一、 程式摘要

1. **邏輯/原理**

求「一串數字中，連續區間中的和的最大值」，我們分別以下列兩種方式處理。

✓ **Recursive**

透過 Divide and Conquer 的概念，我們將數列（大問題）切割為左、右兩區段（子問題），分別找出並比較其最大值，得到的值我們稱為 max。另一方面，最大值也有可能橫跨左、右兩區段，因此仍需將左、右兩區段的值予以相加，得出來的值再與 max 做比較，最後大者即為我們要的結果。利用此方法我們需要不斷地切割問題並比較值的大小，因此適合以 Recursive 的方式來完成，而此方法的時間複雜度為 O(nlog n)。

✓ **Non-recursive**

透過簡單的 Iteration 以及配合 Dynamic Programming 的概念，我們可以利用一變數 sum 來紀錄當前的最大值，每當 loop 運算出更大的值即更新 sum，反之若出現小於等於 0 的情況，則將 sum 重新開始累積，此方法的時間複雜度為 O(n)。

2. **語言**

兩種方式皆以 C 語言實作。

二、 程式內容說明

1. 程式註解

✓ **Recursive**

```c
#include <stdio.h>

// 輸出結果用的結構
struct max_profit_result {
    int first_index;
    int last_index;
    int max_profit;
};

// 將比較的邏輯抽出來
struct max_profit_result find_max(struct max_profit_result a, struct max_profit_result b) {
    return (a.max_profit > b.max_profit) ? a : b;
}
```

```
// 計算 max_profit 的 function
struct max_profit_result find_max_profit(int input[], int left, int right) {
    // 初始化結果結構
    struct max_profit_result result;
    result.first_index = left;
    result.last_index = right;
    result.max_profit = input[left];

    // 若左區段與右區段的索引一樣，則直接回傳
    if (left == right)
        return result;

    // 先算出中間點索引
    int middle = (left + right) / 2;

    // 左區段的最小索引
    int left_left = left;
    // 左區段的最大索引
    int left_right = middle;

    // 右區段的最小索引
    int right_left = middle + 1;
    // 右區段的最大索引
    int right_right = right;

    // 以遞迴計算左區段的最大值
    struct max_profit_result left_result = find_max_profit(input, left_left, left_right);
    // 以遞迴計算右區段的最大值
    struct max_profit_result right_result = find_max_profit(input, right_left, right_right);

    // 初始化索引、左區段最大值、右區段最大值
    int sum = 0;
    int first_index = left_right;
    int last_index = right_left;
    int left_max_profit = input[left_right];
    int right_max_profit = input[right_left];

    // 開始計算左區段最大值
    for(int i = left_right; i >= left_left; i--) {
        sum += input[i];
        if (sum > left_max_profit) {
            left_max_profit = sum;
```

```
                first_index = i;
            }
        }

        // 開始計算右區段最大值
        // sum 先歸零
        sum = 0;
        for(int i = right_left; i <= right_right; i++) {
            sum += input[i];
            if (sum > right_max_profit) {
                right_max_profit = sum;
                last_index = i;
            }
        }

        // 裝填結果
        result.first_index = first_index;
        result.last_index = last_index;
        result.max_profit = left_max_profit + right_max_profit;

        // 比較左區段與右區段的大小，得出的結果再與橫跨區段的結果比較
        // 得到最大值後回傳結果
        return find_max(find_max(left_result, right_result), result);
}

// main funciton
int main() {
        int input_size;
        // 輸入測資個數，並在輸入值未讀完前持續執行
        while(scanf("%i", &input_size) != EOF) {
            int input[input_size];
            // 輸入測資
            for(int i = 0; i < input_size; i++) {
                scanf("%d", &input[i]);
            }
            // 呼叫運算 funciton，將最左邊及最右邊的索引作為參數傳入
            struct max_profit_result result = find_max_profit(input, 0, input_size - 1);
            // 印出結果
            printf("%d %d %d\n", result.first_index, result.last_index, result.max_profit);
        }
}
```

✓ **Non-recursive**

```
#include <stdio.h>

// 輸出結果用的結構
struct max_profit_result {
	int first_index;
	int last_index;
	int max_profit;
};

// 計算 max_profit 的 function
struct max_profit_result find_max_profit(int input[], int input_size) {
	// 初始化索引及儲存最大值的變數
	int sum = 0;
	int first_index = 0;
	int new_first_index = 0;
	int last_index = 0;
	int max_profit = 0;

	// 開始計算，當找到更大的累加值便更新最大值
	for (int i = 0; i < input_size; i++) {
		int temp = sum + input[i];
		if (temp <= 0) {
			sum = 0;
			new_first_index = i + 1;
		} else {
			sum = temp;
		}

		if (sum > max_profit) {
			max_profit = sum;
			last_index = i;
			first_index = new_first_index;
		}
	}

	// 裝填結果
	struct max_profit_result result;
	result.first_index = first_index;
	result.last_index = last_index;
	result.max_profit = max_profit;
	// 回傳結果
```

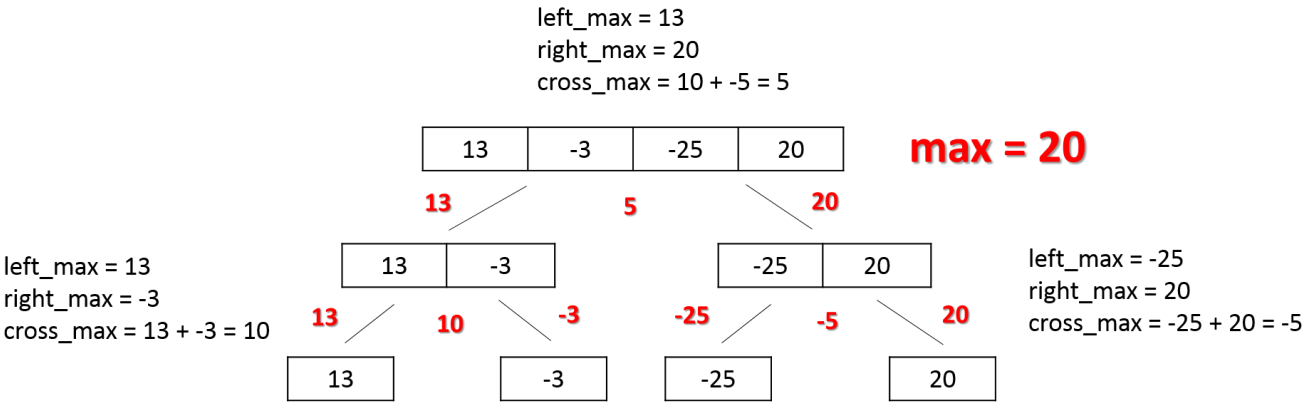
```
    return result;
}

// main funciton
int main() {
    int input_size;
    // 輸入測資個數，並在輸入值未讀完前持續執行
    while(scanf("%i", &input_size) != EOF) {
        int input[input_size];
        // 輸入測資
        for(int i = 0; i < input_size; i++) {
            scanf("%d", &input[i]);
        }
        // 呼叫運算 funciton，將最左邊及最右邊的索引作為參數傳入
        struct max_profit_result result = find_max_profit(input, input_size);
        // 印出結果
        printf("%d %d %d\n", result.first_index, result.last_index, result.max_profit);
    }
}
```

2. 圖解

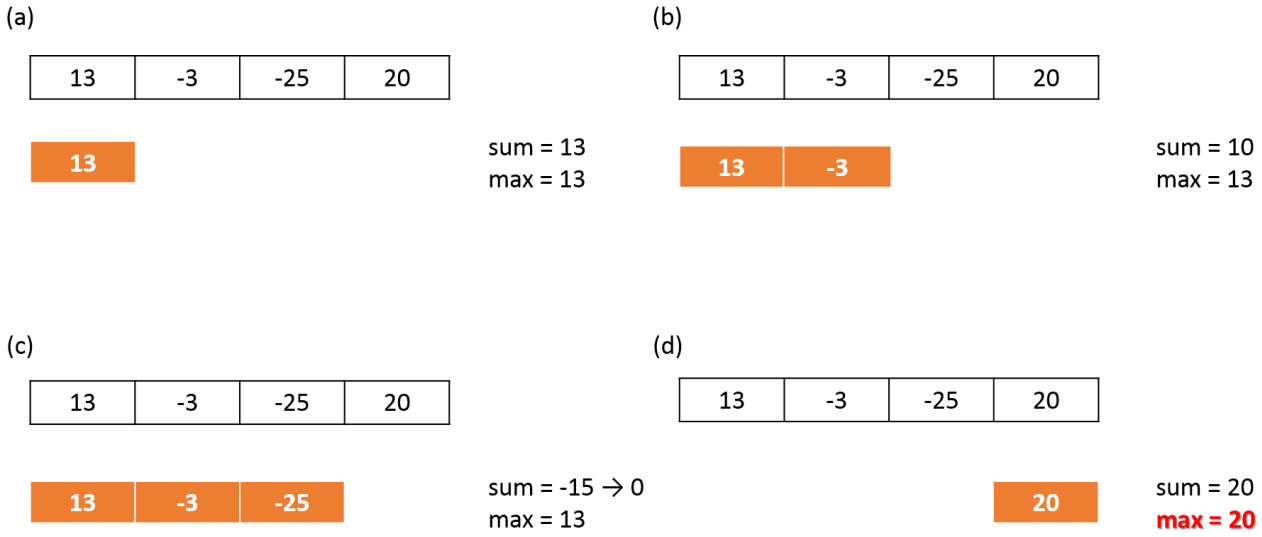
✓ Recursive

我們以[13, -3, -25, 20]為例，以下圖說明其最大連續整數和。



✓ **Non-recursive**

我們以[13, -3, -25, 20]為例，以下圖說明其最大連續整數和。



3. 虛擬碼

✓ **Recursive**

```
FIND_MAX_PROFIT(input, left, right)
    // 計算中間點索引
    middle = (left + right) / 2
    // 以遞迴計算左區段的最大值
    left_result = FIND_MAX_PROFIT(input, left, middle)
    // 以遞迴計算右區段的最大值
    right_result = FIND_MAX_PROFIT(input, middle + 1, right)

    // 暫存目前累加的值
    sum = 0
    // 計算左區段最大值
    for left to middle
        // 累加
        sum = sum + input[i]
        // 當累加的值大於當前左區段最大值
        if sum > left_max_profit
            // 將當前左區段最大值更新
            left_max_profit = sum
            // 紀錄起始索引
            first_index = i

    // sum 先歸零
    sum = 0
```

```
    // 計算右區段最大值
    for middle + 1 to right
        // 累加
        sum = sum + input[i]
        // 當累加的值大於當前右區段最大值
        if sum > right_max_profit
            // 將當前右區段最大值更新
            right_max_profit = sum
            // 紀錄結束索引
            last_index = i

    // 結果相加為橫跨兩區段的最大值
    cross_result = left_max_profit + right_max_profit
    // 比較左區段及右區段，並取最大值
    result = MAX(left_result, right_result)
    // 再與橫跨兩區段的 case 比較，並取最大值回傳
    return MAX(result, cross_result)
```

✓ **Non-recursive**

```
FIND_MAX_PROFIT(input, left, right)

    for i to input_size
        // 先算出目前累加的值
        temp = sum + input[i]
        // 如果累加值小於等於 0，則沒有留下來的必要
        if temp <= 0
            // 將累加值歸零
            sum = 0
            // 起始索引也要更新，因此先暫存起來
            new_first_index = i + 1
        // 反之，將目前累加值保存下來
        else
            sum = temp

        // 確定找到更大的累加值
        if sum > max_profit
            // 更新最大值
            max_profit = sum
            // 更新結束索引
            last_index = i
            // 更新起始索引
            first_index = new_first_index
```