vertex（出支度為零）時，因為adjacent list是空的，q也是空的，就會停跳出迴圈了。

```
int ret = *max_element(finish_time.begin(), finish_time.end());
cout << ret;
```

輸出finish_time中最大的，使用了std::max_element的語法：

```
max_element(vec.begin(), vec.end())
```

兩個參數分別會begin跟end的iterator，並會回傳含有最大元素的iterator（所以ret要加星號）

2. Time complexity
    - Analyze the time complexity in detail

```
    for (int i = 0; i < m; i++) { // O(m)
        cin >> relations[i][0] >> relations[i][1];   // O(1)
    }  // O(m) * O(1) = O(m)

    for (int i = 0; i < n; i++) { // O(n)
        cin >> time[i];     // O(1)
    }   // O(n) * O(1) = O(n)

    for(int i = 0; i < m; i++) { // O(m)
       inDegree[relations[i][1]-1]++;    // O(1)
       adj[relations[i][0]-1].push_back(relations[i][1]-1); // O(1)
    } // O(m) * O(1) = O(m)

    for (int i = 0; i < n; i++) { // O(n)
        if(inDegree[i] == 0) { // O(1)
            finish_time[i] = time[i]; // O(1)
            q.push(i); // O(1)
        }
    } // O(n) * O(1) = O(n)

    while(!q.empty()) { // O(n):iterate all vertex
        int u = q.front(); // O(1)
        q.pop(); // O(1)
        for(int v : adj[u]) {
        // we don't know how many times in each round
        // but we know there's m relations b/w vertices
        // so this would implement m times in the whole while loop
            finish_time[v] =
               max(finish_time[v], time[v] + finish_time[u]); // O(1)
            if(--inDegree[v] == 0) { // O(1)
                q.push(v); // O(1)
            }
        }
    } // O(max(m, n)) = O(m + n)

    int ret = *max_element(finish_time.begin(), finish_time.end());
    // O(1)
    cout << ret; // O(1)
    return 0; // O(1)
}
// O(m) + O(n) + O(m) + O(n) + O(m + n) + O(1) + O(1) + O(1)
// = O(m + n)
```

3. Challenges/ discussion
    - Time complexity
      (1) **Topological sorting + Dynamic Programming**：本題我在實作中使用的方法，用Topological sorting來確定處理vertex的順序，並使用Dynamic Programming來計算each vertex的完成時間。
      Advantage:對於有明確dependency relation的問題效率極好，能確保每個工廠在滿足prerequisites後立即開始。
      Time complexity: O(m + n)

      (2) **DFS**：使用DFS來遍歷整個DAG，並在過程中計算每個vertex的完成時間。
      Advantage:直觀且易於實作，尤其小型圖或dependency relation簡單時。
      Time complexity: O(m + n)，但實作中可能比Topological sorting慢，因為每次遞迴呼叫函式都需要額外的空間。

      (3) **Dijkstra's algorithm**：找最短路徑，在weighted DAG中找到從起點到終點的最短時間，要把vertex的處理時間當作每個edge的時間。
      Disadvantage:對於只需確定完成時間而非最短路徑的問題過於複雜。
      Time complexity: O((m + n)logn)

    - Your discover
      (1) Directed Acyclic Graph
      此問題是標準的DAG，其中vertex為factory，edge為relation。使用DAG來可以更容易地進行Topological sorting，時間複雜度可以達到O(m + n)。

      (2) Topological sorting
      一種線性排序，其中每個vertex u在vertex v之前出現，if and only if 從u到v的edge exist。使用拓樸排序可以確保在處理某個工廠之前，其所有的先決條件都已經被處理。

      (3) Dynamic Programming
      動態規劃可以用來記錄每個factory的最早完成時間，並基於已知的prerequisites 進行更新，以空間換取時間來避免重複計算，並確保在每一步都使用已經計算過的最優解。

    - Which is better algorithm in which condition
      (1) **Topological sorting +Dynamic Programming**
        - DAG： 當問題可以轉換成DAG時，能有效處理依賴關係並計算每個vertex的最早完成時間。
        - 多個起點：當存在多個source時，可以同時處理多個起點。
        - 數據大：適用於 n 和 m 很大時，因為時間複雜度為 O(n + m)，可以在合理的時間內進行計算。

(2) **DFS**

- 單個起點： 當僅存在單個起點時，DFS 可以有效地遍歷整個圖。
- 尋找路徑：如果需要找到從一個節點到另一個節點的所有路徑，DFS 是一個好的選擇。
- recursive structure：DFS 可以處理recursive的問題。

(3) **Dijkstra's algorithm**：

- Single Source Shortest Paths： 找到單個節點到圖中所有其他節點的最短路徑。
- no negative weight：無法處理權重是負數的最短路徑，最短路徑不存在，只要不斷走negative的地方就會一直減少。

- Challenges you encountered

(1) 不熟悉使用adjacent list

一開始沒有要建立adjacent list的觀念，自己模擬output的時候只是大概知道實作的辦法，但化成code的時候就覺得會非常麻煩，最一開始還寫出了慢慢找的funtion來確定所有指入的relation都沒有了：

```
bool hasNoRelations(int node, const vector<vector<int>>& relations) {
    auto it = find_if(relations.begin(), relations.end(), [node](const
vector<int>& relation) {
        return relation[1] == node;
    });
    return it == relations.end();
}
```

但這樣的效率極差，因為害怕會不小心處理到還沒有得出最優解的vertex，所以一直不敢在update數值（讓其維持在-1），但其實只要找個地方存就好了，最後是選擇用一個queue來存。

(2) 無法分析題目

在先備知識不充足的狀況下，我不太清楚要怎麼分類題目跟下關鍵字去知道有什麼演算法可以用，我一開始腦袋想到的其實就是接近Dynamic Programming 跟Topological sorting 的做法，但我好像也只能告訴ChatGPT當它分析出答案之後我才發現我的做法跟哪些比較類似。後來雖然去看了一些Dynamic Programming的資料，但去Leetcode上面看相關的題目我覺得我暫時還是沒辦法寫出來。

(3) 不熟悉C++的標準庫函數

我在使用STL中的queue, vector還有這次有用到的find時每次都要重新查過（回傳的是value還是iterator常常忘記），希望未來能靈活運用標準庫函數來提高效率。