No / Date

001 算法 PS 2

1、

(a)

$x = L.head$

~~while x.next ≠ NIL~~

~~x.next.next ≠ x~~

~~x = x.next~~

$\boxed{\begin{array}{l} x = NIL \\ y = NIL \\ z = L.head \\ \text{while } z \neq NIL \\ \quad x = y \\ \quad y = z \\ \quad z = y.next \\ \quad y.next = x \end{array}}$

(b) 异或计算中：

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

$\Rightarrow \begin{cases} x.prev = x.np \oplus x.next \\ x.next = x.prev \oplus x.np \end{cases}$

① 单改双.

$x = L.head \quad y = NIL \quad L.tail = NIL$

while $x \neq NIL$

    $y = x$

    $x = x.next$

    $x.prev = y$

② 初始化赋值：只需知道第一个元素的 value 即可

$x = L.head \quad x.prev.value = 0$

while $x \neq NIL$

    $x.next.value = x.value \oplus x.prev.value$

    $x = x.next$

③ Insert (x, i) ：插入值可通过已知值算出

$y = L.head$

for $j = 1$ to $i-2$

    $y = y.next$

$z = y.next$

$y.next = x$

$x.next = z$

$x.prev = y$

$z.prev = x$

$x.value = y.value \oplus z.value$

④ Delete(i)

```
y = L.head
for j = 1 to i-2
    y = y.next
z = y.next
k = z.next
y.next = k
k.prev = y
while k ≠ NIL
    k = k.next
    k.value = k.prev.value ⊕ k.prev.prev.value
```

Correctness：
由于第一元素的 value 固定
只能依次改变所有删除
位置之后的 value

2. 思想：另使用一个 maxstack. 使其顶部元素始终是 MAXSTACK 中最大

① Push(x)

```
if x > MAXSTACK.peek()
    maxstack.push(x)
MAXSTACK.push(x)
```

② Pop(x)

```
return MAXSTACK.pop()
```

③ Max()

```
return maxstack.pop()
```

空间复杂度 O(n)

3、字符串存储算式，把！看作不动，先替换 x 的位置，最后是 +

```
Infix_to_postfix (str)
    for i = 0 to len(str)-1
        if str[i] == 'x'
            if str[i+2] ≠ '!'
                str[i] <-> str[i+1]
                i ← i+1
            else
                str[i] <-> str[i+1]
                str[i+1] <-> str[i+2]
    j = 0   /* 将第一个'+'移到字符串末尾即可 */
    while str[j] ≠ '+'
        j ← j+1
    for i = j to len(str)-1
        str[j] <-> str[j+1]
        j ← j+1
```

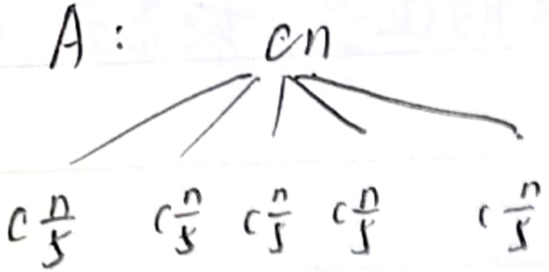
步骤：① 把'!'和数字看成一个数字整体，顺序不再变

② 把每个'x'和它后面最近的一个数字整体~~移动~~互换位置

③ 把第一个'+'移到最后面

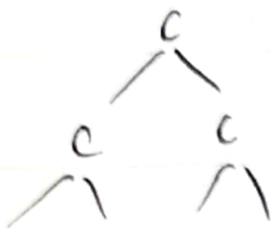
time complexity : $O(n)$

4. A:



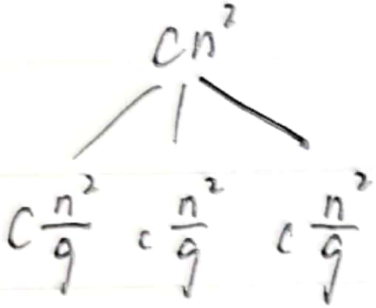
$\Rightarrow O(n\log_5 n)$

B:



$\Rightarrow O(n)$

C:



$\Rightarrow$ 总和 $< \frac{3}{2}n^2$，即 $O(n^2)$

综上，B的running times最优，选B

5. 分治法

先mergesort再去重，merge这一部分与讲义相同
每次分成2个子问题

伪代码：

① Merge (A, p, q, r)　　A是数组，A[p...q]，A[q+1...r]已排序去重
　详见ppt　　Merge完成排序

② Duplicates (A, p, q, r)　　对已Merge排序的两个对象进行去重

j = 1
A = Merge (A, p, q, r)

```
Let B[1...r-p+1] be new array
for i = p to r
    if A[i] = A[i+1]
        if A[i+1] < A[i+2]
            B[j] = A[i+2]
            j = j+1
return B
```

去重后存入B

③ Divide (A, p, r)

```
if p<r
    q = ⌊(p+r)/2⌋
    Divide (A, p, q)
    Divide (A, q+1, r)
    Duplicates (A, p, q, r)
```

running times：Duplicates 与 Merge 均为 $O(r-p)$
　总共为 $O(n\log n)$

correctness：每次Duplicates后得到的均为已排序去重的新数组

6. (a) ~~(8,6) (6,1)~~ (3,4) (4,5) (如果 A[1]=2)

(b) 每有一个 inversion，代表需要进行一轮插入.

证明：假设一共有 x 个 inversion

最不利情况：这 x 次插入都最不利

$$(n-1)+(n-2)+\cdots+(n-x) \Rightarrow O(xn)$$

(c)

```
Count (A, p, r)
    if p < r
        q = ⌊(p+r)/2⌋
        if A[q] > A[q+1]
            return Count(A, p, q) + Count(A, q+1, r) + 1
        else
            return Count(A, p, q) + Count(A, q+1, r)
```