Ps2-201300066

麻超　201300066

南京大学人工智能学院

1　Problem 1

1.1　a

解：在一个单向链表中，一个节点只能指向下一个节点，因此考虑将该节点指向上一个节点。由于需要不占用额外空间，故考虑在该链表内部操作。

```
1.Rerverse(Node,head):
    Node pre=NULL
    While(head ! =NULL):
        Node next = head.next()
        pre = head.next()
        pre = head
        head = next
    return pre
```

最后返回的 pre 恰好为该链表的最后一个节点，且每一个节点的 next 节点都是原链表中的上一个节点，完成了反转，且没有占用其他空间，时间复杂度为 $\Theta(n)$-time.

1.2　b

$\because x.np = x.next \oplus x.prev$

$\therefore x.next = x.np \oplus x.prev, x.prev = x.np \oplus x.next$

故而只需要知道该节点下一个节点的位置就可以求出上一个节点的位置，反之亦然。

由于每次需要两个节点才能够访问下一个节点或上一个节点，故头节点应使用两个节点来表示。

因此可以考虑用头节点和尾节点来作为这两个节点，即令

$head.np \oplus tail = head.next, tail.np \oplus head = tail.prev$

从当前节点前往下一个节点的语句如下：

```
prev = x
x = next
next = prev⊕x.np
```

从当前节点前往前一个节点的语句如下：

```
nextp = x
x = prev
prev = x.np⊕ next
```

以下为需要的伪代码：

```
1.Insert(Node,int i,p):
    x=head,prev=tail
    next=head.np⊕prev
    for k in [1,i]:
        prev = x
        x = next
        next = prev⊕x.np
    p=x.np⊕prev
    prev=x
    next=p.np⊕x
    x=p 2.Delete(Node,p):
    x=head,prev=tail
    next=head.np⊕prev
    while next !=p:
        prev = x
        x = next
        next = prev⊕x.np
    next = p.np⊕x
    next = x.np⊕prev
```

free(p)

2 Problem 2

解：首先要有一个 stack 结构，命名为 S，作为基本。其 pop(),push(x) 操作同一般的 stack，唯一需要考虑的即为 Max() 方法。

可以考虑对该 stack 进行遍历，找寻最大值。

令 S.top=k，即为该 stack 的最顶部元素的序号

1.Push(S,x):

```
    S.top=S.top+1
    S[top+1]=x 2.Pop(S):
    if S.top=0:
        return error
    else:
        S.top=S.top-1
        return S[S.top+1]
```

3.Max(S):

```
    if S.top=0:
        return error
    else:
        maxnum=S[0]
        for i in [0,k]:
        if S[i]>maxnum:
            maxnum=S[i]
        return maxnum
```

但是该方法下 push 及 pop 操作时间复杂度都为 $\Theta(1)$, 该 max 操作的时间复杂度因为 $\Theta(n)$-time. 空间复杂度为 1*O(1).

3 Problem 3

将中缀表达式转化为后缀表达式时，可以发现，数字的排列顺序是正向的，但运算符号为逆向的，因此考虑用队列储存数字，用栈储存运算符号。方法大

致如下：

1. 遇见数字时可直接输出（或储存于队列中），并读取下一个字符

2. 如果读到括号, 则跳过, 到下一个字符

4. 如果读到的是运算符, 则要将此运算符入栈

(1) 若栈空, 则直接入栈;

(2) 若栈不空则要判断栈顶运算符的优先级:

<1> 若栈顶运算符的优先级低于要入栈的运算符, 直接入栈

<2> 若栈顶运算符的优先级高于要入栈的运算符, 要将栈顶位置的高优先级的运算符出栈, 出栈的运算符输出到结果字符串的末尾., 直到栈顶运算符优先级低于要入栈的运算符为止.

伪代码如下：

在运算符优先级方面令！＞×＞+

```
for i=0 to n:
    if A[i] belong to [0,9],print A[i]
    else:
        if S is empty:
            S.push(A[i])
        else:
            while S.top()>A[i]:
                print(S.top())
        S.pop()
while S is not empty :
    print(S.top())
        S.pop()
```

时间复杂度上，由于原算式的每个字符只进行入栈和出栈操作，故时间复杂度为O(n).

4 Problem 4

4.1 a

$$T(n) = 5 \times T(n/2) + O(n)$$

$$= 5 \times (5 \times (n/4) + O(n/2)) + O(n)$$

$= 125T(n/8) + [25O(n/4) + 4O(n/2) + O(n)]$

$n = 2^k, [25O(n/4) + 5O(n/2) + O(n)] = O(n * lgn)$

$2^k T(n/2^k) = cn$

$T(n) = O(n * lg_n + cn) = O(n * lgn)$

时间复杂度为 O(n*lgn)

4.2 b

$$T(n) = 2 * T(n-1) + O(1) = 4 * T(n-2) + 2 * O(1)$$

$=\cdots\cdots$

$= 2^n T(0) + n * O(1)$

$= O(2^n)$

时间复杂度为 $O(2^n)$

4.3 c

$$T(n) = 9 \times T(n/3) + O(n^2)$$

$= 9 \times (9 \times (n/9) + O(n^2/4)) + O(n^2)$

$= 729T(n/27) + [81O(n^3/8) + 9O(n^2)/4 + O(n^2)]$

$n = 3^k, [81O(n^2/16) + 9O(n^2/4) + O(n^2)] = O(n^2 * lgn)$

$T(n) = O(n^2 lgn)$ 故 a) 算法的时间复杂度最小，选用 a) 算法。

5 Problem 5

需要在 O(nlg_n) 时间内完成对数组中重复元素的筛查，主要的方法应是先排序再删除其中重复的元素。

伪代码如下：

```
Select(A[],n):
sort(A,n)
int result[]
for i=0 to n-1:
    insert(result[],A[i])
    while A[i]==A[i+1]:i++
```

return result 时间复杂度分析: 对数组的筛查等于对数组的排序加上对数组的线性遍历，故为 O(nlgn)+O(n)=O(nlgn)

证明：当该数组内每个元素都不相同时，该算法没有遗漏掉任何一个元素，是正确的。

当该数组内有 n 个元素相同时，在 while 循环内，一共循环了 n-1 次，恰好将原重复的 n 个元素略去了 n-1 个，保留了一个。因此，无论有多少重复元素，该算法总是正确的。

6 Problem 6

<2,3,8,6,1>

6.1 a

(1,5),(2,5),(3,5),(4,5),(3,4)

6.2 b

插入排序的运行时间是反转次数的常数倍. 对于每一个 j，若有一个符合上述条件的 i, 则需要执行一次反转操作，反转数 +1，一直需要对 j 从 1 到 n 执行 while 循环。所以插入排序所需要的时间数为 $\sum_{i=1}^{n} f(i)$, 为故其运行时间是反转次数的常数倍。

6.3 c

在归并排序里，逆序对的总数 = 左边数组中的逆序对的数量 + 右边数组中逆序对的数量 + 左右结合成新的顺序数组时中出现的逆序对的数量.

```
Megersort(A,1,n):
l=1,r=n
mid=(l+r)/2
count=Megersort(l,mid)+Megersort(mid+1,r)
L=l,R=mid+1,ans=l,p=[],q=[]
for i=l to mid:
    p.insert(A[i])
```

```
for i=mid+1 to r:
    q.insert(A[i])
while L<=mid and R<=r:
    if p[L]<=q[R]:
        A[ans++]=p[L++]
        count=R-mid-1+count
    else
        A[ans++]=q[R++]
while(L<=mid)
    A[ans++]=p[L++]
    count+=r-mid
while(R<=r)
    A[ans++]=q[R++]
return count
```