HW5

- 姓名：吴欣怡
- 学号：PB21051111

Q1

(1)

```
#include<stdio.h>
#include<stdlib.h>

#define  OK     1
#define  ERROR   0

typedef  int  Elemtype;//数据类型重定义
typedef  int   Status;//状态类型重定义
typedef struct LNode {
    Elemtype data;   //数据域
    struct LNode* next;//指针域
    struct LNode* front;
    struct LNode* child;
}LNode, * Linklist;

Status LengthList(Linklist* L); //函数声明
Status Init_linklist(Linklist* L)
{
    *L = (Linklist)malloc(sizeof(LNode));//创建头结点
    if (!(*L))                           //创建失败
    {
        return ERROR;
    }
    (*L)->next = NULL;                     //将头结点的指针指向为空
    (*L)->front = NULL;
    return OK;
}
/*创建带头结点的单链表尾插法*/
void Creat_Linklist(Linklist* L, int n)
{
    Linklist p, q;
    int i;
    p = *L;
    q = (Linklist)malloc(sizeof(LNode));//创建新的节点
    scanf("%d", &q->data);
    p->next = q;                        //头结点指针指向新生成的节点
    q->front = p;
    p = q;
    p->child = NULL;
    for (i = 1;i < n;i++)
    {
        q = (Linklist)malloc(sizeof(LNode));//创建新的节点
        scanf("%d", &q->data);
        p->next = q;                        //头结点指针指向新生成的节点
```

```
        q->front = p;
        if (p->child == NULL && p->data - q->data == 1)
            p->child = q;
        if (q->child == NULL && q->data - p->data == 1)
            q -> child = p;
        p = q;                              //指针指向链尾
        p->child = NULL;
    }
    p->next = NULL;                         //链尾节点的指针指向为空
}
/* 删除有孩子节点的节点中最大的（之一） */
Status DeleteMaxChildNode(Linklist* L, int* deletedCount) {
    Linklist p = *L, maxNode = NULL, prevNode = NULL;

    while (p->next) {
        if (p->child) {
            if (!maxNode || p->child->data > maxNode->child->data) {
                maxNode = p;
                prevNode = p->front;
            }
        }
        p = p->next;
    }

    if (maxNode) {
        Linklist childNode = maxNode->child;
        maxNode->child = NULL;
        free(childNode);
        if (prevNode) {
            Linklist freeNode = maxNode;
            *deletedCount += 1;
            prevNode->next = maxNode->next;
            maxNode->next->front = prevNode;
            if (prevNode->data - prevNode->next->data == 1 && prevNode->child ==
NULL)
                prevNode->child = prevNode->next;
            if(prevNode->data - prevNode->next->data == -1 && prevNode->next-
>child == NULL)
                prevNode->next->child = prevNode;
            free(freeNode);
        }
        return OK;
    }
    return ERROR;
}

int main()
{
    int n, value, deletedCount = 0;
    Linklist L;
    /*单链表的初始化*/
    value = Init_linklist(&L);
    if (value)
        printf("单链表初始化成功！\n");
    else return ERROR;
```

```
    printf("请输入单链表的长度:");
    scanf("%d", &n);
    Creat_Linklist(&L, n);

    while (DeleteMaxChildNode(&L, &deletedCount) == OK)
        ;

    printf("总共删除了%d个节点\n", deletedCount);

    return 0;
}
```

算法解释：构建一个带头节点的双向链表，其中每个节点的结构体包含前驱指针、后继指针、孩子指针和数据域。以线性表的形式存储数据，每个数据存在一个节点的数据域中。一个节点有孩子的前提是它的任意相邻节点（前驱或者后继）的数据恰好比它的数据小一。遍历链表中的所有节点，找出所有有孩子的节点中数据域最大的一个节点，删除它并比较它的前后两个节点的数据域大小，若其中一个原本没有孩子节点在刚刚的删除操作后与新链接的节点的数据大小恰好满足删除条件，那么这个节点也就有了孩子节点。及时保存删除次数。在主函数中不断调用删除函数直至不能继续删除为止，输出最后的删除次数。

(2.1)

(2)

(2.1) 若 x_i、x_j 能够相邻则说明 $x_i \sim x_j$ 之间的所有元素都能在删除过程中符合被删除的要求而被删除。

设 x_i 的值为 W，下面首先证明，序列中若存在 $x_k = W + C = x_p$，且 $|k-p| > 1$，则 x_k、x_p 不可能在若干次操作后相邻。（其中 C 为一正整数），证明如下：

若存在这样的 x_k、x_p，则最终得到的相邻序列 (W+C)、(W+C) 可以经与插入操作逆向的插入操作还原为原来的 $x_k \sim x_p$ 序列

这些插入操作中的第一步只能在两元素中插入 (W+C+1)，得到 (W+C)、(W+C+1)、(W+C)

第二步只能选择在 (W+C) 和 (W+C+1) 中间插入 (W+C+1)，后面不可能再在 (W+C) 和 (W+C+1) 间插入。此时 x_k 和 x_p 中间所有元素都为不小于 (W+C+1) 的元素。下面用数学归纳法证明 n 次这样的插入操作后，只能得到 (W+C) (W+C) 的序列。

n个 大于(W+C)的元素

假设 (n-1) 次操作后得到的是

(W+C) (W+C) 的序列，那么再一次插

(n-1)个 大于(W+C)的元素

作时，序列中当前最小元素为 (W+C)，

故以

(W+C)(W+C) 为基础得到的插入结果必然为

(W+C)(W+C+1) … (W+C+1)(W+C)

此时最大可删元素不为 x_i。 n个大于(W+C)的元素.

2023

由此可见，2个(W+C)元素若初始时不为相邻元素，则决不可能通过删除操作相邻，否则此则违反了最大可删元素为 x_i 的条件。

那么要证明对于 $\forall x_j = x_i + 1$，$|j-i|>1$，x_i、x_j 不会经过若干次操作相邻，也可以采用相同的插入还原+反证法：

对于 W和(W+1)，第一次插入时只可能插入(W+1)或(W+2)，得到：

①W、(W+1)、(W+1) 或 ②W、(W+2)、(W+1)

以①或②为基础再进入n次插入，由于初始序列中最小元素大小为W，而每次插入的元素必大于等于当前最小元素加1，所以第n次插入的元素使得 x_i 不是当前最小可删除元素。最小可删除元素大于等于第n次插入的元素。($n=1,2,\cdots$)

综上，由反证法证得 对 $\forall x_j = x_i + 1$，$|j-i|>1$，无论接下来如何操作 x，x_i 和 x_j 者不可能相邻。

(2.2)

(2.3)

Q2

(1)

```
def max__sum(X, k):
    n = len(X)

    sum = sum(arr[:k])
    max_sum = sum

    for i in range(k, n):
        sum = sum - arr[i - k] + arr[i]
        # 更新最大和
        max_sum = max(max_sum, sum)
    max_sum=max_sum+k*(k-1)/2
    return max_sum

X = [...]
k = ...
result = max_sum(X, k)
print(result)
```

由于循环中，求和是从X1求到Xk,更新是i从（k+1）到n,所有最坏情况出现在k=n时，最坏时间复杂度为O(n)

(2)

(2) 证明算法正确性：

首先 a_i 为大于等于0的正整数。当 $k \le n$ 时，由于每天晚上每个宿舍的电脑数都 +1，把这些新增加的电脑数置于决策之外也不影响原问题的决策选择。可以理解为：第一次进入第 i 间宿舍，即获得 a_i 台电脑，该宿舍电脑数从此置0，再次访问时电脑数不增加。据此制定 k 天步骤，求出最大电脑数，并在此基础上加上 $\frac{k(k-1)}{2}$ 得到最终结果。

~~而每次决策选择有 种情况。~~

~~(1) 和~~ 在直线尽头（第1、n间）的一侧无宿舍，直接认为已访问置0.

每次决策时有3种情况：

① 当前所在宿舍左右都还没访问，那么选择较大者，为下一步最优。（这只在选第1间访问宿舍时）

② 当前所在宿舍一侧已访问，一侧未访问，那么显然访问临近宿舍中 a_i 较大者。

③ 当前所在宿舍两侧均已访问，则访问更久未访问的那一间。

而根据以上每一步的局部最优策略，~~在~~选择1间宿舍后总以走到没访问过的宿舍为最优，这也构成了最终的最优解，就是求出 X 中 k 中连续宿舍初始电脑数的最大和，再加 $\frac{k(k-1)}{2}$

而算法中的方法是先求 1~k 宿舍的最大和，再滑动窗口进行最大和的更新，得到最大和，再加 $\frac{k(k-1)}{2}$ 得到结果。

(3)

```
def max__sum1(X, k):
    n = len(X)

    sum = sum(arr[:n])
    max_sum=sum+k*(k-1)/2
    return max_sum

X = [...]
k = ...
result = max_sum1(X, k)
print(result)
```

Q3

(1)

```
Partition(A,p,r) //p、r为数组下标
      x = A[r]   //将最后一个元素作为主元素
      i = p-1 // i指向的是比主元素小的位置，
      for  j = p  to  r-1
          do  if  A[j] <= x
                        then  i = i+1
                              exchange A[i] <-> A[j]
      exchange A[i+1]<->A[r]
      return i+1
```

```
QuickSort(A,p,r)
       if p<r
            q = Partition(A,p,r)    //确定划分位置
            QuickSort(A,p,q-1)     //子数组A[p...q-1]
            QuickSort(Q,q+1,r)     //子数组A[q+1...r]
```

由f(y)分析知，要是f(y)取到最大值，则需要Y中大的元素尽量往后排，小的元素尽量往前排。那么我们调用快速排序相关的代码即可得到最佳y，时间复杂度为O(nlgn)

(2)

由f(y)分析知，f(y)相当于舍弃一部分元素，只保留其中的部分元素值，要求从X中选择k个元素（0=<k<=n), 求能够使得这（k个元素和-k(k+1)/2)最大的k个元素，然后在所有的k对应的最大值中找到最大值，即为所求。
首先同样的调用快速排序，复杂度O(nlogn),把元素按照从大到小排列。

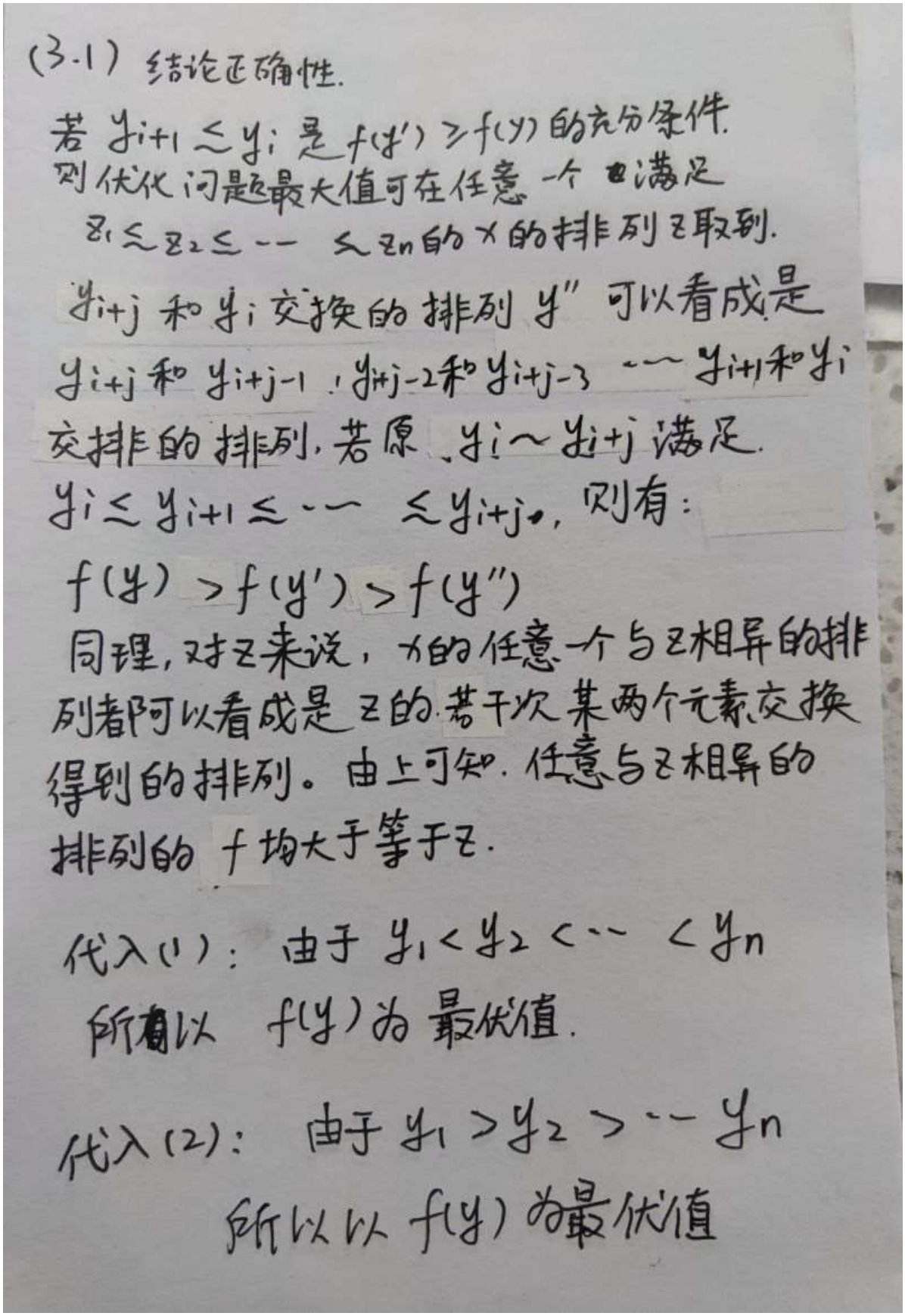
```
Partition(A,p,r) //p、r为数组下标
      x = A[r]   //将最后一个元素作为主元素
      i = p-1 // i指向的是比主元素小的位置，
      for  j = p  to  r-1
          do  if  A[j] >= x
                then  i = i+1
                    exchange A[i] <-> A[j]
      exchange A[i+1]<->A[r]
      return i+1
```

```
QuickSort(A,p,r)
       if p>r
            q = Partition(A,p,r)    //确定划分位置
            QuickSort(A,p,q-1)     //子数组A[p...q-1]
            QuickSort(Q,q+1,r)     //子数组A[q+1...r]
```

设S是得到的新序列,那么这个求和过程应该在S[i]<i+1的时候终止，因为这是再往里面加入新元素S[i]抵消不了被扣掉的(i+1)。

```
for i in range(len(S)):
    sum+=S[i]-i-1
    if S[i]<i+1:
        return sum
```

(3.1)


(3.1) 结论正确性.
若 $y_{i+1} \le y_i$ 是 $f(y') \ge f(y)$ 的充分条件.
则优化问题最大值可在任意一个满足
$z_1 \le z_2 \le \cdots \le z_n$ 的 x 的排列 z 取到.
y_{i+j} 和 y_i 交换的排列 y'' 可以看成是
y_{i+j} 和 y_{i+j-1}, y_{i+j-2} 和 y_{i+j-3} $\cdots$ y_{i+1} 和 y_i
交换的排列. 若原 $y_i \sim y_{i+j}$ 满足.
$y_i \le y_{i+1} \le \cdots \le y_{i+j}$, 则有:
$f(y) > f(y') > f(y'')$
同理, 对 z 来说, x 的任意一个与 z 相异的排列都可以看成是 z 的若干次某两个元素交换得到的排列。由上可知. 任意与 z 相异的排列的 f 均大于等于 z.
代入(1): 由于 $y_1 < y_2 < \cdots < y_n$
所以 $f(y)$ 为最优值.
代入(2): 由于 $y_1 > y_2 > \cdots y_n$
所以以 $f(y)$ 为最优值

(3.2)

Q4

(1.1-1.3)

Q4. (1.1)

举例：☆(())

正常：☆(()) → ☆() → ☆ → true

调换后：☆(()) → (() → (→ false.

此时判断错误.

(1.2)

举例：(☆((☆)

正常：(☆((☆) → (☆((☆ → (☆ → 空 → true

调换后：(☆((☆) → ☆((☆ → ☆(→ false

此时判断错误.

(1.3)

举例：(☆((☆☆

正常：(☆((☆☆ → (☆(☆ → (☆ → 空 → true

调换后：(☆((☆☆ → ☆ ((☆ → ☆ (→ false

此时判断错误.

(1.4)

用栈来实现

```
def checkValidString(s):
    left_stack = []
    star_stack = []

    # 从左到右扫描
    for i in range(len(s)):
        if s[i] == '(':
            left_stack.append(i)
        elif s[i] == '*':
            star_stack.append(i)
        elif s[i] == ')':
            if left_stack:
                left_stack.pop()
            elif star_stack:
                star_stack.pop()
            else:
                return False

    # 通过修改字符串还原删除后的字符串
    modified_s=list()
    COUNT=0
    while(left_stack && star_stack):
        index1=left_stack.pop()
        index2=star_stack.pop()
        if(index1>index2)
            modified_s[COUNT]='*'
            COUNT+=1
            left_stack.append(index1)
        if(index1>index2)
            modified_s[COUNT]='('
            COUNT+=1
            star_stack.append(index2)
    while left_stack:
        index1=left_stack.pop()
        modified_s[COUNT] = '('
        COUNT+=1

    while star_stack:
        index2=star_stack.pop()
        modified_s[COUNT] = '*'
        COUNT+=1

    # 从右到左扫描
    for i in range(len(modified_s) - 1, -1, -1):
        if modified_s[i] == '(':
            return false
            if modified_s[i] == ' ':
                modified_s[i] = ')'
            else:
                return False
```

```
    for i in range(len(modified_s) - 1, -1, -1):
        if modified_s[i] == '(':
            if(star_stack):
                star_stack.pop()
            else:
                return false
        elif modified_s[i] == '*':
            star_stack.append(i)
    return true

# 示例
s = .... #定义字符串s
result = checkValidString(s)
print(result)  # 输出 True或者false
```

算法解释：用栈来实现这一算法，从左到右扫描时用left_stack和star_stack分别存储了当前读取到的左括号、星号索引，遇到右括号就pop(优先pop左括号，左括号没有则pop星号，都没有则报错)。根据全部遍历完后栈中剩下的索引还原删除所有右括号及配对后剩下的字符串。再根据这一字符串从右到左扫描。从左到右、从右到左扫描时由于每个元素遍历一次，时间复杂度为O(n)。在还原剩余字符串时，由于每次都需要比较left_stack和star_stack栈顶的索引值大小，所以最坏时间复杂度是O(n^2),即原字符串中不存在右括号且左括号和星号交替出现时的情况。

(2)

(2) 算法正确性：

① 初始：原始字符串，要分类成
True（括号匹配正确）和false（匹配错误）
两种.

② 保持：在每一次操作前后，
原为true的在删除/修改元素后仍为
true，false同理。这也是要证明的
循环不变式

③ 终止：删除/修改后只剩"*"或为空，
判定为true；其它情况为false。

对于第一轮循环

① 当扫描到右括号时，若左侧有左括号，
两者一起删除，此操作不影响序列正确性，
因为若两者间有若干个星号，星号在正确序列
中有变空、变左括、变右括的可能作用。
而左、右括号一并删除后不影响星号在序列
中发挥的作用。

② 当扫描到右括号时，若左侧无左括号，
当前扫描过的部份右括号偏多，左侧只剩
星号或为空。若为空，则当前这一右括号不
可能有匹配的结果，输出false。若有星
号，则必有星号要变左括号与这一右括号
匹配。

由于星号之间等价且左侧无左括号或右括号，所以取左侧任一星号匹配消除即可。这一步骤不影响序列正确性。

对第二轮扫描，若从右到左有左括号，说明右侧无右括号匹配，输出false。扫描到星号时，由于序列中已无右括号，该星号又为当前最右侧星号，必然要变右括号与左侧括号匹配。消除前后不影响最右左括号与最右星号之间所有星号变左、变右、变空的影响效果。

过程中，若任意时刻字符串中既没有左括号，也没有右括号，则所有星号变空即满足true。

综上所述，算法过程中维持了循环不变式，即操作前后不影响序列正确性。