

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

```

34         self._A[j+1] = self._A[j]
35         self._A[0] = c
36
37     return self
38
39     def rotate(self, k):
40         # your code (Perform rotate in place)
41         # O(N) solution
42         if k > 0:
43             if k >= self._n:
44                 k = k % self._n
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
52             self._reverse(0, self._n-k)
53
54             # reverse the last k elements
55             self._reverse(self._n-k, self._n)
56         elif k < 0:
57             if k <= -self._n:
58                 k = -k % self._n
59             else:
60                 k = -k
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
67             # reverse the first k elements
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):

```

```

86         if self._n==self._capacity:
87             self._resize(2*self._capacity)
88         self._A[self._n]=x
89         self._n+=1
90
91     def _resize(self,newsiz):
92         A=self._make_array(newsiz)
93         self._capacity=newsiz
94         for i in range(self._n):
95             A[i]=self._A[i]
96         self._A=A
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()
104             # * operator was used to unpack the slice tuple
105             for j in range(*i.indices(self._n)):
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:
111             raise IndexError("Index out of range")
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:
129             i = i+self._n
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]
147 print(a.rotate(-3))# Result: [0,1,2,3,4]
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
5     def __init__(self):
6         self._data = [None] * ArrayQueue.DEFAULT_CAPACITY
7         self._size = 0
8         self._front = 0
9
10    def __len__(self):
11        return self._size
12
13    def is_empty(self):
14        return self._size == 0
15
16    def is_full(self):
17        return self._size == ArrayQueue.DEFAULT_CAPACITY
18
19    def first(self):
20        if self.is_empty():
21            raise Exception("Queue is Empty")
22        return self._data[self._front]
23
24    def dequeue(self):

```

```

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

```

```

78         if self.is_empty():
79             raise Exception("infiniteQueue is Empty")
80
81         # take the first queue
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
100        # take the last queue
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)
116     Queue.enqueue(4)
117     Queue.enqueue(0)
118     print(Queue) #[[11, 3, 8, 4, 0]]
119     Queue.enqueue(9)
120     print(Queue) #[[11, 3, 8, 4, 0][9, None, None, None, None]]
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
127     print(Queue.dequeue()) #0
128     print(Queue) #[[9, None, None, None, None]]
129     Queue.enqueue(10)
130     print(Queue) #[[9, 10, None, None, None]]

```

```

131 print(Queue.dequeue()) #9
132 print(Queue.dequeue()) #10
133 print(Queue) #[]
134 #print(Queue.dequeue()) #"listofQueues is empty"
135 Queue.enqueue(11)
136 print(Queue) #[[11, None, None, None, None]]
137 print(Queue.first()) #11
138 Queue.enqueue(3)
139 print(Queue) #[[11, 3, None, None, None]]
140 Queue.enqueue(8)
141 print(Queue) #[[11, 3, 8, None, None]]
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]]
9
10
[]
[[11, None, None, None, None]]
11
[[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:
6     """LIFO Stack implementation using a Python list as underlying storage."""
7
8     def __init__(self, capacity=None):
9         """Create an empty stack."""
10         self._data = [] # nonpublic list instance
11         self._capacity = capacity
12
13     def __len__(self):

```

```

14     """Return the number of elements in the stack."""
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
21     def push(self, e):
22         """Add element e to the top of the stack."""
23         if self._capacity is not None and len(self) >= self._capacity:
24             raise Exception('Max capacity is reached! cannot push anymore!')
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):
46         return str(self._data)

```

Writing array\_stack.py

```

1 from array_stack import ArrayStack
2
3 lefty = '({['
4 righty = ')}]'
5
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]
12     if c in lefty:
13         # push open parenthesis into stack
14         j = lefty.index(c)
15         stack.push(j)
16     return check_parentheses_helper(X, i+1, stack)

```




```

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

```

--NORMAL--

 True  
 True  
 False  
 True  
 False  
 True  
 False  
 False

