PS2-231830106

Problem Set 2

Problem 1

1. 简介：利用双栈，用操作符栈储存 $+ \times$ ()，用数字栈储存数字
   1.1:接收到数字压入num_stack
   1.2:接收到 (压入opt_stack
   1.3:接收到) 从opt_stack中取出操作符直至取出 (
   1.4:接收到$+\times$根据优先级从num_stack中取出数进行运算
2. 伪代码：

```
function evaluateInfix(expression):
    num_stack = empty stack
    opt_stack = empty stack

    precedence = {'+': 1, '×': 2}

    for token in expression:
        if token is a number:
            num_stack.push(int(token))
        else if token is '(':
            opt_stack.push(token)
        else if token is ')':
            while opt_stack.top() != '(':
                performOperation(num_stack, opt_stack)
            opt_stack.pop()  // pop '('
        else if token is an operator:
            while not opt_stack.isEmpty() and opt_stack.top() is not '('
and precedence[opt_stack.top()] >= precedence[token]:
                performOperation(num_stack, opt_stack)
            opt_stack.push(token)

    while not opt_stack.isEmpty():
        performOperation(num_stack, opt_stack)

    return num_stack.pop()

function performOperation(num_stack, opt_stack):
    operator = opt_stack.pop()
```

```
    number2 = num_stack.pop()
    number1 = num_stack.pop()
    result = applyOperation(operator, number1, number2)
    num_stack.push(result)

function applyOperation(operator, number1, number2):
    if operator == '+':
        return number1 + number2
    else if operator == '×':
        return number1 * number2
```

3.时间复杂度:
在线性栈中进行操作，每个元素只被操作至多两次，因此时间复杂度为O(n)

Problem 2

$T(n) = T(\alpha n) + T((1-\alpha)n) + cn$ 不妨设0<a<$\frac{1}{2}$

(a): 递归树:

a.1: 树的高度: $\log_{1/a} n$ 又 0<a<$\frac{1}{2}$，可认为树的高度约为$O(\lg n)$
a.2: 第一层: T(n)
第二层: $T(\alpha n) + T((1-\alpha)n)$
第三层: $T(\alpha^2 n) + 2T(\alpha(1-\alpha)n) + T((1-\alpha)^2 n)$
……
可以看出每一层的节点和为T(n)
a.3: 用高度 $\times$ 每层的节点和: 即$T(n) = O(\lg n) \times O(n) = O(n \lg n)$
a.4: 又每层的节点和至少为$\Omega(n)$高度至少为$\Omega(\lg n)$，显然$T(n) = \Omega(n) \times \Omega(\lg n) = \Omega(n \lg n)$
a.5: 即证

(b):

1. $T(n) = 3T(n/3-2) + n/2$
   $O(n \lg n) \quad \Omega(n \lg n)$
2. $T(n) = 4T(n/2) + n^2\sqrt{n}$
   $O(n^2\sqrt{n}) \quad \Omega(n^2\sqrt{n})$
3. $T(n) = T(n-2) + \lg n$
   $O(n \lg n) \quad \Omega(n \lg n)$
4. $T(n) = \sqrt{n}T(\sqrt{n}) + n$
   $O(n) \quad \Omega(n)$

5. $T(n) = T(n/2) + T(n/4) + n$
   $O(n \lg n) \quad \Omega(n)$

Problem 3

(a):

```
MERGESORT(left,right,A):
        mid = (left+right)//2
        MERGESORT(left,mid,A)
        MERGESORT(mid+1,right,A)
        MERGE(left,right,mid,A)

MERGE(left,right,mid,A):
        L1 = mid - left + 1
        L2 = right -mid

        for i in range(L1-1):
                L[i] = A[i]
        for i in range(L2-1):
                R[i] = A[mid+i+1]

        i = 0, j = 0
        k = left
        while i < L1 and j < L2:
                if  L[i] <= R[j]:
                        A[k] = L[i]
                        i++
                else:
                        A[k] = R[j]
                        j++
                k++

        while i<L1:
                A[k] = L[i]
                k++
                i++

        while j<L2:
                A[k] = R[j]
                k++
                j++
```

时间复杂度：每一层合并都需要n次，深度为$\lg n$ 因此时间复杂度为$O(n\lg n)$
空间复杂度：开辟了2个长度为n/2的数组因此空间复杂度为$O(n)$

(b):

```
MERGESORT_LIST(head):
        if head is null and head.next is null:
                return head

        middle = GET_MIDDLE(head)
        next_to_middle = middle.next
        middle.next = null

    left = MERGESORT_LIST(head)
    right = MERGESORT_LIST(next_to_middle)

        result = MERGE_LIST(left,right)
        return result

GET_MIDDLE(head):
        if head is null:
                return head

        slow = head
        fast = head

        while fast.next is not null and fast.next.next is not null:
                fast = fast.next.next
                slow = slow.next


        return slow

MERGE_LIST(left,right)
        dummy = new Listnode(0)
        tail = dummy

        while left is not null and right is not null:
                if left.value <= right.value:
                        tail.next = left
                        left = left.next
                else:
```

```
                        tail.next = right
                        right = right.next
                tail = tail.next

                if left.next is not null:
                        tail.next = left
                if right.next is not null:
                        tail.next = right

        return dummy.next
```

时间复杂度：每一层合并都需要n次，深度为$\lg n$ 因此时间复杂度为$O(n \lg n)$
空间复杂度：链接链表无需额外空间，空间复杂度只又递归深度决定为$O(\lg n)$

Problem 4

(a):

已知：$x^2 = (x_1 \times 10^{n/2} + x_2)^2$
解：
设$m = x_1 \times 10^{n/2}$，$n = x_2$，则$x^2 = (m+n)^2 = m^2 + n^2 + 2mn = m^2 = m^2 + n^2 + (m^2 + n^2 - (m -$
则$T(n) = 3 \times T(n/2)$，又主定理可推出时间复杂度为$O(n^{\log_2 3})$
伪代码：

```
function K_square(x):
    if x is a small number:
        return x^2

    n = number of digits in x
    m = ceil(n / 2)

    x1 = high m digits of x
    x0 = low m digits of x

    z2 = K_square(x1)
    z0 = K_square(x0)
    z1 = K_square(x1 + x0) - z2 - z0

    return z2 * 10^(2*m) + z1 * 10^m + z0
```

(b):

$xy = ((x+y)^2 - (x-y)^2)/4$
显然两者的时间复杂度应该趋近相同

Problem 5

二分查找即可
伪代码实现：

```
function Binary_search(A, n):
    left = 1
    right = n
    while left <= right:
        mid = floor((left + right) / 2)

                if A[mid] == mid:
            return mid

        if A[mid] > mid:
            right = mid - 1
        else:
            left = mid + 1

    return -1
```

显然二分查找的时间复杂度为$O(\lg n)$

Problem 6

(a):

1. 思路：分治算法
   1.1：将A[1…n]分为left[1…n/2]和right[n/2…n]在right和left中分别寻找最有可能的多数元素
   1.2: 若left的可能多数元素与right的可能多数元素相同，则直接返回。若不同，先检查返回值是否为-1，若不是-1则将数组扫描对可能多数元素计数，返回出现次数较多的元素。
   1.3: 最后一次若返回的元素数量不超过数组长度的1/2则返回-1
2. 伪代码：

```
function FindMajority(left,right,A):
        if left == right:
                return A[left]
```

```
        mid = (left+right)//2
        leftmajority = FindMajority(left,mid,A)
        rightmajority = FindMajority(mid+1,right,A)

        if leftmajority == rightmajority:
                return leftmajority

        if leftmajority != -1:
                leftcount = count(left,right,leftmajority,A)
        else:
                leftcount = 0
        if rightmajority != -1:
                rightcount = count(left,right,rightmajority,A)
        else:
                rightcount = 0

        if leftcount > (right-left+1)//2:
                return leftmajority
        if rightcount > (right-left+1)//2:
                return rightmajority

        return -1

function count(left,right,target,A):
        count = 0
        for i in range(left,right+1):
                if A[i]==target:
                        count++
        return count
```

3. 正确性：
   运用了分治算法，确保了每一个子问题的正确性，因此算法一定正确
4. 时间复杂度：
   $T(n) = 2T(n/2) + 2n$ 根据主定理可得时间复杂度为$O(n \lg n)$

(b)：

1. 思路：（灵感由chatgpt提供思路和代码自己构建）
   1.1: 将元素之间相互抵消，确保如果有一个元素出现次数>n/2，那么它一定能够被筛选出来。
   1.2: 维护一个候选元素，和一个计数器。若候选元素与后一个元素相同则计数器+1不同

则-1，当计数器归零时重置候选元素并重置计数器为1。
1.3: 一次遍历后如果存在多数元素则其一定为候选元素，但候选元素不一定是多数元素，所以在进行一次遍历确认元素为候选元素。

2. 伪代码

```
function find(A):
        candidate = None
        count = 0

        for num in A:
                if count == 0:
                        candidate = num
                        count = 1
                elif candidate != num:
                        count -= 1
                elif candidate == num:
                        count +=1


    count = 0
    for num in A:
            if num == candidate:
                    count += 1:

        if count > len(A)//2
                return candidate
        else:
            return -1
```

3. 正确性：
若存在多数元素α则，在第一轮遍历后，由于α的数量>$\frac{1}{2}len(A)$因此A不会被抵消最后剩下的candidate一定为A。如果不存在多数元素α则无论第一轮遍历结果，第二轮遍历会判断不存在这样一个α，函数返回-1。
4. 时间复杂度：
对数组只进行了两轮遍历，因此时间复杂度为O(n)。·