資料結構與演算法 期末專題 技術報告

電機二 石旻翰 b08502141　　電機二 胡雅晴 b08901201

一、摘要

本專題有兩大主軸: 探討計算機之演算法及效率及作計算機介面。技術報告的第一部分將針對我們嘗試的三種演算法進行分析；接著實驗驗證各演算法的正確性並比較執行時間；最後則是與 GUI 進行結合，做出計算機操作介面。

二、計算機演算法與分析

計算機程式最困難處在於將一般表示法的字元重新排列，須易計算且符合先乘除後加減，括弧內先計算的條件。我們總共嘗試了以下三種演算法，並嚴謹的分析了各字的時間複雜度、空間複雜度，輔以實驗佐證，將在下面進行說明：

	① Parse Tree	② Recursion + Parse Tree	③ Shunting Yard
使用資料結構	Binary Tree	Binary Tree, Stack	Stack, Queue
空間複雜度	$O(n)$	$O(n)$	$O(n)$
時間複雜度	Worst Case $O(n^2)$ Best Case $O(nln(n))$	$O(n)$	$O(n)$
Github 檔名	parse_tree.py	dp.py	shunting_yard.py

定義與假設

1. 定義「計算機」輸入為由 n 個字元組成的字串「運算式」、輸出為計算結果

2. 定義二元樹的節點(下文以 node 代稱) 為含有 node.content、node.parent、node.left_child、node.right_child 的 python class，初始皆設為 None。

3. 下文以「運算子」代稱字元「+ - * / ^」;「數字」、「運算元」代稱由字元「0123456789.」組成的字串。

4. 部分特例與細節因篇幅有限簡化說明，實際情形請以程式為主。

(一) Parse Tree 解法

1. 演算法說明

要解決先乘除後加減、括弧內先運算的問題，我們的第一個想法是建立一個 Binary Parse Tree；將運算式中的字元依運算優先一一放到 node 中：越後進行運算的字元放的越靠近 root。完成 Parse Tree 後再以 left child 和 right child 為運算元，parent 為運算子進行計算，將結果放回 parent。如此反覆將 node 倆倆合併至 root，即可得出正確結果。

2. 時間複雜度

這個演算法包含三個主要步驟: 生成運算優先 list、生成二元樹、合併二元樹。其中一、三都是對每個 node 進行一次的處理，時間複雜度皆是 O(n)。而每次分出 left child, right child 時，需遍歷一次現有字元的評級找出最後計算者，時間複雜度大於 O(n)，因此總體的時間複雜度由這項流程主宰。

仔細觀察字串分裂的情形，當字元 w 由 parent 被分至 left child 或 right child 時，w 在二元樹中的階層，level(w) 增加一。因此 level(w) 即是 w 被下分的次數，而 level(w) + 1 即是 w 被比較評級的次數。令 t(n) 為長度 n 字串所花費的時間，α 為一個字元進行一次比較所花費的時間，那麼：

$$t(n) = \sum_{nodes \in binary\ tree} \alpha \times (level(nodes) + 1)$$

由於 α 是定值，level(nodes) 總和和 t(n) 正相關，因此 t(n) 在 level(nodes) 最大時有最大值、level(nodes)最小時有最小值。

(1) $\sum level(nodes)$ Maximum

node 離 root 越遠，level(node) 值越大；不難想像 t(n)最大時，binary tree 是盡可能向下生長的。由於一運算子需與兩運算元配對，故此樹為 full binary tree。這樣的樹樹高 $H = (n-1)/2$；並且除了 $h = 0$ 的 node 有 $(n+1)/2$ 個外，其餘 $h \leq (n-1)/2$ 的 node 各只有一個 (見圖 1)。沿用先前變數，得：

$$t(n) = 2\alpha \sum_{k=1}^{\frac{n-1}{2}}(k+1) + \alpha = 2\alpha \times (\tfrac{n-1}{2}) + 2\alpha \times \frac{(\frac{n-1}{2}+1)+\frac{n-1}{2}}{2} + \alpha = \alpha n + \alpha\frac{n^2-1}{4}$$

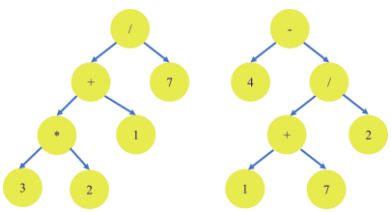
(2) $\sum level(nodes)$ Minimum

level(node)總和在樹為 perfect binary tree 時有最小值。此時， $H = ln(n+1) - 1$；而高度 h 的 node 共有 $(n+1)/(2^{h+1})$ 個 (見圖 2)，沿用(1)之變數，則:

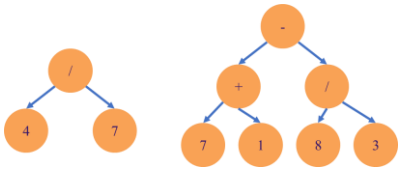
$$t(n) = 1 \times 2^0 + 2 \times 2^1 + \ldots + (ln(n+1) - 1) \times 2^{ln(n+1)-2}$$

且 $t(n) = t(\frac{n}{2}) + (ln(n+1) - 1) \times 2^{ln(n+1)-2} = t(\frac{n}{2}) + \frac{1}{4}(ln(n+1) - 1)(n+1) = t(\frac{n}{2}) + O(n(ln(n)))$

因此，根據 Master Theorm, $t(n)_{min} \in O(nln(n))$



(圖 1) t(n) 最大樹範例



(圖 2) t(n) 最小樹範例

綜(1)、(2) 得出的結論，此演算法的時間複雜度在 worst case 是 $O(n^2)$，在 best case 時為O(n ln(n))。

3. 空間複雜度

此演算法中用到的空間有記錄運算評級的 list 以及 parse tree 的 node，兩者與 n 個字元都是一一對應的關係，因此空間複雜度為 O(n)。

4. 討論

在計算時間複雜度的過程中，我們發現時間複雜度是由「檢索運算評級最低」流程主宰。若要加快速度，便需要尋找排列字元更高效率的做法。

(二) Parse Tree + Recursion 解法

1. 演算法說明

在上一個演算法中，必須檢索運算評級最低的原因在於不同層次括號對計算優先順序的影響；若沒有括號的話，僅需觀察字元本身就能判定 parse tree 新 node 的生成位置。有鑑於此，在此演算法中，我們將嘗試以遞迴函式去除括弧，將 parse tree 分段建立、合併。此演算法能大致以兩函式說明：

(1) def unPar(s):
輸入：運算式 s (可含有括弧)
輸出：運算式 s 的計算結果

while(s 含有括弧於位置 x, y)：
    s = [: x] + unPar(s[x + 1: y]) + s[y+1:] #將括弧、子運算式以結果取代
return findVal(s)

(2) def findVal(s):
輸入：運算式 s (不可含有括弧)
輸出：運算式 s 的計算結果
利用與演算法(一) 相同的作法，以 parse tree 排列字元、計算結果。但因每有括弧問題，可直接從字元中判斷二元樹新產生 node 的位置。

2. 時間複雜度

由於同樣是建立 n-node parse tree，故演算法之時間複雜度為 $O(n)$。

3. 空間複雜度

使用到的空間為 parse tree，空間複雜度 $O(n)$。

4. 討論

我們成功的將時間複雜度降低至$O(n)$，理論上較演算法(一) 更有效率。然而進行實驗發現時間非旦沒有縮短，還增加。我們推測這是因為此演算法中使用了大量的遞迴，因此拖慢了執行時間。

測資數量	① parse_tree.py	② dp.py
500	0.112505	0.141570
1000	0.212519	0.303690
1500	0.394034	0.490472
2000	0.574941	0.959134
2500	0.736105	1.203293

(三) 排成場演算法 (資料結構: stack、queue；見 shunting_yard.py)

0. 背景：逆波蘭表示法 Reversed Polish Notation (RPN)

RPN 表示法不同於一般常用寫法，是將運算子置於運算元後面的。例如常用表示法中，「一加三」寫作「1+3」，而 RPN 表示法則寫作「1 3 +」。

RPN 表示法與演算法(一)、(二)所使用的 parse tree 間存在有趣的關聯。若將一運算式使用 parse tree 表示，則 RPN 表示法恰會是此 parse tree 的 post-order traversal 結果。

1. 演算法說明

Shunting Yard 演算法以兩 list: RPN(queue)、OP(stack) 將運算式轉作逆波蘭表示法 (運算子置於運算元後面)，規則如下：

(1) 字元為數字：
將字元 push 到 RPN 中

(2) 字元為運算子：
a. 若 OP 為空 **or** 字元運算層級高於 OP 頂端元素：
將字元 push 到 OP 中
b. 若字元運算層級低於 OP 頂端元素：
令 temp 為 OP 頂端元素
將 temp push 到 RPN 中
pop OP
重新作(2)的判斷

(3) 字元為右括弧 ‘(‘：
將字元 push 到 OP 中

(4) 字元為左括弧 ‘)’：

重複以下至頂端元素為’(‘：

令 temp 為 OP 頂端元素

將 temp push 到 RPN 中

pop OP

pop OP

最後，將 OP push 至 RPN 中，即可得到逆波蘭表示法的運算式。計算結果時從對 RPN 進行 traverse，當碰到運算子字元時，將字元對前二元素進行運算、取代，最後便會得出運算結果。

3. 時間複雜度

將運算式轉作逆波蘭表示法時，每個字元至多 OP push、OP pop、RPN push 一次；而計算結果時，各字元皆被 traverse 一次，因此時間複雜度為 $O(n)$。

4. 空間複雜度

在此演算法中 RPN、OP 的長度和至多為 n，所以空間複雜度為 $O(n)$。

5. 討論

從實驗結果來看，Shunting Yard 演算法確實的提升了程式執行的速度。我們將 500、1000、1500、2000、2500 筆測資(一)、(三)演算法的執行時間比較，發現(三)最多只有(一)執行時間的 0.5，有顯著的改善。

測資數量	① parse_tree.py	③ shunting_yard.py	①/③
500	0.112505	0.049829	2.2578218
1000	0.212519	0.102583	2.0716785
1500	0.394034	0.171645	2.2956334
2000	0.574941	0.269066	2.1368029
2500	0.736105	0.333777	2.2053796

而在空間上，雖然演算法(一)、(二)、(三)的空間複雜度皆是 O(n)，但 Shunting Yard 使用的是 python 內建的 list，相較自行編纂、每個 node 有 node.content、node.parent、node.right_child、node.left_child 的 parse tree 所佔有的空間較小。

綜合以上兩個原因，在製作演算法介面時，我們便選用該演算法作計算機。

三、設計實驗的內容、實驗正確性、執行時間表格

1.實驗內容：

a. 正確性：在 correctness 資料夾中的測資為簡單四則運算以及我們各個函數的簡單對應的值，其正確性參考於 wolfram alpha 之計算結果(取到小數點後三位)，但因為有計算精度誤差，故使用 shunting_yard.py 的答案做為 correct_ans 之中的 golden.txt 中，後綴數字分別為對應的測資答案，資料夾有分為 windows 系統(correct_ans_win)以及 macOS/Linux 系統(correct_ans_macOS)。

欲一次測試全部的測資使用 bash evaluation_作業系統.sh 測試：windows 系統使用 evaluation_win.sh；macOS 以及 Linux 則使用 evaluation_macOS.sh。此程序會一一比對生成的 output.txt 內容與對應資料夾內的 golden.txt 是否相同。

詳細實驗記錄於下方 Correctness 表格中。

b. 執行時間：在 time_data 資料夾中的測資為使用 exp_generator2.py 產生之隨機 list，並將生成之 list 貼到 exp_generator.py 中的 self.num 以去隨機生成算式，產生之算式紀錄於 input.txt 中，各測資產生長度分別為 500、1000、1500、2000、2500，使用 time 套件紀錄下執行各程式做運算的迴圈時間，並以此當作各測資的執行時間參考。

欲一次測試全部的測資使用 bash time_evaluation_作業系統.sh 測試：windows 系統使用 time_evaluation_win.sh；macOS 以及 Linux 則使用 time_evaluation_macOS.sh。此程序會一一比對生成的 output.txt 內容與對應資料夾內的 golden.txt 是否相同。

詳細實驗記錄於下方 Time Measurement 表格中。

2. Correctness: (三個作業系統執行結果均相同，三種計算機程式的答案均相同)

	Length	Parsetree.py	dp.py	Shuntingyard.py
correct_1.txt	15	correct	correct	correct
correct_2.txt	15	correct	correct	correct
correct_3.txt	15	correct	correct	correct
correct_4.txt	15	correct	correct	correct
correct_5.txt	15	correct	correct	correct

3. Time Measurement: (Ubuntu 20.04 為最佳紀錄，此表格以此做為紀錄)

	Length	Parsetree.py	dp.py	Shuntingyard.py
input_1.txt	500	0.1125s	0.1416s	0.0498s
input_2.txt	1000	0.2125s	0.3037s	0.1026s
input_3.txt	1500	0.3940s	0.4905s	0.1716s
input_4.txt	2000	0.5750s	0.9591s	0.2690s
input_5.txt	2500	0.7361s	1.2033s	0.3338s

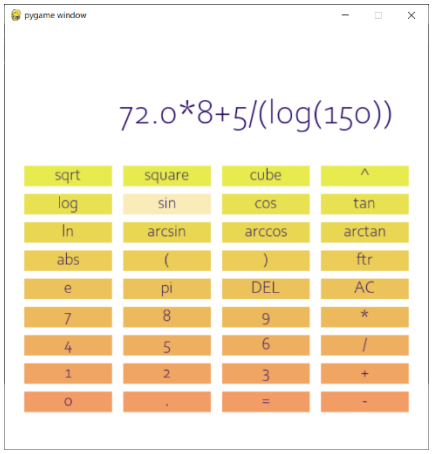
四、GUI

製作計算機介面時我們嘗試了兩個套件：tkinter 及 pygame。tkinter 的優點在有內建的 button class，且可以與函式直接連結，編寫程式較 pygame 方便。然而 tkinter 寫出的介面較為僵硬不人性，因此最後選用了 pygame 作套件。

相較會自行判斷度數、弧度造成誤會的網路計算機，我們將角度的表示直接定為度數。精準度也固定作小數點後三位，不會隨結果的大小而浮動。

介面設計上則是以方便使用為主。較手機的計算機按鍵多，不須切換鍵盤；較實體的工程計算機按鈕更寬大，方便點擊。而儘管計算機程式中的各函式皆以一字母表示(如：sin(45) 寫作 c(45))，在介面中也重新翻譯，能輕鬆使用。

執行指令：python ./GUI.py
執行畫面：



⚠ 注意！此計算機有脾氣，他不能接受不想自己手算還不知道正確輸入算式的人，假設你做出了讓他討厭的事情，他會閃退不理你！！！

五、結論

以正確性的實驗結果來做個簡單的結論：三種不同的程式在輸入相同的算式中均可得出相同的答案，這點可以佐證我們做出來的程式執行正確性相當精準。程式執行時間方面，我們分別測試了三個作業系統下的情況，發現 linux 的執行時間相較於其他兩者明顯快上不少，故我們使用 linux 系統中的執行時間來做個簡單的結論：shunting yard 優於 parse tree 優於 dp，dp 在測試資料較多的測資中，執行時間的表現甚至是暴增的狀況，主要原因應該是出自於 recursive 的部分。另外，shunting yard 的執行時間表現幾乎趨近於理論複雜度 O(n)，對此我們感到很開心能做出一個既可以正確執行的程式且執行快速的計算機。

六、參考資料

1. 7.6. Parse Tree — Problem Solving with Algorithms and Data Structures
   Runestone.academy
   https://runestone.academy/runestone/books/published/pythonds/Trees/ParseTree
   Pars
2. Shunting-yard algorithm – Wikipedia
   En.wikipedia.org
   https://en.wikipedia.org/wiki/Shunting-yard_algorithm
3. 資料結構各個程式作業中的 evaluaiton.sh 內容

七、分工

工作	負責人	檔案
題目發想與進度規劃	石旻翰、胡雅晴	
計算機程式撰寫	胡雅晴	shunting_yard.py, dp.py, parse_tree.py
測資生成	石旻翰	exp_generatpr.py
正確性驗證	石旻翰	evaluation.sh
速度驗證	石旻翰	time_evaluation.sh
使用者介面製做	胡雅晴	GUI.py