Data Structures and Algorithms: Homework #3

Due on April 28, 2015 at 16:20

Instructors Hsuan-Tien Lin, Roger Jang

Tim Liou (b03902028)

3.1 Asymptotic Complexity

(1) Do Exercise R-4.28 of the textbook.

Consider $a_k \geq 0 \ for \ k \geq 0$

$$p(n) = a_0 + a_1 n + a_2 n^2 + a_3 n^3 + \cdots + a_m n^m$$

For $n \geq (a_0 + a_1 + a_2 + \cdots + a_m) \geq 1$, we have

$$\begin{aligned} p(n) &\leq (a_0 + a_1 + a_2 + \cdots + a_m) \times n^m \\ \Rightarrow \log p(n) &\leq \log(a_0 + a_1 + a_2 + \cdots + a_m) + m \log n \\ &\leq \log n + m \log n \\ &= (m+1) \log n \end{aligned}$$

Take $c = m + 1 > 0$, $n_0 = (a_0 + a_1 + a_2 + \cdots + a_m) \geq 1$

$$\log p(n) \leq c \log n \quad for \ n \geq n_0$$

That is, $\log p(n)$ is $O(\log n)$.

(2) Do Exercise R-4.34 of the textbook.

We have $f(n) > 1$ and $\lceil f(n) \rceil \leq f(n) + 1$ by definition. For $n \geq 1$,

$$\begin{aligned} \lceil f(n) \rceil &\leq f(n) + 1 \\ &\leq f(n) + f(n) \\ &= 2f(n) \end{aligned}$$

Take $c = 2 > 0$, $n_0 = 1 \geq 1$

$$\lceil f(n) \rceil \leq cf(n) \quad for \ n \geq n_0$$

That is, $\lceil f(n) \rceil$ is $O(f(n))$.

(3) Prove that $f(n) = \Theta(g(n))$.

By definition of limits at infinity,

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = A$$

means that for every $\epsilon > 0$ there is a corresponding N such that

$$\left| \frac{f(n)}{g(n)} - A \right| < \epsilon \quad for \ n > N$$

That is,

$$A - \epsilon < \frac{f(n)}{g(n)} < A + \epsilon \quad for\ n > N$$

Note that $g(n)$ is a strictly positive function. We have

$$(A - \epsilon)g(n) < f(n) < (A + \epsilon)g(n) \quad for\ n > N$$

Take $\epsilon \in (0, A)$, $c_1 = (A - \epsilon) > 0$, $c_2 = (A + \epsilon) > 0$, $n_0 > N$

$$c_1 g(n) \leq f(n) \leq c_2 g(n) \quad for\ n > n_0$$

This shows that $f(n) = \Theta(g(n))$.

(4) Do Exercise R-4.8 of the textbook.

If A is better than B for $n \geq n_0$, n_0 satisfies the following statement.

$$2{n_0}^3 - 40{n_0}^2 > 0$$

We can easily find that $n_0 > 20$. We choose $n_0 = 21$. It is a possible value for n_0 satisfying the statement that A is better than B for $n \geq n_0$.

(5) Do Exercise C-4.16(b) of the textbook.

This is the pseudo code of the Horner's method.

```
1: function HORNER'S-METHOD(x, CoefficientsOfPolynomial, DegreeOfPolynomial)
2:     Sum ← 0
3:     for all CoefficientsOfPolynomial do
4:         Sum ← Sum × x + CoefficientsOfPolynomial
5:     end for
6:     return Sum
7: end function
```

Algorithm 1: Horner's method for computing polynomial

We can find there is only one **for loop** in this pseudo code, that is, the number of arithmetic operations is $O(n)$.

(6) Consider some $f(n)$ and $g(n)$ such that $\lg f(n) = O(\lg g(n))$ and $g(n) \geq 2$ for $n \geq 1$. Construct a counter-example to disprove that $f(n) = O(g(n))$.

Consider $f(n) = 4^n$, $g(n) = 2^n$, for $n \geq 1$, we can find

$$\begin{aligned}\lg f(n) &= n2 \lg 2 \\ &\leq n4 \lg 2 \\ &= 4 \lg 2^n \\ &= 4 \lg g(n)\end{aligned}$$

Take $c = 4 > 0$, $n_0 = 1 \geq 1$

$$\lg f(n) \leq c \lg g(n) \quad for\ n \geq n_0$$

That is, $\lg f(n)$ is $O(\lg g(n))$. Note that $g(n) \geq 2$ for $n \geq 1$.

If $f(n) = O(g(n))$, $\exists n_0 > 0, c > 0 \ni$

$$4^n \leq c2^n \quad for\ n \geq n_0$$

Take $\log_2$ on both sides,

$$\begin{aligned}&2n \leq \log_2 c + n \quad for\ n \geq n_0 \\ &\Rightarrow \log_2 c \geq n\end{aligned}$$

That is, take $n' = max(n_0, \lceil \log_2 c + 1 \rceil)$

$$\begin{aligned}n' \geq n_0 &\Rightarrow 4^n \leq c2^n \\ n' > \log_2 c &\Rightarrow 4^n > c2^n\end{aligned}$$

This is a contradiction. Therefore, we disprove that $f(n) = O(g(n))$.

3.2 Stack, Queue, Deque

(1) Do Exercise C-5.2 of the textbook.

Pop out the elements in the stack one by one and check if it is equal to element x. After that, enqueue the elements into the queue one by one. Use a varible to store the number of the elements we poped from the stack. Once we find the certain element or the stack is empty, we push the elements into stack from queue, and then enqueue the same number of element into queue from stack. Finally push all these elements from queue into stack again. This will maintain elements' original order.

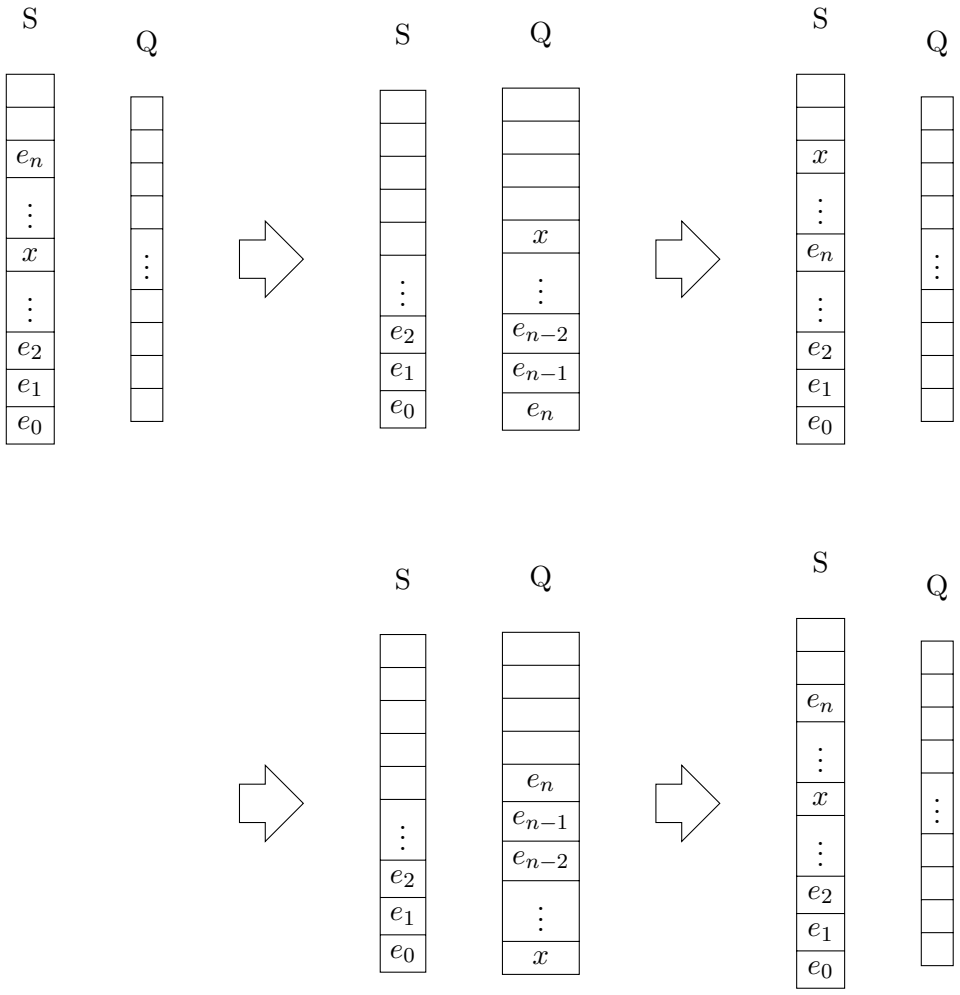

Figure 1: How this algorithm works

(2) Do Exercise C-5.9 of the textbook.

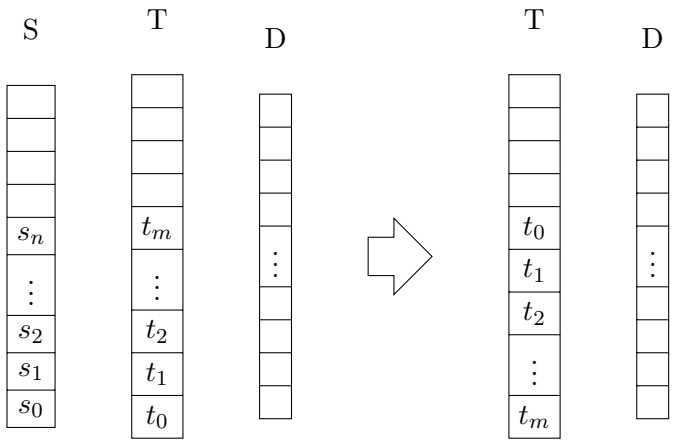

Figure 2: **Pop** all elements from T and **Push_front** them to D. Then **Pop_back** them back to T.

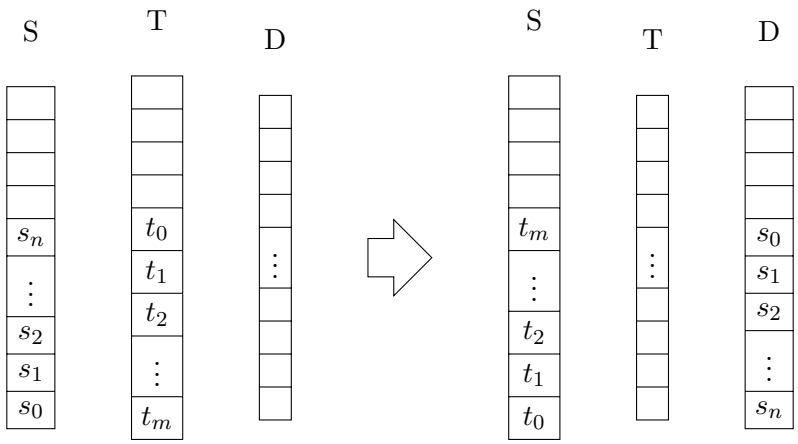

Figure 3: **Pop** all elements from S and **Push_front** them to D. Then **Pop** all elements from T to S.

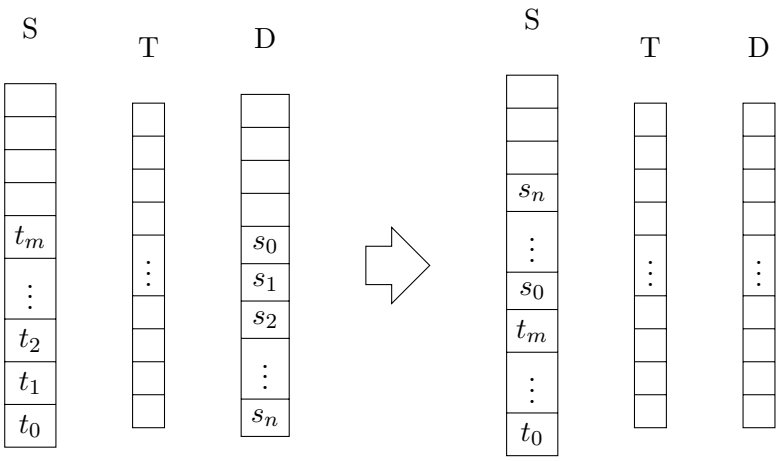

Figure 4: **Pop_front** all elements from D to S

(3) Use any pseudocode to write down an algorithm that uses two stacks (with push, pop and isempty operations but no others) to simulate one deque (for push/pop front and push/pop back operations). What is the total running time after N operations?

Imagine we divide a deque into two stacks named S_f and S_b. **PushFront**, **PopFront** are processed in S_f while **PushBack**, **PopBack** are processed in S_b. However, we need to transport elements from a stack to the other one if we want to **Pop** elements from an empty stack. Note that both stacks are empty means the deque is empty.

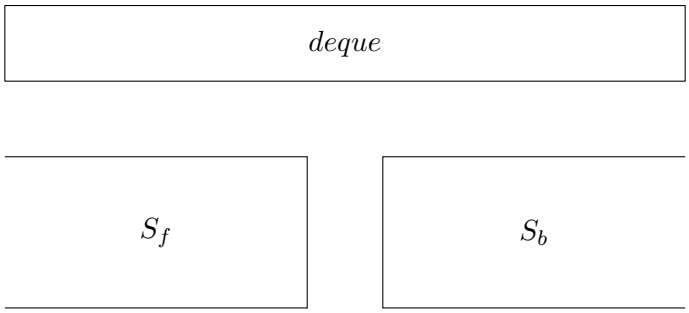

Figure 5: Use two stacks to simulate a deque

```
1: function PopFront
2:     if S_f and S_b aren't both empty then
3:         if S_f is empty then
4:             pop all elements from S_b to S_f
5:         end if
6:         pop from S_f
7:     end if
8: end function
```

Algorithm 2: PopFront of deque using two stacks

```
1: function PopBack
2:     if S_f and S_b aren't both empty then
3:         if S_b is empty then
4:             pop all elements from S_f to S_b
5:         end if
6:         pop from S_b
7:     end if
8: end function
```

Algorithm 3: PopBack of deque using two stacks

```
1: function PushFront(e)
2:     push e into S_f
3: end function
```

Algorithm 4: PushFront

```
1: function PushBack(e)
2:     push e into S_b
3: end function
```

Algorithm 5: PushBack

Suppose **Pop**/**Push** both take t (Time Unit). There are some cases result in different running time.

Case 1: all the operations are either **PushBack** or **PushFront**

Since these two operations are just a **Push** operation for a stack, the time complexity is constant. After N operation, the total running time is simply Nt.

Case 2: operations contain **PopBack** or **PopFront**, but never make any stack empty

Like Case 1, all the opertations are just **Push** or **Pop** for a stack. Therefore, the total running time is also Nt.

Case 3: operations contain **PopBack** or **PopFront**, and try to **Pop** from an empty stack

In this stituation, it would **Pop** all elements from the other stack to its first then **Pop** the desired elements. Suppose this stituation happened k times, and there are a_i elements in the other stack at the ith time. The total time is $N + \sum_{i=1}^{k} a_i$.

(4) Do Exercise C-5.9 of the textbook, but with three stacks instead of two stacks and one deque.

Suppose three stacks are big enough for all elements.

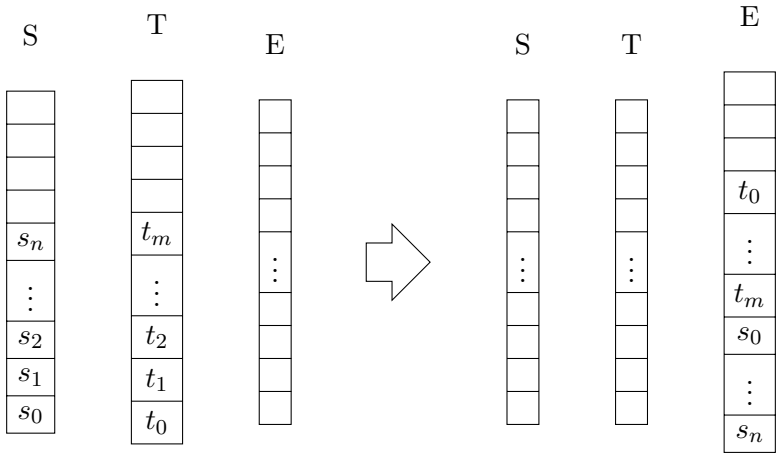

Figure 6: **Pop** all elements from S to E and then **Pop** all elements from T to E.

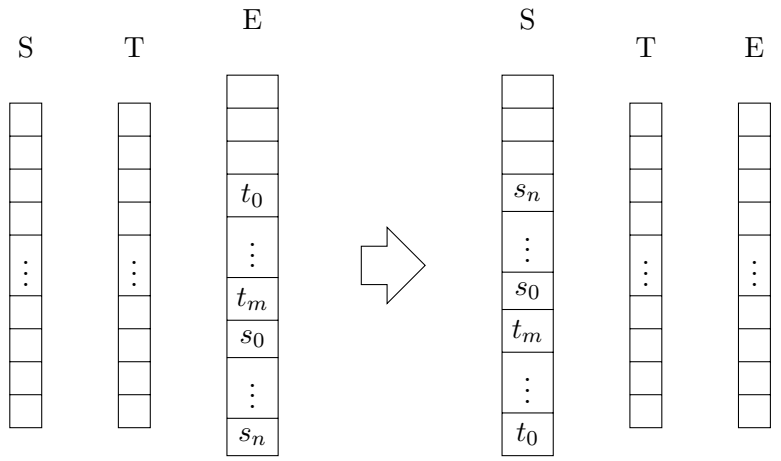

Figure 7: **Pop** all elements from E to S

3.3 List, Iterator

(1) Do Exercise C-6.7 of the textbook.

Like textbook, we view the computer as a coin-operated appliance, which requires the payment of one **cyber-dollar** for a constant amount of computing time. Suppose it costs 6 cyber-dollars to push one element to a non-full array. In fact, only one of them is used to insert a element, and the other five cyber-dollars just store in the place. Consider a case that an array just extended its size from N to $N + \lceil \frac{N}{4} \rceil$. After pushing $\lceil \frac{N}{4} \rceil$ elements, the array is full again. Note that we stored at least $5 \times \lceil \frac{N}{4} \rceil$ cyber-dollars at this moment. Next time before we push a new element, we would have to copy $N + \lceil \frac{N}{4} \rceil$ elements from old array to a new array. And we know that

$$\begin{aligned} 5 \times \lceil \frac{N}{4} \rceil &= 4 \times \lceil \frac{N}{4} \rceil + \lceil \frac{N}{4} \rceil \\ &\geq N + \lceil \frac{N}{4} \rceil \end{aligned}$$

It means we can use the cyber-dollars we stored before to copy elements and without running out of the money. That is, the real average cost of pushing a element to an array is less than 6 cyber-dollars. Therefore, it totally costs less than $6n$ cyber-dollars after a sequence of n push operations. This implies it still run in $O(n)$ in this case.

(2) Do Exercise C-6.13 of the textbook by Googling the *Knuth Shuffle*.

```
1: function KNUTH-SHUFFLE(V, LengthOfV)
2:     for i ← 0 to LengthOfV − 1 do
3:         r ← RANDOMINTGER(i + 1)
4:         Exchange V[i] and V[r]
5:     end for
6: end function
```

Algorithm 6: Knuth-Shuffle

Knuth Shuffle guarantees that every possible ordering is equally likely. The running time of this function is $O(n)$, n is the number of cards.

3.4 Calculators

(1) Three cases for testing integer calculator

```
Input:

123+2*3*(5-3+48/2) + 34
-~((1024 >> (23 % 3)) + !0)
((3 && +1 || 1) ^ 12 << 1 ) & 16 | 2
---
Output:

--- postfix expression transforming ---
encounter 123: push to output
       current output: 123
encounter +: push to the stack directly
       current output: 123
       current stack: +
encounter 2: push to output
       current output: 123 2
       current stack: +
encounter *: push to the stack directly
       current output: 123 2
       current stack: + *
encounter 3: push to output
       current output: 123 2 3
       current stack: + *
encounter *: stack.top() has greater or the same precdence, after pop
   something out to output, then push to the stack
       current output: 123 2 3 *
       current stack: + *
encounter (: push to the stack directly
       current output: 123 2 3 *
       current stack: + * (
encounter 5: push to output
       current output: 123 2 3 * 5
       current stack: + * (
encounter -: push to the stack directly
       current output: 123 2 3 * 5
       current stack: + * ( -
encounter 3: push to output
       current output: 123 2 3 * 5 3
       current stack: + * ( -
encounter +: stack.top() has greater or the same precdence, after pop
```

```
    something out to output, then push to the stack
        current output: 123 2 3 * 5 3 -
        current stack: + * ( +
encounter 48: push to output
        current output: 123 2 3 * 5 3 - 48
        current stack: + * ( +
encounter /: push to the stack directly
        current output: 123 2 3 * 5 3 - 48
        current stack: + * ( + /
encounter 2: push to output
        current output: 123 2 3 * 5 3 - 48 2
        current stack: + * ( + /
encounter ): flush the stack to output until meeting '('
        current output: 123 2 3 * 5 3 - 48 2 / +
        current stack: + *
encounter +: stack.top() has greater or the same precdence, after pop
    something out to output, then push to the stack
        current output: 123 2 3 * 5 3 - 48 2 / + * +
        current stack: +
encounter 34: push to output
        current output: 123 2 3 * 5 3 - 48 2 / + * + 34
        current stack: +
encounter NOTHING: flush the stack to output
        current output: 123 2 3 * 5 3 - 48 2 / + * + 34 +
--- postfix expression transforming complete :) ---
Postfix Exp: 123 2 3 * 5 3 - 48 2 / + * + 34 +
RESULT: 313
--- postfix expression transforming ---
encounter U-: push to the stack directly
        current output:
        current stack: -
encounter ~: push to the stack directly
        current output:
        current stack: - ~
encounter (: push to the stack directly
        current output:
        current stack: - ~ (
encounter (: push to the stack directly
        current output:
        current stack: - ~ ( (
encounter 1024: push to output
        current output: 1024
        current stack: - ~ ( (
```

```
encounter >>: push to the stack directly
       current output: 1024
       current stack: - ~ ( ( >>
encounter (: push to the stack directly
       current output: 1024
       current stack: - ~ ( ( >> (
encounter 23: push to output
       current output: 1024 23
       current stack: - ~ ( ( >> (
encounter %: push to the stack directly
       current output: 1024 23
       current stack: - ~ ( ( >> ( %
encounter 3: push to output
       current output: 1024 23 3
       current stack: - ~ ( ( >> ( %
encounter ): flush the stack to output until meeting '('
       current output: 1024 23 3 %
       current stack: - ~ ( ( >>
encounter ): flush the stack to output until meeting '('
       current output: 1024 23 3 % >>
       current stack: - ~ (
encounter +: push to the stack directly
       current output: 1024 23 3 % >>
       current stack: - ~ ( +
encounter !: push to the stack directly
       current output: 1024 23 3 % >>
       current stack: - ~ ( + !
encounter 0: push to output
       current output: 1024 23 3 % >> 0
       current stack: - ~ ( + !
encounter ): flush the stack to output until meeting '('
       current output: 1024 23 3 % >> 0 ! +
       current stack: - ~
encounter NOTHING: flush the stack to output
       current output: 1024 23 3 % >> 0 ! + ~ -
--- postfix expression transforming complete :) ---
Postfix Exp: 1024 23 3 % >> 0 ! + ~ -
RESULT: 258
--- postfix expression transforming ---
encounter (: push to the stack directly
       current output:
       current stack: (
encounter (: push to the stack directly
```

```
124         current output:
125         current stack: ( (
126   encounter 3: push to output
127         current output: 3
128         current stack: ( (
129   encounter &&: push to the stack directly
130         current output: 3
131         current stack: ( ( &&
132   encounter U+: push to the stack directly
133         current output: 3
134         current stack: ( ( && +
135   encounter 1: push to output
136         current output: 3 1
137         current stack: ( ( && +
138   encounter ||: stack.top() has greater or the same precdence, after pop
          something out to output, then push to the stack
139         current output: 3 1 + &&
140         current stack: ( ( ||
141   encounter 1: push to output
142         current output: 3 1 + && 1
143         current stack: ( ( ||
144   encounter ): flush the stack to output until meeting '('
145         current output: 3 1 + && 1 ||
146         current stack: (
147   encounter ^: push to the stack directly
148         current output: 3 1 + && 1 ||
149         current stack: ( ^
150   encounter 12: push to output
151         current output: 3 1 + && 1 || 12
152         current stack: ( ^
153   encounter <<: push to the stack directly
154         current output: 3 1 + && 1 || 12
155         current stack: ( ^ <<
156   encounter 1: push to output
157         current output: 3 1 + && 1 || 12 1
158         current stack: ( ^ <<
159   encounter ): flush the stack to output until meeting '('
160         current output: 3 1 + && 1 || 12 1 << ^
161   encounter &: push to the stack directly
162         current output: 3 1 + && 1 || 12 1 << ^
163         current stack: &
164   encounter 16: push to output
165         current output: 3 1 + && 1 || 12 1 << ^ 16
```

```
166         current stack: &
167     encounter |: stack.top() has greater or the same precdence, after pop
            something out to output, then push to the stack
168         current output: 3 1 + && 1 || 12 1 << ^ 16 &
169         current stack: |
170     encounter 2: push to output
171         current output: 3 1 + && 1 || 12 1 << ^ 16 & 2
172         current stack: |
173     encounter NOTHING: flush the stack to output
174         current output: 3 1 + && 1 || 12 1 << ^ 16 & 2 |
175     --- postfix expression transforming complete :) ---
176     Postfix Exp: 3 1 + && 1 || 12 1 << ^ 16 & 2 |
177     RESULT: 18
```

(2) Three cases for testing scientific calculator

```
1     Input:
2
3     - pow( (2.3 + 3) *2 , exp( log(2) ) )
4     sqrt(1/16) + fabs(sin(3 / 2 * 3.1415926)) + +cos(3.1415926)
5     0.00 + 1.2
6     ---
7     Output:
8
9     --- postfix expression transforming ---
10    encounter U-: push to the stack directly
11          current output:
12          current stack: -
13    encounter pow: push to the stack directly
14          current output:
15          current stack: - pow
16    encounter (: push to the stack directly
17          current output:
18          current stack: - pow (
19    encounter (: push to the stack directly
20          current output:
21          current stack: - pow ( (
22    encounter 2.3: push to output
23          current output: 2.300000
24          current stack: - pow ( (
25    encounter +: push to the stack directly
26          current output: 2.300000
27          current stack: - pow ( ( +
28    encounter 3: push to output
```

```
        current output: 2.300000 3.000000
        current stack: - pow ( ( +
encounter ): flush the stack to output until meeting '('
        current output: 2.300000 3.000000 +
        current stack: - pow (
encounter *: push to the stack directly
        current output: 2.300000 3.000000 +
        current stack: - pow ( *
encounter 2: push to output
        current output: 2.300000 3.000000 + 2.000000
        current stack: - pow ( *
encounter exp: push to the stack directly
        current output: 2.300000 3.000000 + 2.000000
        current stack: - pow ( * exp
encounter (: push to the stack directly
        current output: 2.300000 3.000000 + 2.000000
        current stack: - pow ( * exp (
encounter log: push to the stack directly
        current output: 2.300000 3.000000 + 2.000000
        current stack: - pow ( * exp ( log
encounter (: push to the stack directly
        current output: 2.300000 3.000000 + 2.000000
        current stack: - pow ( * exp ( log (
encounter 2: push to output
        current output: 2.300000 3.000000 + 2.000000 2.000000
        current stack: - pow ( * exp ( log (
encounter ): flush the stack to output until meeting '(' and pop function 'log
    ' to output
        current output: 2.300000 3.000000 + 2.000000 2.000000 log
        current stack: - pow ( * exp (
encounter ): flush the stack to output until meeting '(' and pop function 'exp
    ' to output
        current output: 2.300000 3.000000 + 2.000000 2.000000 log exp
        current stack: - pow ( *
encounter ): flush the stack to output until meeting '(' and pop function 'pow
    ' to output
        current output: 2.300000 3.000000 + 2.000000 2.000000 log exp * pow
        current stack: -
encounter NOTHING: flush the stack to output
        current output: 2.300000 3.000000 + 2.000000 2.000000 log exp * pow -
--- postfix expression transforming complete :) ---
Postfix Exp: 2.300000 3.000000 + 2.000000 2.000000 log exp * pow -
RESULT: -789.048100
```

```
69    --- postfix expression transforming ---
70    encounter sqrt: push to the stack directly
71           current output:
72           current stack: sqrt
73    encounter (: push to the stack directly
74           current output:
75           current stack: sqrt (
76    encounter 1: push to output
77           current output: 1.000000
78           current stack: sqrt (
79    encounter /: push to the stack directly
80           current output: 1.000000
81           current stack: sqrt ( /
82    encounter 16: push to output
83           current output: 1.000000 16.000000
84           current stack: sqrt ( /
85    encounter ): flush the stack to output until meeting '(' and pop function '
          sqrt' to output
86           current output: 1.000000 16.000000 / sqrt
87    encounter +: push to the stack directly
88           current output: 1.000000 16.000000 / sqrt
89           current stack: +
90    encounter fabs: push to the stack directly
91           current output: 1.000000 16.000000 / sqrt
92           current stack: + fabs
93    encounter (: push to the stack directly
94           current output: 1.000000 16.000000 / sqrt
95           current stack: + fabs (
96    encounter sin: push to the stack directly
97           current output: 1.000000 16.000000 / sqrt
98           current stack: + fabs ( sin
99    encounter (: push to the stack directly
100          current output: 1.000000 16.000000 / sqrt
101          current stack: + fabs ( sin (
102   encounter 3: push to output
103          current output: 1.000000 16.000000 / sqrt 3.000000
104          current stack: + fabs ( sin (
105   encounter /: push to the stack directly
106          current output: 1.000000 16.000000 / sqrt 3.000000
107          current stack: + fabs ( sin ( /
108   encounter 2: push to output
109          current output: 1.000000 16.000000 / sqrt 3.000000 2.000000
110          current stack: + fabs ( sin ( /
```

```
111     encounter *: stack.top() has greater or the same precdence, after pop
            something out to output, then push to the stack
112             current output: 1.000000 16.000000 / sqrt 3.000000 2.000000 /
113             current stack: + fabs ( sin ( *
114     encounter 3.1415926: push to output
115             current output: 1.000000 16.000000 / sqrt 3.000000 2.000000 / 3.141593
116             current stack: + fabs ( sin ( *
117     encounter ): flush the stack to output until meeting '(' and pop function 'sin
            ' to output
118             current output: 1.000000 16.000000 / sqrt 3.000000 2.000000 / 3.141593
                    * sin
119             current stack: + fabs (
120     encounter ): flush the stack to output until meeting '(' and pop function '
            fabs' to output
121             current output: 1.000000 16.000000 / sqrt 3.000000 2.000000 / 3.141593
                    * sin fabs
122             current stack: +
123     encounter +: stack.top() has greater or the same precdence, after pop
            something out to output, then push to the stack
124             current output: 1.000000 16.000000 / sqrt 3.000000 2.000000 / 3.141593
                    * sin fabs +
125             current stack: +
126     encounter U+: push to the stack directly
127             current output: 1.000000 16.000000 / sqrt 3.000000 2.000000 / 3.141593
                    * sin fabs +
128             current stack: + +
129     encounter cos: push to the stack directly
130             current output: 1.000000 16.000000 / sqrt 3.000000 2.000000 / 3.141593
                    * sin fabs +
131             current stack: + + cos
132     encounter (: push to the stack directly
133             current output: 1.000000 16.000000 / sqrt 3.000000 2.000000 / 3.141593
                    * sin fabs +
134             current stack: + + cos (
135     encounter 3.1415926: push to output
136             current output: 1.000000 16.000000 / sqrt 3.000000 2.000000 / 3.141593
                    * sin fabs + 3.141593
137             current stack: + + cos (
138     encounter ): flush the stack to output until meeting '(' and pop function 'cos
            ' to output
139             current output: 1.000000 16.000000 / sqrt 3.000000 2.000000 / 3.141593
                    * sin fabs + 3.141593 cos
140             current stack: + +
```

```
141    encounter NOTHING: flush the stack to output
142            current output: 1.000000 16.000000 / sqrt 3.000000 2.000000 / 3.141593
                   * sin fabs + 3.141593 cos + +
143    --- postfix expression transforming complete :) ---
144    Postfix Exp: 1.000000 16.000000 / sqrt 3.000000 2.000000 / 3.141593 * sin fabs
           + 3.141593 cos + +
145    RESULT: 0.250000
146    --- postfix expression transforming ---
147    encounter 0.00: push to output
148            current output: 0.000000
149    encounter +: push to the stack directly
150            current output: 0.000000
151            current stack: +
152    encounter 1.2: push to output
153            current output: 0.000000 1.200000
154            current stack: +
155    encounter NOTHING: flush the stack to output
156            current output: 0.000000 1.200000 +
157    --- postfix expression transforming complete :) ---
158    Postfix Exp: 0.000000 1.200000 +
159    RESULT: 1.200000
```