HW2 hand-writting

Problem 1 - Sort

1.

```
procedure boundary_cake(P, from, to)
    set left to from
    set mid to from + 1
    set right to from + 2
    set next to from + 3
    while(next <= to + 1)
        set tmp to pancake-god-oracle(P, P[left], P[mid], P[right])
        if tmp == P[right]
            right = next
        else if tmp == [left]
            left = next
        else if tmp == [mid]
            mid = next
        end if
        set next to next + 1
    end while

    if right == next - 1
        return P[mid], P[left]
    else if mid == next - 1
        return P[right], P[left]
    else if left == next - 1
        return P[mid], P[right]}
    end if
end procedure
```

2.

```
procedure sort(P, front, end)
    if(P.size > 1)
        set mid to (front + end) / 2
        sort(P, front, mid)
        sort(P, mid + 1, end)
        set bound to boundary_cake(P, from, end)
        set left_index to front
        set right_index to mid + 1
        set tmp_index to 2
        set array(end - front)
        set array[1] to bound.first

        while(left_index != mid + 1 and right_index != end + 1)
```

```
            set tmp to pancake-god-oracle(P, P[left_index], P[right_index], bound.second)
            if tmp.second == P[left_index] or tmp.first == P[left_index]
                set array[tmp_index] to P[left_index]
                set left_index to left_index + 1
            else
                set array[tmp_index] to P[right_index]
                set right_index to right_index + 1
            end if
                tmp_index to tmp_index + 1
        end while

        while(left_index != mid)
            set arr[tmp_index] to P[left_index]
            set left_index to left_index + 1
            set tmp_index to tmp_index + 1
        end while

        while(right_index != end)
            set arr[tmp_index] to P[right_index]
            set right_index to right_index + 1
            set tmp_index to tmp_index + 1
        end while
    end if
end procedure
```

3.

```
procedure insert(P, ins, left, right)
    if P[left] == pancake-god-oracle(P, P[left],  P[right], ins)
        set mid to left
        set tmp to P[mid]
        set P[mid] to ins
        for i ( mid + 1 to right + 1) do
            set tmp2 to P[i]
            set P[i] to tmp
            set tmp to tmp2
        end for
        return
    else if P[right] == pancake-god-oracle(P, P[left],  P[right], ins)
        P[right + 1] = ins
        return
    end if

    set orig_right to right
    while left <= right
        set mid to (left + right) / 2
        if ins == pancake-god-oracle(P, P[right],  P[mid], ins)
```

```
            left = mid + 1
        else if ins == pancake-god-oracle(P, P[left],  P[mid], ins)
            right = mid
        else
                break
        end if
    end while

    set tmp to P[mid]
    set P[mid] to ins
    for i ( mid + 1 to orig_right + 1 ) do
        set tmp2 to P[i]
        set P[i] to tmp
        set tmp to tmp2
    end for

end procedure
```

4.
```
procedure sort(P)
    for i(2 to P.size()) do
        insert(P, P[i], 1, i - 1)
    end for
end procedure
```

5. Definition of $f(x) = \mathrm{o}(g(x))$ means $f(x) < cg(x)$ for any $x > x_0$

   So for any comparison sort method if we want to find $f(x) = o(nlog(n))$ means there is a comparison sort method is faster than $nlog(n)$

   but it is impossible.

   Prove:

   The n-permutation have $n!$ condition. So if we change one element to the right place at a time, the remain elements have $(n-1)!$ permutation, $\frac{n!}{2} \geq (n-1)!$ for $n >= 2$, so after an exchange, we can at least decrease a half of permutation condition, so we can get $2^T \geq n! \Rightarrow T \geq log_2 n!$, T is the total steps that we can at least finish sorting.

   $T \geq log_2 n! = log_2 n + log_2(n-1)...+log_2(1) \geq$

   $log(\frac{n}{2}) + log(\frac{n}{2}) + log(\frac{n}{2}).... + log(\frac{n}{2})$(total $\frac{n}{2}$ terms) $= \frac{n}{2} log \frac{n}{2}$

   $T \geq \frac{n}{2} log \frac{n}{2} = \Omega(nlog(n))$

   So it is impossible to find $T < cnlog(n)$ because $T >= nlog(n)$

   Q.E.D

6. when n = 1, it is true because there is only one element.

When n = 2, it swap the number when the P_r is the larger number, so it cause the decreasing permutation.

Hypothesis: Assume that n = k is true, which means ELF-SORT(P, 1, k) can sorted the an array under 1, 2, 3, ... , k elements in descending order.

Then when n = k + 1, $\triangle = floor(\frac{(k+1-1+1)}{3}) = floor(\frac{k+1}{3})$

1. it will call ELF-SORT(P, 1, k + 1 - $floor(\frac{k+1}{3})$) which has at most $\frac{2k}{3}+1$ elements that is no more than than k, so the first $\frac{2k}{3}+1$ elements in the array can be sorted by hypothesis, but at that time the last $\frac{k}{3}$ elements still not be sorted.
2. And it will call ELF-SORT(P, 1 + $floor(\frac{k+1}{3})$, k + 1) which has at most $\frac{2k}{3}+1$ elements, so by the hypothesis, the last $\frac{2k}{3}+1$ elements can be sorted, but at that time the middle $\frac{k}{3}$ elements are not sorted because it may contain the elements of the last $\frac{k}{3}$ in the 1 step.
3. In the end, it calls ELF-SORT(P, 1, k + 1 - $floor(\frac{k+1}{3})$) again, which will make at most the first $\frac{2k}{3}$ be sorted and make the middle $\frac{n}{3}$, elements be sorted, also, by hypothesis, it is correct.
4. After the three steps, the array has been sorted.

Q.E.D

7. The problem is divide into three sub-problems and each of the sub-problem has $\frac{2n}{3}$ scale, and in the final step, is $\mathbb{O}(1)$ because we may need the exchange when $P_r > P_l$, so we can get the recursive function $T(n) = 3T(\frac{2n}{3}) + \mathbb{O}(1)$

8.

$$T(n) = 3T(\frac{2n}{3}) + c = 3(3T(\frac{4n}{9}) + c) + c... = \Sigma_{i=0}^{log_{\frac{3}{2}} n} 3^i c$$

$$= 3^{\log_{\frac{3}{2}} n} + 3^{\log_{\frac{3}{2}} n-1} ... +1 = n^{log_{\frac{3}{2}} 3} + (n-1)^{log_{\frac{3}{2}} 3} ... +1 = \mathbb{O}(n^{log_{\frac{3}{2}} 3}) = \mathbb{O}(n^3)$$

Problem 2 - Tree

1.

```
procedure find-prev(T)
    if T.parent == NIL
        return NIL
    end if
    if T.left != NIL
        set tmp to T.left
        while tmp.right != NIL
            tmp = T.left
        end while
    end if
    set prev to tmp
    if T.parent.right == T
```

```
        if prev <= T.parent
            prev = T.parent
        else
            if T.parent.left == T
                if T.parent.parent.right == T.parent.parent
                    if prev <= T.parent.parent
                        prev = T.parent.parent
                    end if
                end if
            end if
        end if
    end if

    return prev
end procedure
```

2. BST has two nature:

   Given a node T,

   1. T.val < T_i.val $(T_i, T_i \in T.right)$,
   2. T.val > T_i.val $(T_i, T_i \in T.left)$
   3. $T \in T.parent$

   Prove:

   If we want to find the preview node, which means we have to find $max(T_i.val), (T_i.val < T.val)$, we have four conditions.

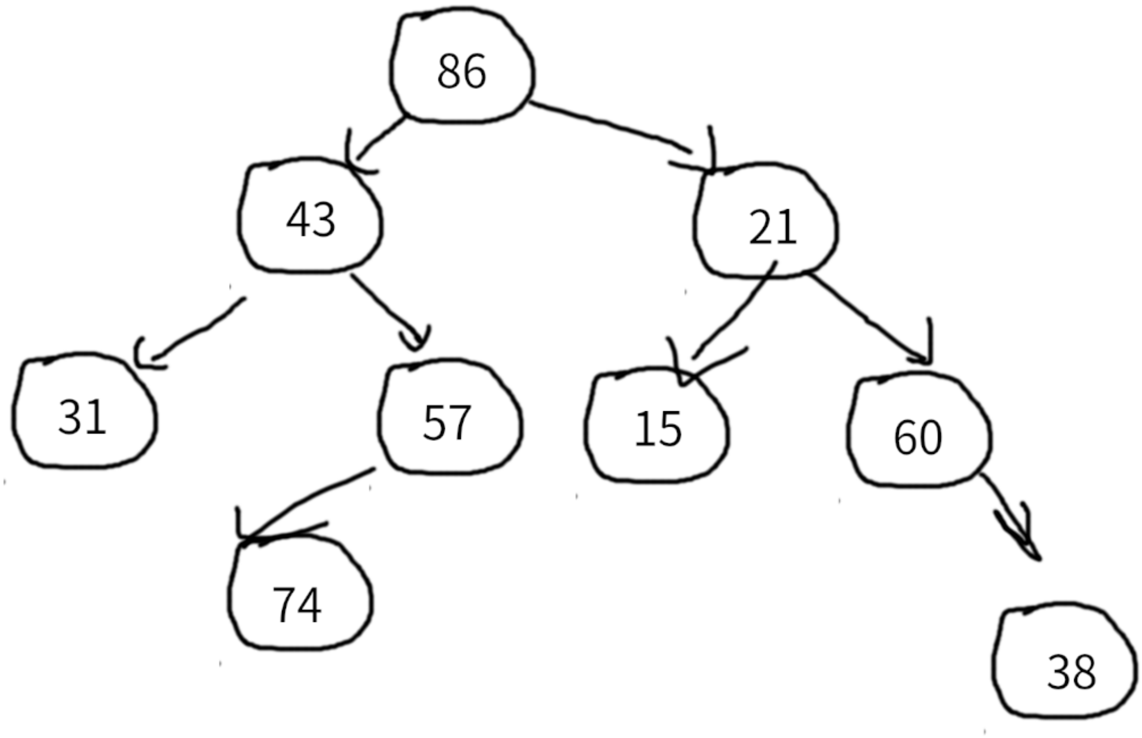
   - T.right: According to the first nature of BST, we couldn't find any node that it's value is smaller then T itself in T.right.
   - T.left: According to the second nature of BST, all the value in T.left is smaller than T. In these nodes of T.left subtree, we have to find the biggest number. So according to the first nature of BST, we have to find the rightest node in T.left
   - T is the right child of it's parent(T.parent.right == T): According to the first and second natures of BST, $T.parent.left < T.parent < T$ and according to the three natures of BST, T.parent.parent > T or T.parent.parent < T.parent < T, so the node in this branch must be T.parent.
   - T is the left child of it's parent(T.parent.left == T): According to the second nature of BST, T.parent > T, because the answer is not in T.parent, we have to extend to T.parent.parnet.

     According to the three natures of BST, T.parent.parent > T(No because prev_node < T) or T.parent.parent < T < T.parent, we only the second condition which means T.parent is the right child of T.parent.parent.

So T.parent.parent < T. According to the three natures of BST, T.parent.parent.left < T.parent.parent < T and T.parent.parent.parent > T > T.parent.parent or T.parent.parent.parent < T.parent.parent < T. So T.parent.parent.parent and it's parent must not be the answer.

The result in this brench is T.parent.parent while T.parent.parent exist and T.parent.parent.right = T.parent.parent.

  - Compare the one result of T.left and two result of T.parent, the answer must the biggest one.

3. 

4. No

   Prove:

   Inorder: left->mid->right

   Preorder: mid->left->right

   We can build a binary tree with three conditions

   - a node has two child: If we have only inorder array, we can create another tree by letting the the left-most array to the root, and other node is create by the order of mid->right->parent recursively. So if we have preorder, we can ensure the left and mid order. In advance, the right node is ensure. So we can create a unique permutation.
   - a node has one child: If we have only one child, if we have only preorder or inorder array, we will create two permutation because (NIL-parent-right_child) and (left_child-parent-NIL) will have the same preorder array, so we need inorder to ensure our only permutation
   - a node has no child need not be consider because there is only one node's value enter the inorder and preorder tree.

By consider three condition, we can create a unique binary tree.

5.
```
assume root is a tree
procedure buildTree(inorder, preorder)
    root.val = preorder[1]

    set middle to 0
    while inorder[i] != preorder[1]
        set middle to middle + 1
    end while

    root.left = buildTree(inorder[0 : middle - 1], preorder[2:
preorder.length()])
    root.right = buildTree(inorder[middle + 1: inorder.length()],
preorder[2: preorder.length()])

    return root
end procedure
```

Problem 3 - Heap

> Reference for how to write array in pseudo code:

1.
```
procedure modify(x, v)
    x.val = v
    if x.val > v
        while x.left != NIL and x.right != NIL and x.val  <
max(x.left.val, x.right.val)
            if max(x.left.val, x.right.val) == x.left.val
                swap(x.val, x.left.val)
                x = x.left
            else if max(x.left.val, x.right.val) == x.right.val
                swap(x.val, x.right.val)
                x = x.right
            end if
        end while
    else if x.val < v
        while x.parent != NIL and x.parent.val < x.val
            swap(x.parent.val, x.val)
            x = x.parent
        end while
    end if
end procedure

procedure delete(x)
    set min to extractMin()
    modify(x, min.val - 1)
    exractMin()
```

```
    insert(min)
end procedure
```

Prove:

1. Modify: In the two branch of if

   We only change the x node to it's parent or it's child, which means the number we at most need to traverse is it's height

   So in the two branch it will meet the requirement of $\mathbb{O}$(height-of-heap) $\leq \mathbb{O}(\lg h)$ (by consider it is a normal binary heap(balanced))

   Q.E.D

2. delete: by the problem describe, it cost $\mathbb{O}(lgh) + \mathbb{O}(lgh) + \mathbb{O}(lgh) + \mathbb{O}(lgh) = \mathbb{O}(lgh)$

   Q.E.D

2.

   1.

NAn	NAn	NAn	NAn
NAn	NAn	NAn	NAn
NAn	NAn	NAn	1
4	NAn	NAn	2

   2.

NAn	NAn	NAn	NAn
NAn	NAn	NAn	NAn
NAn	NAn	NAn	1
4	NAn	NAn	NAn

   3.

NAn	NAn	NAn	NAn
NAn	NAn	NAn	3
NAn	NAn	NAn	1
4	NAn	NAn	NAn

   4.

NAn	NAn	NAn	NAn
NAn	NAn	NAn	3
NAn	NAn	NAn	NAn
4	NAn	NAn	NAn

5.

NAn	NAn	NAn	NAn
NAn	NAn	NAn	NAn
NAn	NAn	NAn	NAn
4	NAn	NAn	NAn

3. The heap in this problem store the index of the array and therefore the top value is the smallest value's index in A instead of the smallest value.

```
assume row_heap is a heap(N)
assume col_heap is a heap(M)

struct D
    row_heap
    col_heap
end struct

procedure add(i, j, v)
    set A[i][j] to v
    row_heap[i].insert(j)
    col_heap[j].insert(i)
end procedure

procedure extractMinRow(i)
    set index to row_heap[i].extractMin()
    set A[i][index] to 0
    col_heap[index].delete(i)
end procdure

procedure extractMinCol(j)
    set index to col_heap[j].extractMin()
    set A[index][j] to 0
    row_heap[index].delete(j)
end procedure

procedure delete(i, j)
    set A[i][j] to 0
    col_heap[i].delete(j)
```

```
    row_heap[j].delete(i)
end procedure
```

4. - add: insert in a heap cost $\mathbb{O}(lg(x))$ which x is the element numbers of the heap . So the total element in the row_heap[i] will not exceed N because it store a column of a 2D array. And similarly, the total element in the col_heap[j] will not exceed M, so the total time cost is
   $\mathbb{O}(1) + \mathbb{O}(lg(N)) + \mathbb{O}(lg(M)) = \mathbb{O}(lg(MN))$
   Q.E.D
   - extractMinRow: according the description on the top of the problem, delete a node and extractMin in a heap is $\mathbb{O}(lg(x))$ which x is the element numbers in the heap. So the total time cost is $\mathbb{O}(N) + \mathbb{O}(1) + \mathbb{O}(M) = \mathbb{O}(lg(MN))$

     Q.E.D
   - extractMinCol: according to the description on the top of the problem, delete a node and extractMin in a heap is $\mathbb{O}(lg(x))$ which x is the element numbers in the heap. So the total time cost is $\mathbb{O}(M) + \mathbb{O}(1) + \mathbb{O}(N) = \mathbb{O}(lg(MN))$
   - delete: according to the description on the top of the problem, delete a node in a heap is $\mathbb{O}(lg(x))$ which x is the element numbers in the heap. So the total time cost is $\mathbb{O}(1) + \mathbb{O}(N) + \mathbb{O}(M) = \mathbb{O}(lg(MN))$