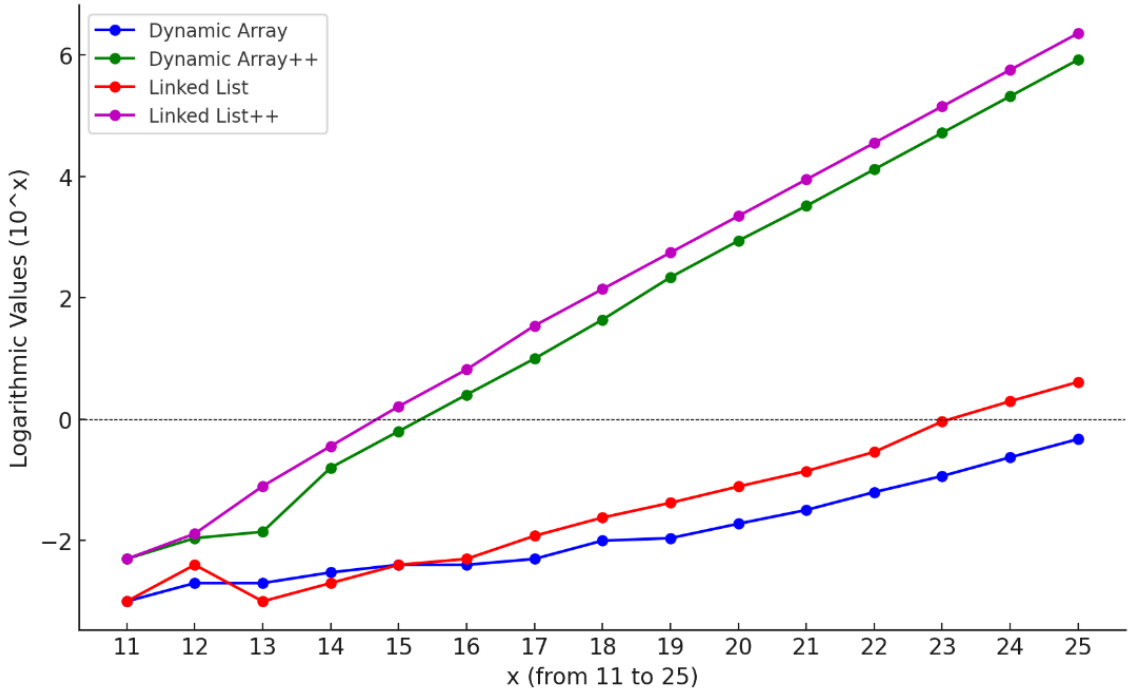
112703003 資訊二 黃柏淵 DS_HW1

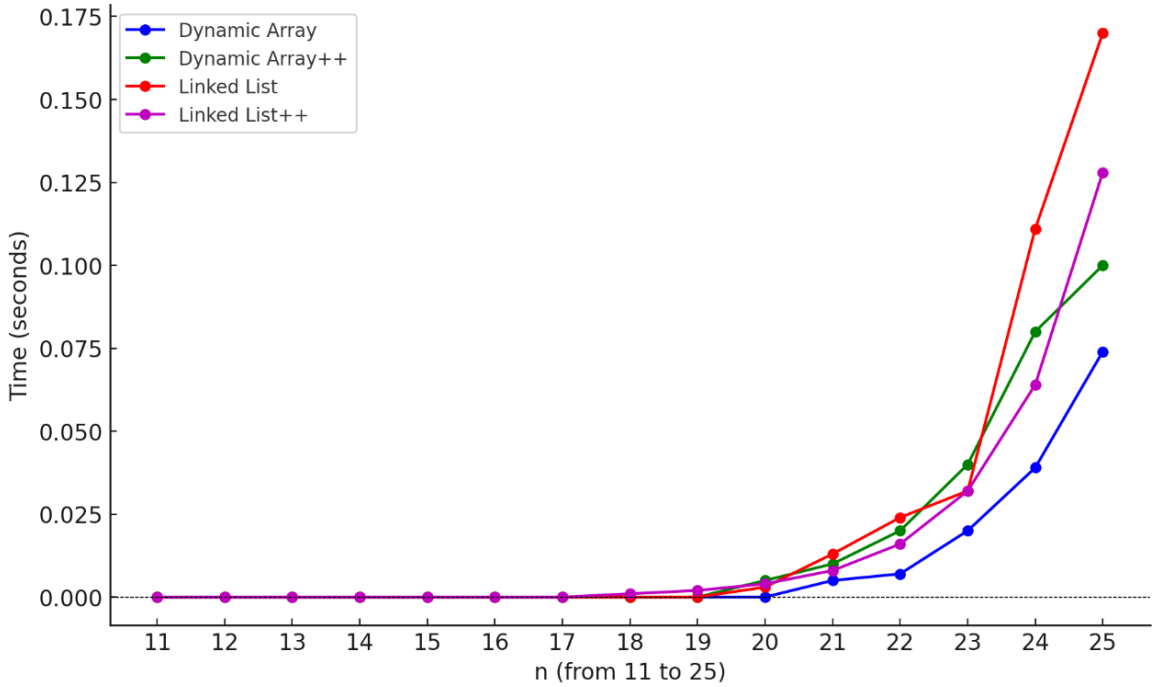
三張折線圖

第一張圖 ： 插入 n 筆資料所需時間



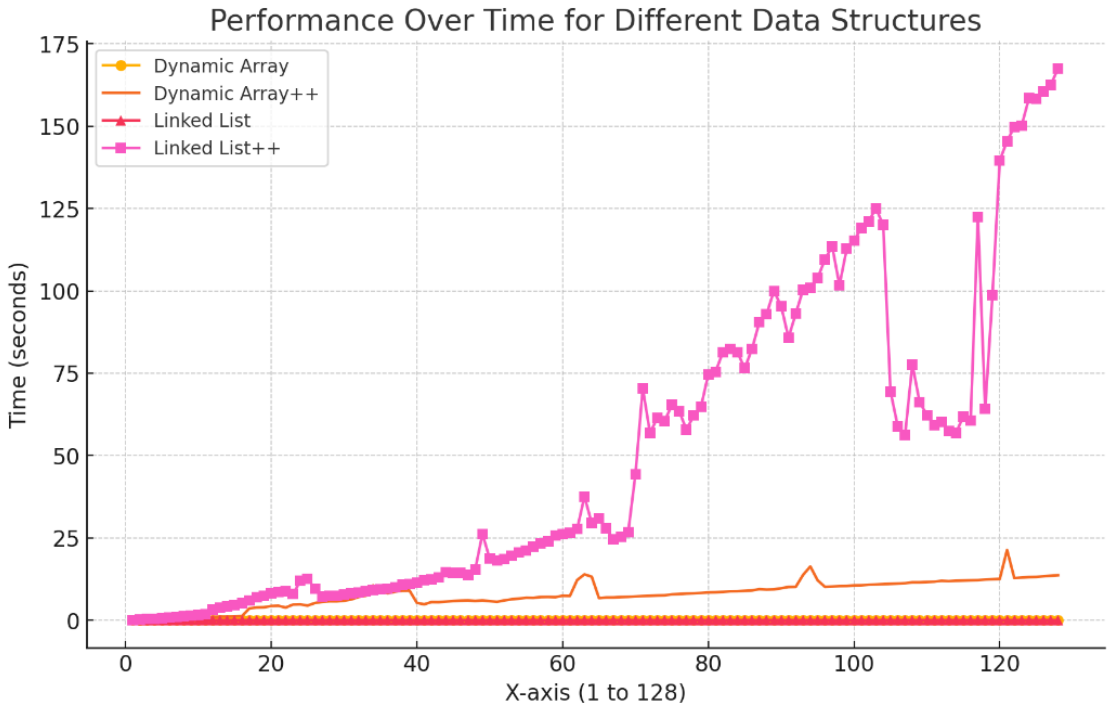
- Dynamic Array: 沒有用到估計，插入資料(resize)的時間複雜度為 O(n)。
- Dynamic Array++: 在 2 的 20 次方時就用到估計，因為插入資料的總時間複雜度是 O(n^2)。每次擴容只增加 1 個單位的容量，使得擴容次數較為頻繁。因此在每次擴容時，必須重新複製整個陣列，造成插入的總時間複雜度高。因此當插入資料變成兩倍時，時間會變成四倍。
- Linked List: 沒有用到估計，插入資料的總時間複雜度是 O(n)
- Linked List++: 在 2 的 18 次方時就需要估計，因為插入資料的總時間複雜度是 O(n^2)。每次插入一個元素時，必須遍歷鏈結部分或全部的串列來找到合適的插入位置。插入 n 個元素的總時間複雜度為：O(1) + O(2) + O(3) + … + O(n) = O(n ^ 2)。因此當插入資料變成兩倍時，時間上推論也會變成四倍。

第二張圖 ： 插入 n 筆資料後，計算資料結構內數值總和所需的時間



- Dynamic Array: 在計算資料結構內數字總和用到的時間複雜度是 O(n)，且插入資料的時間複雜度是 O(n)，因此不需要用到估計。
- Dynamic Array++: 雖然計算資料結構內數字總和的時間複雜度是 O(n)，但插入資料的時間複雜度是 O(n^2)，因為插入 2^20 個資料時的時間需要估計，所以計算資料結構內數字總和的時間也需要估計。
- Linked List: 在計算資料結構內數字總和用到的時間複雜度是 O(n)，且插入資料的時間複雜度是 O(n)，因此不需要用到估計。
- Linked List++: 雖然計算資料結構內數字總和的時間複雜度是 O(n)，但插入資料的時間複雜度是 O(n^2)，因為插入 2^20 個資料時的時間需要估計，所以計算資料結構內數字總和的時間也需要估計。

第三張圖 ： 每新增 2^13 筆資料所需時間



在第三張圖中，四個資料結構都不需用到估計時間。

資料結構程式碼與解釋

1. Dynamic Array

```
class DynamicArray {
    short* arr;
    long capacity;
    long size;

public:
    DynamicArray() : capacity(1), size(0) {
        arr = new short[capacity];
    }

    ~DynamicArray() {
        delete[] arr;
    }

    void push_back(short value) {
        if (size == capacity) {
            resize();
        }
        arr[size++] = value;
    }
```

```
    long long addUp() const {
        long long total = 0;
        for (long i = 0; i < size; i++) {
            total += arr[i];
        }
        return total;
    }

    long getSize() const {
        return size;
    }

    long getCapacity() const {
        return capacity;
    }
```

```
private:
    void resize() {
        long new_capacity = capacity * 2;  // 增加容量的方式改為乘以2
        short* new_arr = new short[new_capacity];
        for (long i = 0; i < size; i++) {
            new_arr[i] = arr[i];
        }
        delete[] arr;
        arr = new_arr;
        capacity = new_capacity;
    }
};
```

成員變數

- arr: 指向實際存儲數據的動態陣列。
- capacity: 表示動態陣列的當前容量。
- size: 表示當前陣列中的有效元素數量，初始值為 0。

建構函數和解構函數

push_back 函數

- 這個函數在陣列的尾部插入一個新值 value。如果當前 size 等於 capacity，會呼叫 resize 函數來擴展陣列的容量。
- 插入後，size 會遞增，表示新增了一個元素。

resize 函數

- 當容量不足時，會將容量擴大到當前的兩倍，然後動態分配新的陣列。
- 將舊陣列中的所有數據複製到新的陣列，釋放舊陣列的記憶體，並更新 arr 指向新的陣列。
- 將 capacity 更新為新的容量。

addUp 函數

- 這個函數會遍歷整個陣列，將所有元素的數值累加並返回總和。時間複雜度是 O(n)，其中 n 是陣列的元素數量。

getSize 和 getCapacity 函數

- getSize: 返回當前陣列中的有效元素數量。
- getCapacity: 返回當前陣列的容量。

2. Dynamic Array++

```
7  class DynamicArray {
8      short* arr;
9      long capacity;
10     long size;
11
12 public:
13     DynamicArray() : capacity(1), size(0) {
14         arr = new short[capacity];
15     }
16
17     ~DynamicArray() {
18         delete[] arr;
19     }
20
21     void push_back(short value) {
22         if (size == capacity) {
23             resize();
24         }
25         arr[size++] = value;
26     }
```

```
28     long long addUp() const {
29         long long total = 0;
30         for (long i = 0; i < size; i++) {
31             total += arr[i];
32         }
33         return total;
34     }
35
36     long getSize() const {
37         return size;
38     }
39
40     long getCapacity() const {
41         return capacity;
42     }
```

```
44  private:
45      void resize() {
46          long new_capacity = capacity + 1;
47          short* new_arr = new short[new_capacity];
48          for (long i = 0; i < size; i++) {
49              new_arr[i] = arr[i];
50          }
51          delete[] arr;
52          arr = new_arr;
53          capacity = new_capacity;
54      }
55  };
```

成員變數

- arr: 動態分配的陣列，用來存儲元素。
- capacity: 這個動態陣列目前的容量。
- size: 表示已經存入的元素數量，初始為 0。

建構函數和解構函數

push_back 函數

- push_back: 在陣列末尾插入一個元素。若當前大小達到容量限制，則呼叫 resize() 來擴展容量。

resize 函數

- 當容量不夠時，resize 函數會將容量增加 1，動態分配一個新的陣列，並將舊陣列的數據複製過來，然後釋放舊陣列的記憶體。

addUp 函數

- addUp: 遍歷所有已插入的元素，將其累加並返回總和。

getSize 和 getCapacity 函數

- getSize: 返回當前已存儲的元素數量。
- getCapacity: 返回當前陣列的容量。

3. Linked List

```
class Node {
public:
    short data;
    Node* next;

    Node(short val) : data(val), next(nullptr) {}
};

class LinkedList {
private:
    Node* head;
    Node* tail;  // 新增一個 tail 指標來追蹤鏈結串列的尾節點
    long size;
```

```
public:
    LinkedList() : head(nullptr), tail(nullptr), size(0) {}

    // 插入元素到鏈結串列的尾部
    void push_back(short value) {
        Node* newNode = new Node(value);  // 建立新節點
        if (tail == nullptr) {  // 如果鏈結串列為空
            head = tail = newNode;  // 新節點同時作為頭和尾
        } else {
            tail->next = newNode;  // 將目前的尾節點指向新節點
            tail = newNode;        // 更新 tail 為新節點
        }
        size++;  // 增加鏈結串列的大小
    }

    long long addUp() const {
        long long total = 0;
        Node* current = head;
        while (current != nullptr) {
            total += current->data;
            current = current->next;
        }
        return total;
    }
```

```
        long getSize() const {
            return size;
        }
    };
```

類別 Node

- Node 類代表鏈結串列的節點，每個節點包含兩個成員：
    1. data: 存儲節點的數據。
    2. next: 指向下一個節點的指標，如果是尾節點則為 nullptr。
- 建構函數接受一個數值 val，並將 next 設置為 nullptr（因為這是新創建節點，還沒有連接到其他節點）。

類別 LinkedList

- 成員變數
  1. head: 指向鏈結串列的頭節點。如果鏈結串列是空的，head 為 nullptr。
  2. tail: 指向鏈結串列的尾節點，這樣插入新節點時可以直接更新尾節點。
  3. size: 記錄鏈結串列中節點的數量。
- push_back 函數
  - ✓ push_back: 這個函數用來在鏈結串列的尾部插入新元素。首先創建一個新的節點叫 newNode。如果 tail 是 nullptr，表示鏈結串列是空的，則 head 和 tail 都指向這個新節點。否則，將當前尾節點的 next 指向新節點，然後更新 tail 為新節點。最後將鏈結串列的大小 size 增加 1。
- addUp 函數
  - ✓ addUp: 這個函數遍歷鏈結串列中的每個節點，將每個節點的 data 加到總和 total 中，最後返回總和。從 head 開始，遍歷所有節點，直到 current 變為 nullptr（即鏈結串列的尾部）。
- getSize 函數
  - ✓ getSize: 返回鏈結串列的大小，即節點數量。

4. Linked List++

```
8  class Node {
9  public:
10     short data;
11     Node* next;
12
13     Node(short val) : data(val), next(nullptr) {}
14 };
15
16 class LinkedList {
17 private:
18     Node* head;
19     long size;
```

```
public:
    LinkedList() : head(nullptr), size(0) {}

    // 插入元素並按數據排序
    void insert_sorted(short value) {
        Node* newNode = new Node(value);
        if (head == nullptr || head->data >= value) {
            // 如果鏈結串列為空或新節點數據比頭節點小，直接插入到頭部
            newNode->next = head;
            head = newNode;
        } else {
            // 否則找到適合的位置進行插入
            Node* current = head;
            while (current->next != nullptr && current->next->data < value) {
                current = current->next;
            }
            newNode->next = current->next;
            current->next = newNode;
        }
        size++;
    }
```

```
    long long addUp() const {
        long long total = 0;
        Node* current = head;
        while (current != nullptr) {
            total += current->data;
            current = current->next;
        }
        return total;
    }

    long getSize() const {
        return size;
    }
};
```

類別 Node

- Node 類代表鏈結串列的節點，每個節點包含兩個成員：
  1. data:存儲節點的數據。
  2. next: 指向下一個節點的指標，如果是尾節點則為 nullptr。
- 建構函數接受一個數值 val，並將 next 設置為 nullptr。

類別 LinkedList

- 成員變數
  1. head: 指向鏈結串列的頭節點。如果鏈結串列是空的，head 為 nullptr。
  2. size: 記錄鏈結串列中節點的數量。
- insert_sorted 函數
  - ✓ insert_sorted: 這個函數會將新節點按數據大小有序插入鏈結串列中。如果鏈結串列是空的，或新插入的數值比頭節點的數值小，直接將新

節點插入到鏈結串列的頭部。否則，遍歷鏈結串列，找到一個合適的位置，確保新節點插入後仍然保持鏈結串列的有序性。

- addUp 函數
  - ✓ addUp: 這個函數遍歷鏈結串列中的每個節點，將每個節點的 data 加到總和 total 中，最後返回總和。從 head 開始，遍歷所有節點，直到 current 變為 nullptr（即鏈結串列的尾部）。
- getSize 函數
  - ✓ getSize: 返回鏈結串列的大小，即節點數量。

5. 執行實驗的主函式

```
57  void push(DynamicArray& d, int amount) {
58      clock_t start = clock();
59      for (long i = 0; i < amount; i++) {
60          d.push_back(rand() % 10);
61      }
62      clock_t end = clock();
63      // 計算時間並轉換為秒
64      double duration = double(end - start) / CLOCKS_PER_SEC;
65      cout << "push took: " << duration << " seconds" << endl;
66  }
67
68  void addUp(DynamicArray& d) {
69      clock_t start = clock();
70      long long total = d.addUp();
71      clock_t end = clock();
72      double duration = double(end - start) / CLOCKS_PER_SEC;
73      cout << "addUp took: " << duration << " seconds, result: " << total << endl;
74  }
```

```
76  // 插入每次 2^13 筆資料，總共進行 128 次
77  /*void push(DynamicArray& d) {
78      const int chunk = pow(2, 13);  // 每次插入的資料數量
79      const int total_batches = 128; // 總共插入 128 次
80      clock_t start, end;
81
82      for (int i = 0; i < total_batches; i++) {
83          start = clock();  // 記錄插入開始時間
84          for (int j = 0; j < chunk; ++j) {
85              d.push_back(rand() % 10);
86          }
87          end = clock();  // 記錄插入結束時間
88          // 計算每次插入所花費的時間並轉換為秒
89          double duration = double(end - start) / CLOCKS_PER_SEC;
90          cout << "Batch " << i + 1 << " (Inserted " << (i + 1) * chunk
91              << " elements) took: " << duration << " seconds" << endl;
92      }
93  }*/
94
95  void run(int k) {
96      long n = pow(2, k);
97      cout << endl;
98      cout << "k = " << k << ", n = " << n << endl;
99      DynamicArray d;
100     //LinkedList v;
```

```
102         push(d, n);
103         addUp(d);
104     }
105
106 int main() {
107     srand(time(NULL));
108     for (int i = 11; i <= 25; i++) {
109         run(i);
110     }
111 }
```

- push 和 addUp 函式使用 clock()函數來測量整個插入跟加總過程所花費的時間。

- 註解掉的 push 函數，總共會執行 128 次插入，每次插入 2^13 = 8192 個隨機數據。在每次插入後，記錄這次插入的時間。

- run 函數根據給定的 k 計算 n = 2 ^ k，然後測試插入和計算總和的過程。

- 創建一個 Dynamic Array 或是 Linked List，然後向其中插入 n 個隨機數據，並計算這些數據的總和。顯示 k 和對應的 n，並呼叫 push 函數進行插入，之後呼叫 addUp 進行總和的計算。

- main 主程式中，使用 srand(time(NULL))初始化隨機數生成器，確保每次執行程式時隨機數的序列都不同。從 k=11 到 k=25 執行 run 函數，逐步增加插入的數據量並測量插入和加總操作的時間。

解釋實驗圖

- 第一張圖是四種資料結構插入資料量由 2^11 到 2^25 時所花費的時間。
    1. Dynamic Array: 當數組達到容量上限時，動態數組會進行擴展，將數組的容量增大到原來的兩倍，擴展操作的時間複雜度是 O(n)。通常的插入操作（不發生擴展時）時間複雜度是 O(1)，只有在發生擴展時，時間複雜度才會是 O(n)。因此動態數組的插入操作時間會變成兩倍，是因為發生了擴展操作。
    2. Dynamic Array++: 在最壞情況下，對於每個元素 i，都需要複製 i 個元素。插入 n 個元素的總時間複雜度是 O(n ^ 2)。因此當插入資料量為兩倍時，時間會變成四倍。
    3. Linked List: 在 insert 函數中，使用了 push_back 函數 n 次，其中 n 是插入的元素數量。每次插入的時間複雜度是 O(1)，因此總插入時間複雜度為 O(n)，操作時間也會變成兩倍。

4. Linked List++: 對於每次插入操作，要從頭節點開始遍歷鏈結串列，直到找到適合的位置插入新元素。在 insert 函數中，執行了 n 次插入操作。對於第 i 次插入，鏈結串列的長度為 i，所以最壞情況下的插入時間複雜度為 O(i)，而插入所有 n 個元素的總時間複雜度為 O(n ^ 2)。因此當插入資料量為兩倍時，時間會變成四倍。

統整上述四個資料結構的性質跟第一張折線圖，可以得出 Dynamic Array 跟 Linked List 的折線比較平緩、斜率小，而 Dynamic Array++跟 Linked List++的折線較陡峭、斜率大。

- 第二張圖是遍歷資料結構並相加所需的時間
    1. Dynamic Array: 加總的時間複雜度是 O(n)，與數組的大小呈線性關係。
    2. Dynamic Array++: 加總操作的時間複雜度是 O(n)，其時間複雜度與數組的大小成線性關係。
    3. Linked List: addUp 需要遍歷鏈結串列中的每個節點，因此時間複雜度是 O(n)，時間複雜度與鏈結串列的大小呈線性關係。
    4. Linked List++: 加總部分的時間複雜度是 O(n)，因為需要遍歷鏈結串列中的所有節點，並對每個節點的數據進行累加操作。隨著鏈結串列的大小增大，加總操作的時間會線性增長。

    綜合四個資料結構的性質跟第二張圖，四條折線的趨勢是相同的，但在增加的速度上，仍有些微差異，Linked List 相對於 Dynamic Array 的操作速度快一些，在 Linked List 中，插入新元素，特別是在頭部或尾部的插入操作，通常是 O(1)的時間複雜度。如果有 tail 指標，在尾部插入元素的操作同樣是 O(1)。並且 Linked List 是動態分配內存的結構，每個節點可以獨立分配內存，不需要像動態數組一樣使用連續的內存空間。

- 第三張圖是每新增 2^13 筆資料所花費的時間
    1. Dynamic Array 跟 Linked List 所花費的時間都接近 0 秒，因為每次插入的資料量想對較小，動態擴容很少發生。此外可以被快取處理，無需頻繁訪問較慢的主記憶體。
    2. Dynamic Array++: 每新增一次資料的時間略多於 Dynamic Array 跟 Linked List，因為每次容量滿時，只會擴展 1 個位置，導致頻繁執行擴容和元素複製操作，所以插入操作的時間複雜度趨近於 O(n ^ 2)，因此效能會大幅降低。
    3. Linked List++: 需要在每次插入時從頭開始遍歷鏈結串列，直到找

到適合的插入位置。由於鏈結串列沒有隨機存取，必須依賴線性遍歷來找到插入位置。因此，插入操作的時間複雜度會隨著鏈結串列的長度增加而增加，最終導致每次插入的時間大幅增長，這也是四個資料結構中花最多時間的一個。

遇到的問題

1. 在第二張圖，Dynamic Array++跟 Linked List++，在可以跑出數據的時候都是 0 秒，這會讓估計難度增加，因為 0 乘上多少倍都會是 0。
2. 在第三張圖中，Linked List++那條，在 60 到 80 次之間以及 100 到 120 之間，插入資料的時間有大幅度的上下落差，這部分目前沒有獲得解答。