# 一、 程式摘要

## 1. 邏輯/原理

「給定 N 個工作，每個工作由編號、開始時間、結束時間及產值構成，找出最大產值的集合且時間不能重疊」，我們以下列方式實作。

### ✓ 最大產值

若今天只是單純的 Activity Selection Problem，我們可以使用 Greedy 的解法，不考慮結果，僅找尋當下看似最佳的選擇。但因為本題有加上產值的變因，每次的選擇並不一定會是最佳解（即當前最大產值），不能保證有 Greedy Choice Property，因此需要改用 Dynamic Programming 的解法。我們先將工作以結束時間作排序，並建立大小為 N 的表格，將產值初始化至表格中，接著逐一判斷工作與工作之間的時間是否重疊，若不重疊則將當前兩工作的產值相加，並與表格中的值作比較，若較大則更新至表格中，最終表格就會有完整的產值累加記錄。而在過程中我們可再以變數 max_value 記錄當前最大產值，將該變數與表格中的值持續作比較，若較大則更新該變數的值，最終即可得到最大產值。

### ✓ 最大產值編號集合

前述邏輯的反向操作，從表格中擁有最大產值的索引開始，不斷往前推算前一個工作在哪，同時記錄工作編號，最後將編號排序完輸出即可。

## 2. 語言

以 C 語言實作。

# 二、 程式內容說明

## 1. 程式註解

```c
#include <stdio.h>
#include <stdlib.h>

// 工作個數最大值常數化
#define MAX_ACTIVITY 1000
// 工作屬性個數常數化
#define ACTIVITY_STRUCT_NUM 4

// 裝填工作的結構
struct activity {
    int number;
    int start_time;
```

```
        int finish_time;
        int value;
};
// 裝填工作集合的結構
struct activity_blob {
        struct activity list[MAX_ACTIVITY];
        int size;
};
// 輸出結果用的結構
struct activity_result {
        int max_value;
        int numbers[MAX_ACTIVITY];
        int size;
};

// 比較 int 大小的 funciton
int find_max(int a, int b) {
        return (a > b) ? a : b;
}

// qsort 所需比較 int 大小的 funciton
int compare(const void *a, const void *b) {
        return *(int *) a - *(int *)b;
}
// qsort 所需比較 activity 結構大小(以 finish_time 為基準)的 funciton
int activity_compare(const void *a, const void *b) {
        struct activity *acty_a = (struct activity *)a;
        struct activity *acty_b = (struct activity *)b;
        return (acty_a->finish_time - acty_b->finish_time);
}
// 計算 activity selection problem 的 function
struct activity_result find_activity(struct activity list[], int size) {

        // 以 qsort 先對工作集合進行排序
        qsort(list, size, sizeof(struct activity), activity_compare);

        // 宣告結果結構
        struct activity_result result;
        // 宣告變數，用來儲存當前最大產值及其索引
        int max_value = list[0].value;
        int max_index = 0;
```

```
    // 宣告一陣列，用來儲存產值的變化 (DP)
    // 將產值初始化至表格中
    int temp[size];
    for (int i = 0; i < size; i++) {
        temp[i] = list[i].value;
    }

    // 計算最大產值
    for (int i = 1; i < size; i++) {
        for (int j = 0; j < i; j++) {
            if (list[i].start_time >= list[j].finish_time) {
                temp[i] = find_max(temp[i], temp[j] + list[i].value);
            }
            if (temp[i] > max_value) {
                max_value = temp[i];
                max_index = i;
            }
        }
    }
    // 裝填最大產值
    result.max_value = max_value;

    // 初始化工作編號集合大小
    int count = 0;
    // 初始化暫存用的工作索引
    int temp_index = max_index;
    // 蒐集最大產值的工作編號集合
    for (int i = max_index; i >= 0 ; i--) {
        if (temp[i] == max_value) {
            if (i == max_index || list[i].finish_time <= list[temp_index].start_time) {
                max_value -= list[i].value;
                result.numbers[count] = list[i].number;
                count++;
                temp_index = i;
            }
        }
    }
    // 將蒐集好的工作編號集合進行排序
    qsort(result.numbers, count, sizeof(int), compare);
    // 裝填工作編號集合大小
    result.size = count;
    // 回傳結果
```

```
    return result;
}
// main funciton
int main() {
    // 輸入測資筆數
    int input_num;
    scanf("%i", &input_num);

    // 宣告裝填多個工作集合的結構
    struct activity_blob blob[input_num];
    // 裝填每一筆測資
    for (int i = 0; i < input_num; i++) {
    // 輸入該筆測資所需的工作集合大小
        int activity_num;
        scanf("%i", &activity_num);
        // 裝填工作集合
        for (int j = 0; j < activity_num; j++) {
            // 輸入工作
            int activity_property[ACTIVITY_STRUCT_NUM];
            for(int k = 0; k < ACTIVITY_STRUCT_NUM; k++) {
                scanf("%d", &activity_property[k]);
            }
             // 裝填工作
            blob[i].list[j].number = activity_property[0];
            blob[i].list[j].start_time = activity_property[1];
            blob[i].list[j].finish_time = activity_property[2];
            blob[i].list[j].value = activity_property[3];
        }
        // 裝填該工作集合的大小
        blob[i].size = activity_num;
    }

    // 將裝填的測資拿出來一筆一筆執行
    for (int i = 0; i < input_num; i++) {
        // 呼叫運算 funciton，將工作集合及集合大小作為參數傳入
        struct activity_result result = find_activity(blob[i].list, blob[i].size);
        // 印出結果
        printf("%d\n", result.max_value);
        for (int i = 0; i < result.size; i++) {
            printf("%d", result.numbers[i]);
            if (i < result.size - 1)
                printf(" ");
```

```
            }
            printf("\n");
        }
    }
```

## 2. 圖解

### ✓ 最大產值

我們以工作集合 A = {{1, 1, 3, 5}, {2, 2, 5, 6}, {3, 4, 6, 5}, {4, 6, 7, 4}} 為例，以下圖說明取得最大產值的過程。

（工作結構定義為｛編號 number, 開始時間 start_time, 結束時間 finish_time, 產值 value｝；T 為建立 Dynamic Programming 所需表格，並將產值初始化至其中；i 為比較基準的工作索引；j 為該基準以前的工作索引）

(a)

| | j | i | | |
|---|---|---|---|---|
| (start_time , finish_time) | (1 , 3) | (2 , 5) | (4 , 6) | (6 , 7) |
| value | 5 | 6 | 5 | 4 |
| T | 5 | 6 | 5 | 4 |

A[i].start_time < A[j].finish_time

(b)

| | j | | i | |
|---|---|---|---|---|
| (start_time , finish_time) | (1 , 3) | (2 , 5) | (4 , 6) | (6 , 7) |
| value | 5 | 6 | 5 | 4 |
| T | 5 | 6 | 10 | 4 |

A[i].start_time >= A[j].finish_time
T[i] = T[j] + A[j].value = 5 + 5 = 10

(c)

| | | j | i | |
|---|---|---|---|---|
| (start_time , finish_time) | (1 , 3) | (2 , 5) | (4 , 6) | (6 , 7) |
| value | 5 | 6 | 5 | 4 |
| T | 5 | 6 | 10 | 4 |

A[i].start_time < A[j].finish_time

(d)

| | j | | | i |
|---|---|---|---|---|
| (start_time , finish_time) | (1 , 3) | (2 , 5) | (4 , 6) | (6 , 7) |
| value | 5 | 6 | 5 | 4 |
| T | 5 | 6 | 10 | 9 |

A[i].start_time >= A[j].finish_time
T[i] = T[j] + A[j].value = 5 + 4 = 9

(e)

| | | j | | i |
|---|---|---|---|---|
| (start_time , finish_time) | (1 , 3) | (2 , 5) | (4 , 6) | (6 , 7) |
| value | 5 | 6 | 5 | 4 |
| T | 5 | 6 | 10 | 10 |

A[i].start_time >= A[j].finish_time
T[i] = T[j] + A[j].value = 6 + 4 = 10

(f)

| | | | j | i |
|---|---|---|---|---|
| (start_time , finish_time) | (1 , 3) | (2 , 5) | (4 , 6) | (6 , 7) |
| value | 5 | 6 | 5 | 4 |
| T | 5 | 6 | 10 | 14 |

A[i].start_time >= A[j].finish_time
T[i] = T[j] + A[j].value = 10 + 4 = 14
max_value = 14

✓ **最大產值編號集合**

我們依然以工作集合 A = {{1, 1, 3, 5}, {2, 2, 5, 6}, {3, 4, 6, 5}, {4, 6, 7, 4}} 為例，以下圖說明蒐集最大產值編號集合的過程。
（根據前述範例，最大產值 max_value 為 14；i 為比較基準的工作索引，從 max_value 所在的工作索引開始，逐一往前尋找；j 為暫存使用，如果找到目標工作，則儲存該工作索引）

(a)

| | | | | i j |
|---|---|---|---|---|
| number | 1 | 2 | 3 | 4 |
| (start_time , finish_time) | (1 , 3) | (2 , 5) | (4 , 6) | (6 , 7) |
| value | 5 | 6 | 5 | 4 |
| T | 5 | 6 | 10 | 14 |

| | | | |
|---|---|---|---|
| 4 | 0 | 0 | ... |

max_value = T[i] = 14
max_value = 14 – 4 = 10
j = 3

(b)

| | | | i | j |
|---|---|---|---|---|
| number | 1 | 2 | 3 | 4 |
| (start_time , finish_time) | (1 , 3) | (2 , 5) | (4 , 6) | (6 , 7) |
| value | 5 | 6 | 5 | 4 |
| T | 5 | 6 | 10 | 14 |

| | | | |
|---|---|---|---|
| 4 | 3 | 0 | … |

max_value = T[i] = 10
A[j].start_time >= A[i].finish_time
max_value = 10 – 5 = 5
j = 3 → 2

(c)

| | | i | j | |
|---|---|---|---|---|
| number | 1 | 2 | 3 | 4 |
| (start_time , finish_time) | (1 , 3) | (2 , 5) | (4 , 6) | (6 , 7) |
| value | 5 | 6 | 5 | 4 |
| T | 5 | 6 | 10 | 14 |

| | | | |
|---|---|---|---|
| 4 | 3 | 0 | … |

max_value != T[i]
max_value = 5
j = 2

(d)

| | i | | j | |
|---|---|---|---|---|
| number | 1 | 2 | 3 | 4 |
| (start_time , finish_time) | (1 , 3) | (2 , 5) | (4 , 6) | (6 , 7) |
| value | 5 | 6 | 5 | 4 |
| T | 5 | 6 | 10 | 14 |

| | | | |
|---|---|---|---|
| 4 | 3 | 1 | ... |

max_value = T[i] = 5
A[j].start_time >= A[i].finish_time
max_value = 5 – 5 = 0
j = 2 → 0

**3. 虛擬碼**

```
FIND_ACTIVITY(activities, size)
    // 將工作集合排序 (依工作結束時間)
    QSORT(activities, activity_compare)

  // 初始化最大產值
  max_value = activities[0].value

  // 表格 temp 用來儲存產值在每個工作階段的變化
  // 將產值初始化至表格中
  for i = 0 to size
      temp[i] = activities[i].value

  // 計算最大產值
  // 以第 i + 1 筆工作為基準
  for i = 1 to size
      // 將該筆工作之前的工作拿出來比較
      for j = 0 to i
        // 判斷兩工作時間是否重疊
        // 即當前工作的開始時間必須大於等於前一筆工作的結束時間
        if activities[i].start_time >= activities[j].finish_time
          // 將表格中前一筆工作所累計的產值取出
          // 與當前工作的產值相加，即為當前累計的產值
          // 與當前工作的產值作比較，將產值較大者更新至表格中
```

```
            temp[i] = FIND_MAX(temp[i], temp[j] + activities[i].value)
        // 更新最大產值
        max_value = FIND_MAX(temp[i], max_value)

// 蒐集最大產值的工作編號集合
// 初始化工作編號集合
num[MAX_ACTIVITY];
// 初始化暫存用的的工作索引
temp_index = size - 1

// 判斷邏輯為計算最大產值的反向路徑
// 從尾到頭逐一比較
for i = size - 1 to 0
        // 若從表格中取出的產值與當前累計產值相等
        // 同時，當前工作要與前一個比較對象的時間不重疊
        // 代表找到目標工作
        if temp[i] == max_value and activities[i].finish_time <= activities[temp_index].start_time
            // 當前累計產值減掉當前工作的產值，即會是下一筆目標工作所累計的產值
            max_value -= activities[i].value
            // 裝填工作編號
            num[i] = activities[i].number
            // 記錄當前工作索引 (下一筆工作將與此筆工作比較)
            temp_index = i

// 將蒐集好的工作編號集合進行排序
QSORT(num, compare)

// 回傳最大產值及其工作編號集合
return max_value, num
```