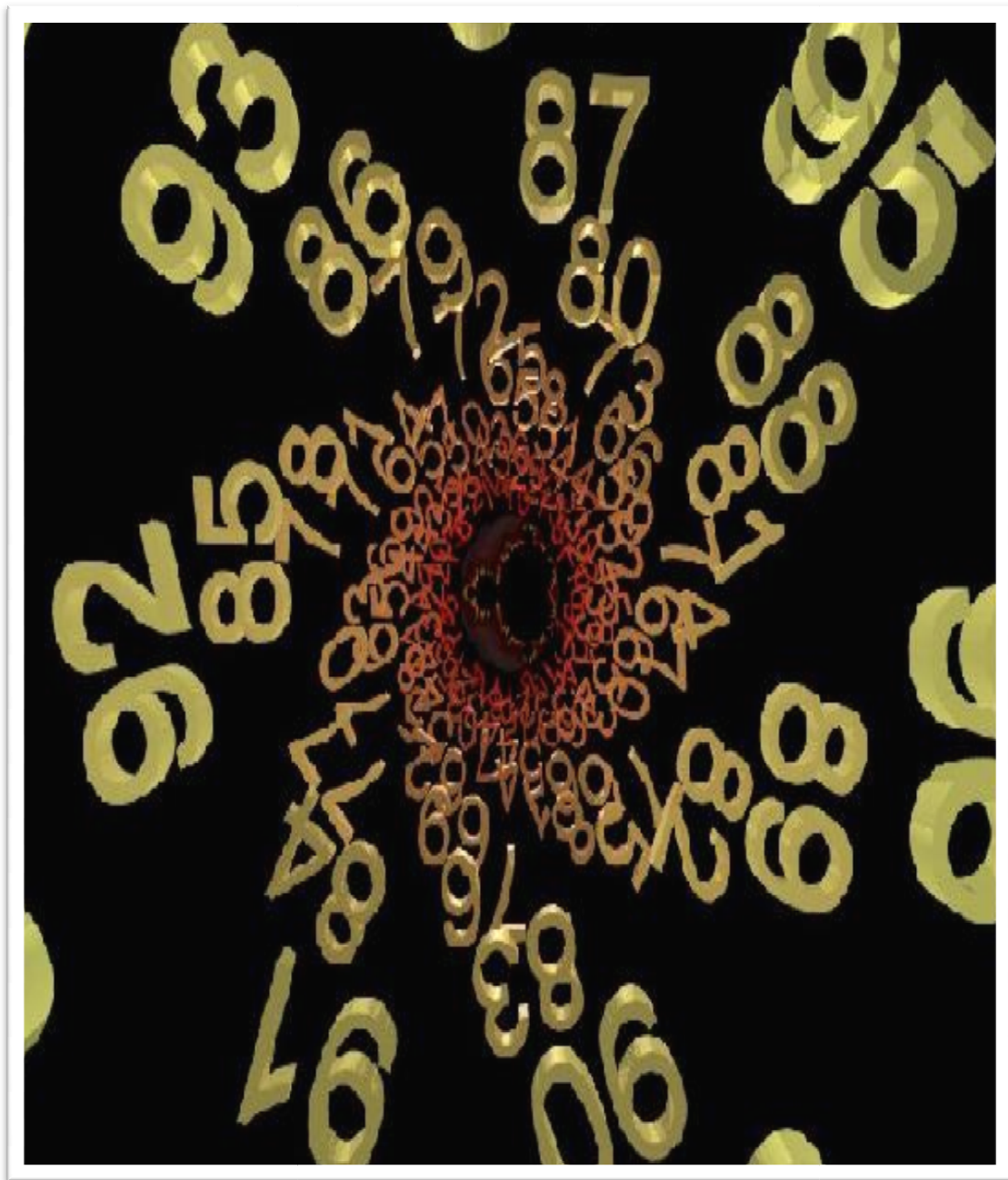




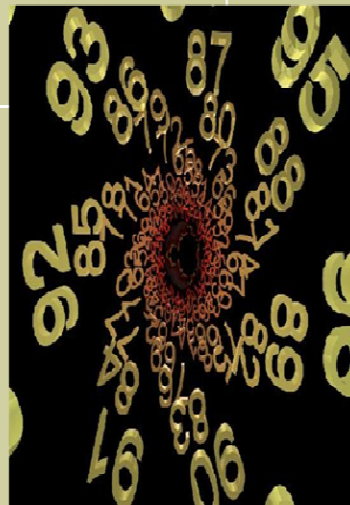
Assembly Final Project —

Longest Increasing Subsequence

Performance improvement with optimized assembly code

b92902121 郭涵煦

Assembly Final Project



Index:

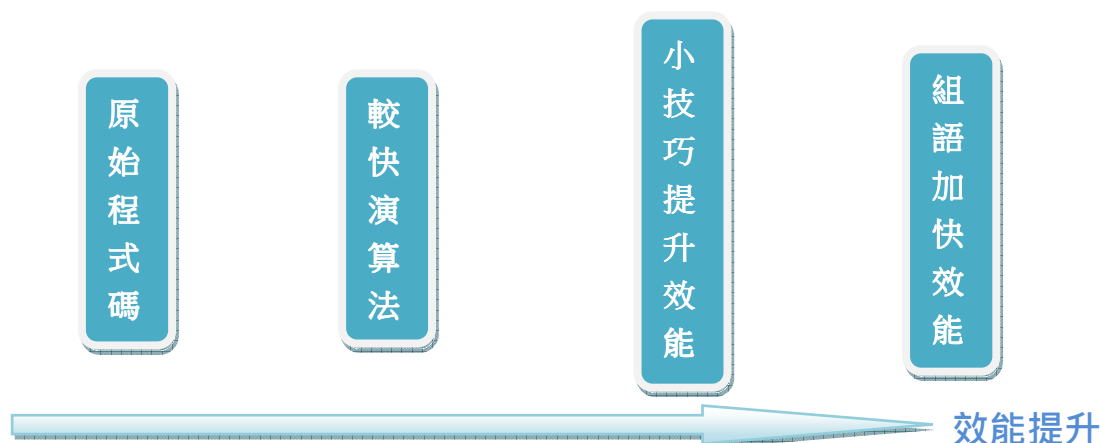
- 一、 Research motivation
- 二、 LIS introduction
- 三、 LIS algorithm
- 四、 Performance bottleneck and assembly solution
- 五、 Performance Evaluation

一、Research Motivation

還記得大二時修了演算法課程，對於利用快速的演算法來提升許多程式的效能十分有興趣。以當時的能力所及，要改善特定程式的效能，除了採用一個較快速的演算法以外，常用的方法不外乎是 Loop Unrolling、重新安排迴圈的內外層順序以充分使用 Cache 效能等少數幾種方法。至於要減少記憶體存取次數、活用暫存器、或是充分使用 CPU 製造商所提供的指令集等組合語言層級的方法，則是直到這學期上了組語課程之後，才學到的方法。

很想了解以前在演算法課程所作的作業，是否能採用組語來使效能最佳化，所以便挑選了以前作業當中，自己十分有興趣的一個作業 “Longest Increasing Subsequence(LIS)” 來當作本次要改善的目標程式，希望透過改寫其中重要的程式碼，來達成提高執行速度的成果。

本報告一開始從 LIS 的基本觀念出發，接著進一步介紹一般求取 LIS 可用的演算法，以及其實作之 C 語言程式。接著開始討論使用 C 語言程式編譯之後所可能產生的效能瓶頸，以及採用 Intel X86 組合語言後可提升效能之方式，最後則實際使用不同大小的測資，來評量其改善的效能。



二、LIS Introduction

最長遞增子數列定義如下：

給予一整數數列 $X = \{X_1, X_2, X_3, \dots, X_m\}$ ，其最長遞增子數列 LIS

$$LIS = \{L_1, L_2, L_3, \dots, L_n \mid L_1 \leq L_2 \leq L_3 \leq \dots \leq L_n\}$$

為滿足以上關係之 X 之子數列中長度最長者。

例如：

數列 $\{3, 1, 3, 2\}$ 其 LIS 為 $\{1, 3\}$

數列 $\{5, 5, 18, 37, 4, 13\}$ 其 LIS 為 $\{5, 5, 18, 37\}$

數列 $\{9, 15, 7, 6, 11, 12, 4\}$ 其 LIS 為 $\{9, 11, 12\}$

三、LIS Algorithm

1. 基礎想法：

一般來說，解決 LIS 問題所採用的演算法為 Dynamic Programming(DP)。

舉例來說，假若我們要求出數列 $\{7, 1, 5, 2, 4\}$ 的最長遞增數列長度，則可使用

以下表格來實作：

- 首先視同整個數列只有 7 這個數字，則在目前 LIS 長度=1 的欄位紀錄數字 7

值	7	1	5	2	4
---	---	---	---	---	---

長度	0	1	2	3	4	5
值	0	7	N	N	N	N

- 接著輪到 1 這個數字，由於遞增數列中 1 無法接在 7 的後面，而 1 又比 7

有更大的潛力被大於(大於 7 者必大於 1)，所以將 LIS 長度=1 的欄位紀錄

為 1

值	7	1	5	2	4
---	---	---	---	---	---

長度	0	1	2	3	4	5
值	0	1	N	N	N	N

- 接下來是 5，由於 5 可以接在 1 後面成為遞增數列，所以在 LIS 長度=2 的

欄位紀錄上數字 5

值	7	1	5	2	4
---	---	---	---	---	---

長度	0	1	2	3	4	5
值	0	1	5	N	N	N

- 再來是 2，以同樣方法將 LIS 長度=2 的欄位寫上數字 2

值	7	1	5	2	4
---	---	---	---	---	---

長度	0	1	2	3	4	5
值	0	1	2	N	N	N

- 最後是數字 4，在 LIS 長度=3 的欄位寫上數字 4

值	7	1	5	2	4
---	---	---	---	---	---

長度	0	1	2	3	4	5
值	0	1	2	4	N	N

- 最後得到最終 LIS 長度=3

2. 以 C 實作演算法

按照以上基本觀念，可以寫出一演算法來實作 DP

```
A[0] = -1;
A[1] = MaxNum;           //初始化表格
for(i=0; i < Length; i++){ //開始讀取輸入串列
    while(A[L] > Data[i]){ //當輸入值比目前表格最大值還小
        L--;              //則往前一格尋找能比輸入值還小的值
    }
    A[L+1] = Data[i];      //改寫該表格數值的下一格為目前讀取值
    if(L+1 > max){         //如果改寫的表格長度大於目前最長長度，則最長長度增加
        max++;
        A[max+1] = MaxNum;
    }
}
```

3. 小技巧加速搜尋 — Binary Search

在以上方法當中，當輪到的數字為 n 時，常常要找出表格當中，第 i 個以

及第 $i+1$ 個數字，使得 $L[i] \leq n < L[i+1]$ ，若想加快此搜尋步驟，則可使用

二元搜尋法(Binary Search)來取得 $O(\log(m))$ $< m: \text{目前 LIS 長度} >$ 的搜尋時

間。

4. 以 C 實作演算法

我們可以寫出一完整演算法來實做 DP 與 binary search。

```
A[0] = -1;
A[1] = MaxNum;           //初始化表格
for(i=0; i < Length; i++){ //開始讀取輸入串列
    a = 0;
    b = max + 1;          //Initial Binary search的兩端點a.b
    while( !(A[L] <= Data[i] && Data[i] < A[L+1])){ //當輸入值比目前表格最大值還小
        L = (a+b)/2;           //時，使用Binary search來找到其
        if(A[L] <= Data[i])     //可取代的表格位置
            a = L+1;
        else
            b = L-1;
    }
    A[L+1] = Data[i];          //改寫該表格數值的下一格為目前讀取值
    if(L+1 > max){             //如果改寫的表格長度大於目前最長長度，則最長長度增加
        max += 1;
        A[max+1] = MaxNum;
    }
}
```

四、Performance Bottleneck and Assembly Solution

1. 減少Memory Access次數

```
while( !(A[L] <= Data[i] && Data[i] < A[L+1])){ }
```

以上程式碼若以GCC Compile成 Assembly則成為以下程式碼

```

movl    -12(%rbp), %eax

movl    -6032(%rbp, %rax, 4),%edx; 以上兩行將A[L] 值存至edx

movl    -20(%rbp), %eax

movl    -6080(%rbp, %rax, 4),%eax; 以上兩行將Data[i]值存至eax

cmpl    %eax, %edx

jg      .L5                      ; if (A[i] > data[i]) 跳出 while loop

movl    -20(%rbp), %eax

movl    -6080(%rbp,%rax,4), %edx; 以上兩行將Data[i]值存至edx

movl    -12(%rbp), %eax

addl    $1, %eax

movl    -6032(%rbp, %rax, 4),eax ;以上三行將A[L+1] 直存到eax

cmpl    %eax, %edx

jge     .L5                      ; if(data[i] >= A[L+1]) 跳出while loop

```

觀察後可發現此程式一共存取memory 4次但其中兩次為重複的data[i]

故重新改寫後則可使其存取memory 次數降為3次。再者，剩餘兩次的memory access為A[L], A[L+1]。利用mmx的64-bit register則可一次讀取這兩個data，又可將其存取memory次數下降，故以上code經過改寫後將原本4次存取降為兩次存取。

2. 減少branch次數

再度觀察由GCC compile出來的Assembly，其將原程式切割為以下區塊：

```
.L3 ; 原來的最外層 for loop 起始點

... ; a = 0 , b = max + 1 接著跳往L4

.L5

... ;Binary search找出新L值，跳回L4繼續測試

.L4

... ;while loop 條件測試，若條件成立跳入L5

... ; 若條件不成立代表Binary search成功

    ; 將值寫入Array內

.L2 ; 原來的最外層 for loop 條件後測點
```

觀察以上結構可發現，在L3讀取一新的data[i]之後，無論如何其必先跳到L4；

而L4條件若成立又要跳到L5開始進行Binary Search，在L5做完一次後，又

要跳回L4繼續條件測試。這樣做一次Binary Search需要3次conditional

jump，若能將此區塊改寫為以下情況：

```
.L3 ; 原來的最外層 for loop 起始點

... ; a = 0 , b = max + 1 直接接L4

.L4

... ;while loop 條件測試，若條件成立跳入L5

... ; 若條件不成立代表Binary search成功
```

```

; 將值寫入Array內

.L2 ; 原來的最外層 for loop 條件後測點

.L5

... ;Binary search找出新L值，跳回L4繼續測試

```

則做一次Binary search只需要2次 conditional jump，有效減少了branch 次數。

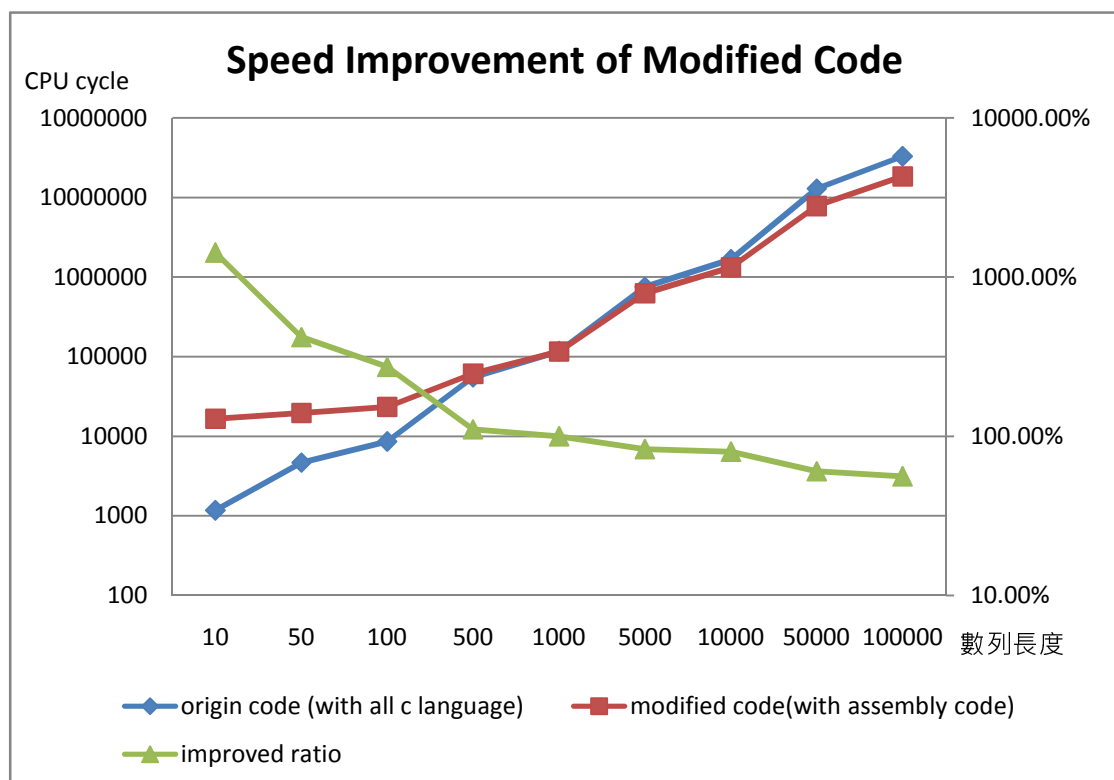
3. 使用較快的Instruction取代較慢者

> 以 shl reg, 2 取代 mul reg, 4

五、Performance Evaluation

利用Assembly 改良原本C 程式的效能之後，實際以測資來測試兩者效能

間的差異：



List length	Cycle time of origin C code	Cycle time of modified assembly code	Improved ratio
10	1167	16602	1422.62%
50	4678	19635	419.73%
100	8566	23350	272.59%
500	55116	60968	110.62%
1000	116566	116152	99.64%
5000	755385	624855	82.72%
10000	1664676	1327574	79.75%
50000	12960527	7819999	60.34%
100000	32918733	18377497	55.83%