COMP 2123 Assignment

Problem 1

Solution:

a.

$$\begin{aligned} S &= 3 \\ f_1 &= 6, f_2 = 5, f_3 = 4; \\ s_1 &= 3, s_2 = 2, s_3 = 1; \end{aligned} \tag{1}$$

`function PickLargest`的计算结果应该是选取f_1，显然这不是最优的，因为如果选择f_2, f_3可以带来更大乐趣并且满足空间的条件. 这种简单的贪心算法是错误的.

b.

`function PickLargestRatio`是不正确的，反例如下:

$$\begin{aligned} S &= 8 \\ f_1 &= 15, f_2 = 4, f_3 = 4, f_4 = 4, f_5 = 4; \\ s_1 &= 7, s_2 = 2, s_3 = 2, s_4 = 2, s_5 = 2; \end{aligned} \tag{2}$$

根据算法，它应该先选择f_1，因为$\frac{f_1}{s_1} > 2$，但是显然选择f_2, f_3, f_4, f_5会带来更大的乐趣(等于$16 > f_1$).

Problem 2

Solution:

a.

构建两个容量为k的堆，堆里的每个元素是**指向双向链表的指针**(双向链表元素的地址)，一个是大根堆，一个是小根堆，注意他们的容量为k. 堆的实现采用树或者数组都可以，堆的建立(数据结构的建立)将带来$O(2k) < O(n)$的时间复杂度. 堆的插入更新的时间复杂度为$O(log\ k)$，再次强调因为这里两个堆的容量为k.

算法思路:

1. 建立堆: 不妨设k个排序好的双向链表为从小到大排序，算法在链表两端选取k个元素，每个链表选一个建堆，大根堆在值更大的那端，小根堆在值更小的那端.
2. 更新堆: 不妨设$d_j^{(i)}$为第j个链表的第i个元素. 则在`remove-min`之后，小根堆用$d_j^{(i+1)}$更新(插入)堆(即每次都是选择出堆元素位于链表位置的**后一个**，同理大根堆每次都选取链表的最大元素，更新使用$d_j^{(i-1)}$(链表的**前一个**)
3. 维护k个链表的k个最小值始终在堆中.

伪代码如下:

```
struct DataNode {
    DataNode *next, prev;
    Data data;
}
def ConstructHeap:
    Select the smallest element from each List: min_{0, ..., k - 1}
    Build a minimal heap with these k elements
    In a similar way, Build the maximum heap

```

```
def remove-min:
    ret = minimal_heap.pop()
    minimal_heap.insert(ret->next)
    return ret

def remove-max:
    ret = maximum_heap.pop()
    maximum_heap.insert(ret->prev)
    return ret
```

b.

证明思路：抓住每次更新堆都是用出堆元素的后一个(对于小根堆)/前一个(对于大根堆)，不妨以remove-min为例，remove-max同理可证：

- 初始步：remove-min弹出的第1个元素一定是最小的，因为当前的全局最小的元素一定是某个链表的最小元素，在建堆时会被选择.
- 迭代步：设当前remove-min弹出的元素为：$d_j^{(p)}$(第j个链表的第p个结点)，证明下一个remove-min的元素一定在堆中，此时$d_j^{(p+1)}$进堆，如果$d_j^{(p+1)}$就是下一个remove-min，成立. 若不然，其他链表的最小值结点都在堆中，所以可以保证下一次remove-min的元素在堆中.

c.

- 建立堆的时间：$O(2k) = O(n)$

$$\sum_{h=0}^{\lfloor \log k \rfloor} \left\lceil \frac{k}{2^{h+1}} \right\rceil O(h) = O(k) \tag{3}$$

- remove-min/remove-max: $O(log\ k)$ 更新堆，因为堆的容量是k.

Problem 3

Solution:

a.

算法思路：

不妨设两棵树大小分别为p, q，最左端的结点为m_1, n_1(最小值)，最右端的结点为m_p, n_q(最大值). 把中位数转化成寻找第$\frac{p+q}{2}$大的元素. 即比该结点大的元素有$\frac{p+q}{2}$个，比它小的有$\frac{p+q}{2}$个.

对任意结点i，记在这个结点的AVL树上 i的左子树个数为L_i，右子树个数为R_i. (输入给出的)

1. 找到两棵树的中位数a, b，因为AVL树左右子树平衡因子的绝对值不超过1，在AVL树中这两个结点就是两棵树的根节点.
2. 如果$a = b$，则输出a.
3. 若不然，不妨设$a > b$($a < b$同理可得). 记录a结点右子树个数R_a，b结点左子树个数L_b. 问题转化为在第一棵树的左子树和第二棵树的右子树上(即区间$[\ m_1, a\ ] \cup [\ b, n_1\ ]$)找结点$i \in tree\ A, j \in tree\ B,$，满足：

$$\begin{cases} R_i + R_a + R_j = L_i + L_j + L_b & (i.data = j.data) \\ R_i + R_a = L_j + L_b = \frac{p+q}{2} & (i.data \neq j.data) \end{cases} \tag{4}$$

伪代码如下：

```
def find-middle-element:
```

```
    # given tree A and B:
    a, b = A.root, B.root
    while a != b:
        Update(La_prev, Ra_prev, Lb_prev, Rb_prev)
        if a > b:
            if a->left != NULL:
                a = a->left
            if b->right != NULL:
                b = b->right
        else:
            if a->right != NULL:
                a = a->right
            if b->left != NULL:
                b = b->left
        Update(La, Ra, Lb, Rb)
        if Ra_prev + Ra + Rb_prev = La_prev + Lj + Lb_prev

     return a
```

b.

1. 初始步：如果A的根节点a等于B的根节点b，那么它就是中位数，从划分角度解释就是它带来了全局上相等的划分.
2. 迭代步：如果$a > b$，那么中位数一定落在$[\,m_1, a\,] \cup [\,b, n_1\,]$，即比$a$更小，比$b$更大，由上述迭代条件，每次二分舍去的区间都是不可能有中位数的区间. 递归地搜索，终止条件为：$R_i + R_a + R_j = L_i + L_j + L_b$，即比$i/j$大的元素和比它们小的元素个数相等，正确性得证.

c.

$$O(log\, n) \tag{5}$$

因为在两个AVL树上的搜索都是不回头的，只存在从上到下的搜索，所以最坏时间复杂度为$2\,O(log\,\frac{n}{2}) = O(log\, n)$.