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

```
import ctypes

class UserDefinedDynamicArray:
    def __init__(self,C=100):
        self._n=0
        self._capacity=C
        self._A=self._make_array(self._capacity)

    def rotate0(self, k):
        # your code (Perform rotate in place)
        # O(kN) solution, slower
        if k > 0:
          if k >= self._n:
            k = k % self._n

          # move the first k elements to the back, one by one
          for i in range(k):
            c = self._A[0]
            # rotate elements to the left
            for j in range(1, self._n):
              self._A[j-1] = self._A[j]
            self._A[self._n-1] = c
        elif k < 0:
          if k <= -self._n:
            k = -k % self._n
          else:
            k = -k

          # move the last k elements to the front, one by one
          for i in range(k):
            c = self._A[self._n-1]
            # rotate elements to the right, starting from the back
            for j in range(self._n-2, -1, -1):
              self._A[j+1] = self._A[j]
            self._A[0] = c

        return self

    def rotate(self, k):
        # your code (Perform rotate in place)
        # O(N) solution
        if k > 0:
          if k >= self._n:
            k = k % self._n

          # now k refers to the first k elements

          # reverse the whole thing. The first k elements appear in the back
          self._reverse(0, self._n)

          # reverse the first N-k elements
          self._reverse(0, self._n-k)

          # reverse the last k elements
          self._reverse(self._n-k, self._n)
        elif k < 0:
          if k <= -self._n:
            k = -k % self._n
          else:
            k = -k

          # now k > 0 and refers to the last k elements

          # reverse the whole thing. The first k elements appear in the back
          self._reverse(0, self._n)

          # reverse the first k elements
          self._reverse(0, k)

          # reverse the last N-k elements
          self._reverse(k, self._n)

        return self

    def _reverse(self, I, J):
      i, j = I, J-1
      while i < j:
        self._A[j], self._A[i] = self._A[i], self._A[j]
        i += 1
        j -= 1

    def __len__(self):
        return self._n

    def append(self,x):
        if self._n==self._capacity:
            self._resize(2*self._capacity)
        self._A[self._n]=x
        self._n+=1

    def _resize(self,newsize):
        A=self._make_array(newsize)
        self._capacity=newsize
        for i in range(self._n):
            A[i]=self._A[i]
        self._A=A

    def _make_array(self,size):
        return (size*ctypes.py_object)()

    def __getitem__(self,i):
        if isinstance(i,slice):
            A=UserDefinedDynamicArray()
            # * operator was used to unpack the slice tuple
            for j in range(*i.indices(self._n)):
                A.append(self._A[j])
            return A
        if i<0:
            i=self._n+i
        if not 0<=i<self._n:
            raise IndexError("Index out of range")
        return self._A[i]

    def __str__(self):
        return "[" \
               +"".join( str(i)+"," for i in self[:-1]) \
               +(str(self[-1]) if not self.is_empty() else "") \
               +"]"

    def is_empty(self):
        return self._n == 0

    def __iter__(self):
        for i in range(len(self)):
            yield self._A[i]

    def __setitem__(self,i,x):
        if i<0:
            i = i+self._n

        if not 0<=i<self._n:
            raise IndexError("Index out of range")

        self._A[i] = x

def main():
  a = UserDefinedDynamicArray(100)
  for i in range(5):
    a.append(i)

  print(a)          # Result: [0,1,2,3,4]
  print(a.rotate(1)) # Result: [1,2,3,4,0]
  print(a.rotate(1)) # Result: [2,3,4,0,1]
  print(a.rotate(2)) # Result: [4,0,1,2,3]
  print(a.rotate(-1))# Result: [3,4,0,1,2]
  print(a.rotate(-3))# Result: [0,1,2,3,4]
  print(a.rotate(0)) # Result: [0,1,2,3,4]
  print(a.rotate(6)) # Result: [1,2,3,4,0]

if __name__ == '__main__':
  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

```
class ArrayQueue:

    DEFAULT_CAPACITY = 5

    def __init__(self):
        self._data = [None] * ArrayQueue.DEFAULT_CAPACITY
        self._size = 0
        self._front = 0

    def __len__(self):
        return self._size

    def is_empty(self):
        return self._size == 0

    def is_full(self):
        return self._size == ArrayQueue.DEFAULT_CAPACITY

    def first(self):
        if self.is_empty():
            raise Exception("Queue is Empty")
        return self._data[self._front]

    def dequeue(self):
        if self.is_empty():
            raise Exception("Queue is Empty")
        ans = self._data[self._front]
        self._data[self._front] = None
        self._front = (self._front + 1) % len(self._data)
        self._size -= 1
        return ans

    def enqueue(self, e):
        if self._size == len(self._data):
            raise Exception("Queue is Full")
        loc = (self._front + self._size) % len(self._data)
        self._data[loc] = e
        self._size += 1

    def __str__(self):
        return str(self._data)

class infiniteQueue:
    def __init__(self):
        #you can define more variables
        self._data = []

    def __len__(self):
        # return how many ArrayQueue in the infiniteQueue
        z = 0
        for q in self._data:
          z += len(q)
        return z

    def is_empty(self):
        #check whether the infiniteQueue is empty or not
        return len(self) == 0

    def first(self):
        # Like the first() function in ArrayQueue,
        # but this time should return the first element from infiniteQueue
        if self.is_empty():
          raise Exception("infiniteQueue is Empty")

        # take the first queue
        q = self._data[0]
        ans = q.first()
        return ans

    def dequeue(self):
        # Like the dequeue() function in ArrayQueue,
        # but this time should dequeue from infiniteQueue

        # The expensive operation happens when the first queue becomes
        # empty and need to remove. (pop(0))
        # Otherwise, most of the time dequeue is O(1).

        if self.is_empty():
          raise Exception("infiniteQueue is Empty")

        # take the first queue
        q = self._data[0]
        ans = q.dequeue()
        if q.is_empty():
          # besides pop(0), we can also use front pointer to point to the
          # first non-empty queue (will leave as an exercise to you)
          self._data.pop(0)

        return ans

    def enqueue(self, e):
        # O(1): no need to resize when the last ArrayQueue is full.
        # Just create a new one.
        # Like the enqueue() function in ArrayQueue,
        # but this time should enqueue from infiniteQueue
        if self.is_empty() or self._data[-1].is_full():
          # create a new queue at the back
          self._data.append(ArrayQueue())

        # take the last queue
        q = self._data[-1]
        q.enqueue(e)

    def __str__(self):
        #should print out the string object as the comments shown in main().
        return str([q._data for q in self._data])

def main():
    Queue = infiniteQueue()
    Queue.enqueue(11)
    print(Queue) #[[11, None, None, None, None]]
    Queue.enqueue(3)
    print(Queue) #[[11, 3, None, None, None]]
    print(Queue.first()) #11
    Queue.enqueue(8)
    Queue.enqueue(4)
    Queue.enqueue(0)
    print(Queue) #[[11, 3, 8, 4, 0]]
    Queue.enqueue(9)
    print(Queue) #[[11, 3, 8, 4, 0][9, None, None, None, None]]
    print(Queue.first()) #11
    print(Queue.dequeue()) #11
    print(Queue.dequeue()) #3
    print(Queue.dequeue()) #8
    print(Queue.first()) #4
    print(Queue.dequeue()) #4
    print(Queue.dequeue()) #0
    print(Queue) #[[9, None, None, None, None]]
    Queue.enqueue(10)
    print(Queue) #[[9, 10, None, None, None]]
    print(Queue.dequeue()) #9
    print(Queue.dequeue()) #10
    print(Queue) #[]
    #print(Queue.dequeue()) #"listofQueues is empty"
    Queue.enqueue(11)
    print(Queue) #[[11, None, None, None, None]]
    print(Queue.first()) #11
    Queue.enqueue(3)
    print(Queue) #[[11, 3, None, None, None]]
    Queue.enqueue(8)
    print(Queue) #[[11, 3, 8, None, None]]


if __name__ == '__main__':
    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]]
9
10
[]
[[11, None, None, None, None]]
11
[[11, 3, None, None, None]]
[[11, 3, 8, None, None]]
```

Recursion

```
%%writefile array_stack.py

"""Basic example of an adapter class to provide a stack interface."""

class ArrayStack:
  """LIFO Stack implementation using a Python list as underlying storage."""

  def __init__(self, capacity=None):
    """Create an empty stack."""
    self._data = []                       # nonpublic list instance
    self._capacity = capacity

  def __len__(self):
    """Return the number of elements in the stack."""
    return len(self._data)

  def is_empty(self):
    """Return True if the stack is empty."""
    return len(self._data) == 0

  def push(self, e):
    """Add element e to the top of the stack."""
    if self._capacity is not None and len(self) >= self._capacity:
      raise Exception('Max capacity is reached! cannot push anymore!')
    self._data.append(e)                  # new item stored at end of list

  def top(self):
    """Return (but do not remove) the element at the top of the stack.

    Raise Empty exception if the stack is empty.
    """
    if self.is_empty():
      raise Exception('Stack is empty')
    return self._data[-1]                 # the last item in the list

  def pop(self):
    """Remove and return the element from the top of the stack (i.e., LIFO).

    Raise Empty exception if the stack is empty.
    """
    if self.is_empty():
      raise Exception('Stack is empty')
    return self._data.pop()               # remove last item from list

  def __repr__(self):
      return str(self._data)
```

```
Writing array_stack.py
```

```
from array_stack import ArrayStack

lefty = '({['
righty = ')}]'

def check_parentheses_helper(X, i, stack):
  if i >= len(X):
    # after processing the last character, there should be no parentheses there
    return stack.is_empty()

  c = X[i]
  if c in lefty:
    # push open parenthesis into stack
    j = lefty.index(c)
    stack.push(j)
    return check_parentheses_helper(X, i+1, stack)
```

```python
  elif c in righty:
    # got an closed parenthesis. So check the one on the top of stack
    j = righty.index(c)
    if not stack.is_empty() and stack.top() == j:
      # we got a match
      stack.pop()
      return check_parentheses_helper(X, i+1, stack)
    return False
  else:
    # got other symbols. Just skip and proceed
    return check_parentheses_helper(X, i+1, stack)

def check_parentheses(X):
  stack = ArrayStack()
  return check_parentheses_helper(X, 0, stack)

if __name__ == '__main__':
  print(check_parentheses("(1+2)(())((((())))))")) # True
  print(check_parentheses("()(2+4)((3))()")) # True
  print(check_parentheses("()(()((((()))))")) # False
  print(check_parentheses("({})")) # True
  print(check_parentheses("({)}")) # False
  print(check_parentheses("(1+2)*(4+6)")) # True
  print(check_parentheses("(){({}")) # False
  print(check_parentheses("()(()))")) # False
```

--NORMAL--

```
True
True
False
True
False
True
False
False
```