Big-O

```
Q1: O(\sqrt{n})
```

Base condition is when i*i = n, so this means # of time to execute the while loop is $O(\sqrt{n})$

```
Q2: O(3^n)
```

Each time a recursive call generates 3 recursive child calls, and each time n is reduced by 1, so the depth of recusion tree is O(N). Total number of nodes in recursion tree is $O(3^N)$, coming from summation over nodes at each level of tree, i.e 3^i where i is the level index start from 0. Each node takes O(1) time. As a result, the total complexity is $O(3^N) \cdot O(1) = O(3^N)$.

▼ Dynamic Array

```
1 import ctypes
 2
 3 class UserDefinedDynamicArray:
       def init (self,C=100):
 4
           self. n=0
 5
           self._capacity=C
 6
 7
           self._A=self._make_array(self._capacity)
 8
 9
       def rotate0(self, k):
10
           # your code (Perform rotate in place)
           # O(kN) solution, slower
11
           if k > 0:
12
             if k >= self._n:
13
               k = k \% self._n
14
15
             # move the first k elements to the back, one by one
16
17
             for i in range(k):
               c = self. A[0]
18
               # rotate elements to the left
19
               for j in range(1, self._n):
20
                 self.\_A[j-1] = self.\_A[j]
21
               self. A[self. n-1] = c
22
           elif k < 0:
23
             if k <= -self._n:</pre>
24
25
               k = -k \% self._n
26
             else:
               k = -k
27
28
             # move the last k elements to the front, one by one
29
             for i in range(k):
30
               c = self. A[self. n-1]
31
32
               # rotate elements to the right, starting from the back
               for j in range(self._n-2, -1, -1):
33
```

```
self.\_A[j+1] = self.\_A[j]
34
35
               self._A[0] = c
36
37
           return self
38
39
       def rotate(self, k):
40
           # your code (Perform rotate in place)
           # O(N) solution
41
42
           if k > 0:
43
             if k >= self._n:
               k = k \% self. n
44
45
46
             # now k refers to the first k elements
47
48
             # reverse the whole thing. The first k elements appear in the back
49
             self._reverse(0, self._n)
50
51
             # reverse the first N-k elements
             self._reverse(0, self._n-k)
52
53
54
             # reverse the last k elements
55
             self._reverse(self._n-k, self._n)
           elif k < 0:
56
57
             if k \le -self. n:
58
               k = -k \% self. n
             else:
59
               k = -k
60
61
62
             \# now k > 0 and refers to the last k elements
63
64
             # reverse the whole thing. The first k elements appear in the back
65
             self._reverse(0, self._n)
66
             # reverse the first k elements
67
68
             self._reverse(0, k)
69
70
             # reverse the last N-k elements
71
             self._reverse(k, self._n)
72
73
           return self
74
75
       def _reverse(self, I, J):
76
         i, j = I, J-1
77
         while i < j:
78
           self. A[j], self. A[i] = self. A[i], self. A[j]
79
           i += 1
80
           j -= 1
81
82
       def __len__(self):
83
           return self._n
84
85
       def append(self,x):
```

```
if self._n==self._capacity:
 86
 87
                self._resize(2*self._capacity)
 88
            self._A[self._n]=x
            self. n+=1
 89
 90
 91
        def _resize(self,newsize):
 92
            A=self. make array(newsize)
 93
            self._capacity=newsize
 94
            for i in range(self._n):
 95
                A[i]=self._A[i]
            self. A=A
 96
 97
 98
        def _make_array(self,size):
 99
            return (size*ctypes.py_object)()
100
101
        def getitem (self,i):
102
            if isinstance(i,slice):
103
                A=UserDefinedDynamicArray()
                # * operator was used to unpack the slice tuple
104
                for j in range(*i.indices(self._n)):
105
106
                    A.append(self._A[j])
107
                return A
108
            if i<0:
109
                i=self. n+i
110
            if not 0<=i<self._n:</pre>
                raise IndexError("Index out of range")
111
112
            return self._A[i]
113
114
        def __str__(self):
115
            return "[" \
116
                   +"".join( str(i)+"," for i in self[:-1]) \
117
                   +(str(self[-1]) if not self.is_empty() else "") \
                   +"]"
118
119
120
        def is empty(self):
121
            return self._n == 0
122
123
        def __iter__(self):
124
            for i in range(len(self)):
125
                yield self._A[i]
126
127
        def __setitem__(self,i,x):
128
            if i<0:
                i = i+self._n
129
130
131
            if not 0<=i<self. n:
132
                raise IndexError("Index out of range")
133
134
            self._A[i] = x
135
136 def main():
137
     a = UserDefinedDynamicArray(100)
```

```
138
139
     for i in range(5):
140
        a.append(i)
141
142
     print(a)
                        # Result: [0,1,2,3,4]
143
     print(a.rotate(1)) # Result: [1,2,3,4,0]
144
     print(a.rotate(1)) # Result: [2,3,4,0,1]
145
     print(a.rotate(2)) # Result: [4,0,1,2,3]
146
     print(a.rotate(-1))# Result: [3,4,0,1,2]
     print(a.rotate(-3))# Result: [0,1,2,3,4]
147
148
     print(a.rotate(0)) # Result: [0,1,2,3,4]
149
     print(a.rotate(6)) # Result: [1,2,3,4,0]
150
151 if __name__ == '__main__':
152
     main()
     [0,1,2,3,4]
     [1,2,3,4,0]
     [2,3,4,0,1]
     [4,0,1,2,3]
     [3,4,0,1,2]
     [0,1,2,3,4]
     [0,1,2,3,4]
     [1,2,3,4,0]
```

Queue

```
1 class ArrayQueue:
 2
 3
      DEFAULT CAPACITY = 5
 4
      def __init__(self):
 5
 6
           self. data = [None] * ArrayQueue.DEFAULT CAPACITY
           self._size = 0
7
           self._front = 0
 8
9
      def len_(self):
10
11
           return self. size
12
13
      def is empty(self):
14
           return self._size == 0
15
      def is full(self):
16
17
           return self. size == ArrayQueue.DEFAULT CAPACITY
18
      def first(self):
19
           if self.is_empty():
20
               raise Exception("Queue is Empty")
21
           return self._data[self._front]
22
23
24
      def dequeue(self):
```

```
25
           if self.is_empty():
               raise Exception("Queue is Empty")
26
           ans = self._data[self._front]
27
           self._data[self._front] = None
28
           self._front = (self._front + 1) % len(self._data)
29
30
           self._size -= 1
           return ans
31
32
       def enqueue(self, e):
33
           if self._size == len(self._data):
34
               raise Exception("Queue is Full")
35
           loc = (self._front + self._size) % len(self._data)
36
           self._data[loc] = e
37
           self._size += 1
38
39
40
       def __str__(self):
           return str(self._data)
41
42
43 class infiniteQueue:
       def __init__(self):
44
           #you can define more variables
45
           self. data = []
46
47
       def __len__(self):
48
           # return how many ArrayQueue in the infiniteQueue
49
50
           z = 0
51
           for q in self._data:
             z += len(q)
52
53
           return z
54
       def is empty(self):
55
           #check whether the infiniteQueue is empty or not
56
           return len(self) == 0
57
58
       def first(self):
59
           # Like the first() function in ArrayQueue,
60
           # but this time should return the first element from infiniteQueue
61
           if self.is_empty():
62
             raise Exception("infiniteQueue is Empty")
63
64
           # take the first queue
65
           q = self._data[0]
66
           ans = q.first()
67
68
           return ans
69
       def dequeue(self):
70
           # Like the dequeue() function in ArrayQueue,
71
           # but this time should dequeue from infiniteQueue
72
73
74
           # The expensive operation happens when the first queue becomes
           # empty and need to remove. (pop(0))
75
           # Otherwise, most of the time dequeue is O(1).
76
77
```

```
if self.is_empty():
 78
 79
              raise Exception("infiniteQueue is Empty")
 80
            # take the first queue
 81
 82
            q = self. data[0]
 83
            ans = q.dequeue()
 84
            if q.is empty():
 85
              # besides pop(0), we can also use front pointer to point to the
 86
              # first non-empty queue (will leave as an exercise to you)
 87
              self._data.pop(0)
 88
 89
            return ans
 90
 91
       def enqueue(self, e):
 92
            # O(1): no need to resize when the last ArrayQueue is full.
 93
            # Just create a new one.
 94
            # Like the enqueue() function in ArrayQueue,
 95
           # but this time should enqueue from infiniteQueue
 96
            if self.is empty() or self. data[-1].is full():
 97
              # create a new queue at the back
98
              self._data.append(ArrayQueue())
99
           # take the last queue
100
101
            q = self. data[-1]
102
            q.enqueue(e)
103
104
       def str (self):
105
            #should print out the string object as the comments shown in main().
106
            return str([q. data for q in self. data])
107
108 def main():
109
       Queue = infiniteQueue()
110
       Queue.enqueue(11)
111
       print(Queue) #[[11, None, None, None, None]]
112
       Queue.enqueue(3)
113
       print(Queue) #[[11, 3, None, None, None]]
114
       print(Queue.first()) #11
115
       Queue.enqueue(8)
       Queue.enqueue(4)
116
117
       Queue.enqueue(0)
       print(Queue) #[[11, 3, 8, 4, 0]]
118
119
       Queue.enqueue(9)
       print(Queue) #[[11, 3, 8, 4, 0][9, None, None, None, None]]
120
121
       print(Queue.first()) #11
122
       print(Queue.dequeue()) #11
123
       print(Queue.dequeue()) #3
124
       print(Queue.dequeue()) #8
125
       print(Queue.first()) #4
126
       print(Queue.dequeue()) #4
       print(Queue.dequeue()) #0
127
128
       print(Queue) #[[9, None, None, None, None]]
129
       Queue.enqueue(10)
130
        print(Oueue) #[[9, 10, None, None, None]]
```

```
131
        print(Queue.dequeue()) #9
132
        print(Queue.dequeue()) #10
        print(Queue) #[]
133
        #print(Queue.dequeue()) #"listofQueues is empty"
134
135
        Queue.enqueue(11)
        print(Queue) #[[11, None, None, None, None]]
136
        print(Queue.first()) #11
137
        Queue.enqueue(3)
138
139
        print(Queue) #[[11, 3, None, None, None]]
140
        Queue.enqueue(8)
        print(Queue) #[[11, 3, 8, None, None]]
141
142
143
144 if __name__ == '__main__':
145
        main()
     [[11, None, None, None, None]]
     [[11, 3, None, None, None]]
     11
     [[11, 3, 8, 4, 0]]
     [[11, 3, 8, 4, 0], [9, None, None, None, None]]
     11
     11
     3
     8
     4
     4
     0
     [[9, None, None, None, None]]
     [[9, 10, None, None, None]]
     10
     [[11, None, None, None, None]]
     [[11, 3, None, None, None]]
     [[11, 3, 8, None, None]]
```

▼ Recursion

```
1 %%writefile array_stack.py
2
3 """Basic example of an adapter class to provide a stack interface."""
4
5 class ArrayStack:
    """LIFO Stack implementation using a Python list as underlying storage."""
7
8
    def __init__(self, capacity=None):
      """Create an empty stack."""
9
      self._data = []
10
                                             # nonpublic list instance
      self._capacity = capacity
11
12
13
    def len (self):
```

```
"""Return the number of elements in the stack."""
14
15
       return len(self. data)
16
17
    def is empty(self):
18
       """Return True if the stack is empty."""
19
       return len(self._data) == 0
20
    def push(self, e):
21
22
       """Add element e to the top of the stack."""
23
       if self. capacity is not None and len(self) >= self. capacity:
         raise Exception('Max capacity is reached! cannot push anymore!')
24
25
       self._data.append(e)
                                              # new item stored at end of list
26
27
    def top(self):
28
       """Return (but do not remove) the element at the top of the stack.
29
30
       Raise Empty exception if the stack is empty.
31
32
       if self.is_empty():
33
         raise Exception('Stack is empty')
34
       return self. data[-1]
                                              # the last item in the list
35
36
    def pop(self):
37
       """Remove and return the element from the top of the stack (i.e., LIFO).
38
39
       Raise Empty exception if the stack is empty.
40
41
       if self.is_empty():
42
         raise Exception('Stack is empty')
43
       return self._data.pop()
                                              # remove last item from list
44
45
    def __repr__(self):
         return str(self._data)
46
    Writing array_stack.py
 1 from array_stack import ArrayStack
 2
 3 \text{ lefty} = '(\{[']
 4 \text{ righty} = ')}]'
 6 def check_parentheses_helper(X, i, stack):
 7
    if i >= len(X):
 8
       # after processing the last character, there should be no parentheses there
 9
       return stack.is empty()
10
11
    c = X[i]
    if c in lefty:
12
13
       # push open parenthesis into stack
       j = lefty.index(c)
14
15
       stack.push(j)
       return check parentheses helper(X, i+1, stack)
16
```

```
17
    elif c in righty:
      # got an closed parenthesis. So check the one on the top of stack
18
19
      j = righty.index(c)
      if not stack.is_empty() and stack.top() == j:
20
         # we got a match
21
22
         stack.pop()
23
         return check_parentheses_helper(X, i+1, stack)
24
      return False
25
    else:
      # got other symbols. Just skip and proceed
26
      return check_parentheses_helper(X, i+1, stack)
27
28
29 def check_parentheses(X):
    stack = ArrayStack()
30
31
    return check parentheses helper(X, 0, stack)
32
33 if __name _ == '__main__':
    print(check\_parentheses("(1+2)(())((((()))))")) # True
34
    print(check_parentheses("()(2+4)((3))()")) # True
35
36
    print(check_parentheses("()(()((((()))))")) # False
37
    print(check_parentheses("({})")) # True
    print(check_parentheses("({)}")) # False
38
    print(check_parentheses("(1+2)*(4+6)")) # True
39
    print(check_parentheses("(){({)")) # False
40
    print(check_parentheses("()(()))")) # False
41
    --NORMAL--
    True
    True
    False
    True
    False
    True
    False
    False
```

