FA HW 5

Fundamental Algorithms
HW5

Problem 1.

1.(a)

```
Multiway-Merge (A_i, _, A_j, n_i, _, n_j)
{

# R ≡ (B_r, m_r) is a set of arrays produced
  in the main loop.

   if (j - i + 1 == 2) :
        return Merge (A_i, n_i, A_j, n_j)

   else :
        y = 1
        x = i
        while (x <= j-1) {

      (B_y, m_y) = Merge (A_x, n_x, A_x+1, n_x+1)
```

```
        x = x + 2
        y = y + 1

    }
    if (x == j):
        (B_y, m_y) = (A_j, n_j)


  return Multiway-Merge (B_1, m_1, — B_y, m_y)

}
```

- We have Arrays $A_0, \ldots, A_{k-1}$
- We merge 2 arrays at a time in one pass and reduce the number of arrays to merge by half. We take into consideration the odd and even number of arrays cases.
- We recursively perform this procedure on the new set of arrays.
- Finally, we merge the last two Arrays.
- We will have $\log_2 k$ such calls to merge.

```
Main () {
```

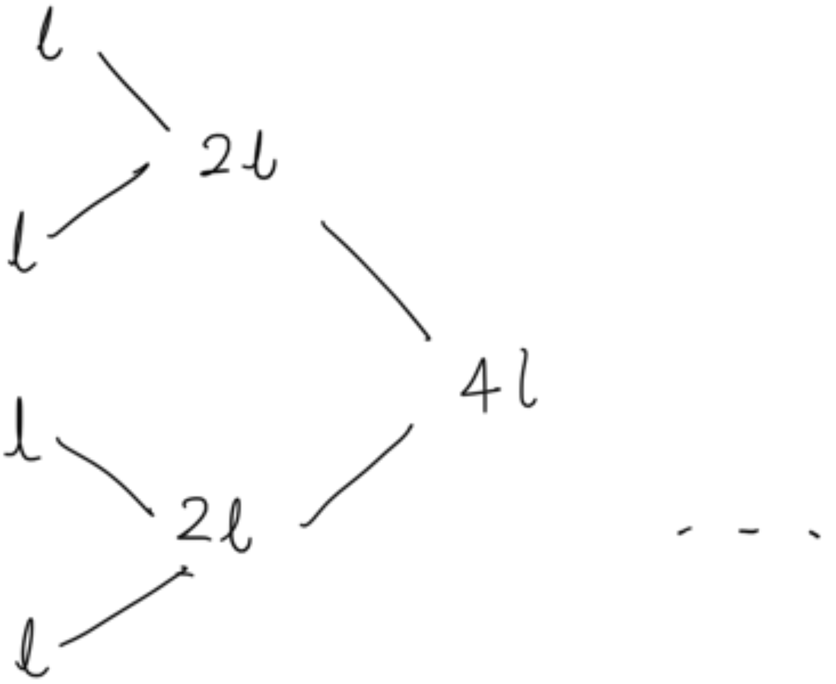
Initialize Arrays $((A_0, n_0), (A_1, n_1), — (A_{k-1}, n_{k-1}))$
where array A_i has size n_i.

A_{final} = Multiway_Merge $(A_0, - A_{k-1}, n_0 - n_{k-1})$
return A_{final}
}

1.(b)

length of A_i - l
$n = kl$

We represent a tree for the merge procedure calls.



The first set of merge calls will be $\Theta(l+l)$
i.e. $\Theta(2l)$.

There will be $\frac{k}{2}$ such calls.

$\therefore$ The first call to Multiway merge will be of time complexity,

$$\frac{k}{2}\,\theta(2l) \equiv \theta(kl)$$

Using the definition of the theta notation,

$$c_1 2l \leq \theta(2l) \leq c_2\, 2l$$

Multiplying by $k/2$,

$$c_1\, kl \leq \frac{k}{2}\,\theta(2l) \leq c_2\, kl$$
$$\equiv \theta(kl)$$

The 2^{nd} set of calls to the merge procedure will be $\theta(2l + 2l) \equiv \theta(4l)$ each. At each step, the breadth of the tree is reduced by half, there will be $\frac{k}{4}$ such calls,

$\therefore$ The 2^{nd} call to the multiway merge procedure will be $\frac{k}{4}\cdot\theta(4l) \equiv \theta(kl)$ as we have seen before.

- At each call to the multiway merge procedure, the number of arrays to be merged is halved, at the i^{th} call to the merge procedure, we will have k arrays to be merged. and will have $\frac{k}{2^i}$ arrays to be merged and each array will have the size $2^i l$.

- For the corresponding tree representing calls, there will be $(\log_2 k)$ height.

$$\therefore T(k) = \left[\Theta(kl) + \Theta(kl) \ldots\right] \Theta(\log_2 k) \text{ times}$$

$$\therefore T(k) = \Theta(kl \log k)$$

For the recurrence relation for our multiway merge implementation,

$$T(k) = T(k/2) + \Theta(kl)$$

$$T(k/2) = T(k/4) + \Theta(kl)$$

term does not change with k and will always be $\Theta(kl)$

$$T(k/4) = T(k/8) + \Theta(kl)$$

$$\vdots$$

$$T(2) = T(1) + \Theta(kl) \left(k \quad 2^{\log_2 k - 1} l \right)$$

$T(2) = 2T(1) + O(kl) \quad \left(\frac{k}{2^{\log_2 k - 1}} = 2 \right)$

\# Merge two lists of size $\frac{kl}{2}$

$T(1) = O(1)$

\# already merged single list.

Adding both sides for all equations,

$$T(k) = \Theta(kl \log k)$$

Problem 1 (c).

We have two sorted arrays of size $n/2$ which constitute the final array A.

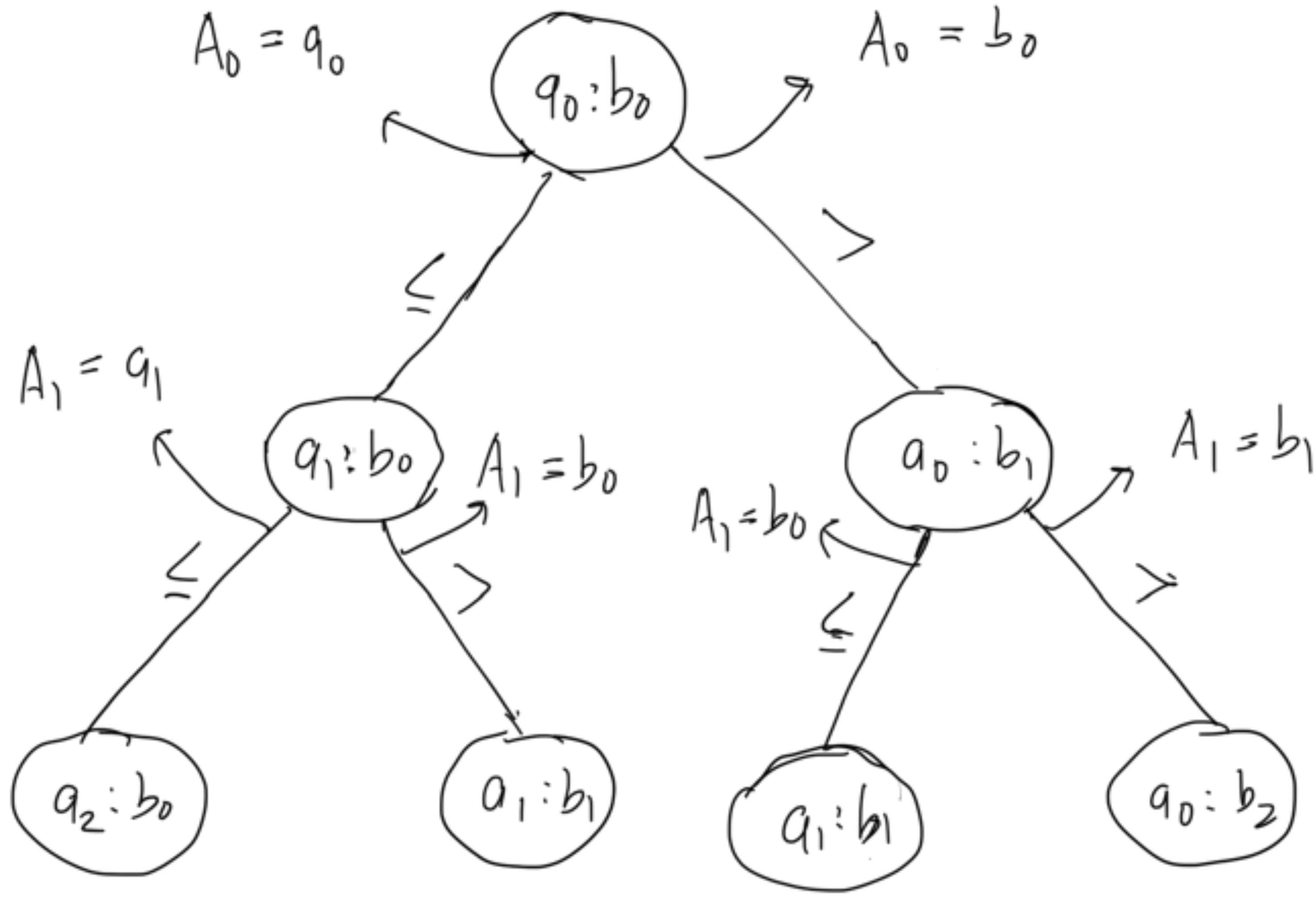
$A_0 : [a_0, \_\_, a_{n/2}]$

$A_1 : [b_0, \_\_, b_{n/2}]$

- In the merge procedure, we have one iterator for A_0 and one for A_1, we compare the first two elements a_0 and b_0, and select the least of the two elements for our final array and move the iterator forward for that element.
- The comparisions done in the merge procedure can

be represented in the form of a decision tree.

- Consider the decision tree,



⋮

- At every level of the decision tree, we will select a corresponding element for the final array A.
- Every branch of the decision tree will correspond to the running of the Merge procedure for some $a_0, \_\_, a_{n/2}$ and $b_0, \_\_, a_{n/2}$

which are sorted sub sets of A.

- The number of ways in which we can choose $n/2$ elements from the array A of $n/2$ elements, 13 $\quad {}^nC_{n/2}$

- For each such resulting A_0, A_1, our decision tree will yield a unique branch in the process of building up to A.
- There fore the number of leaves in our decision tree will be ${}^nC_{n/2}$.

- For every unique combination of A_0, A_1, the order of comparisions in the merge procedure must be unique.
- Suppose the order of comparisions for two different combinations is the same.

$A_0^1: a_1^1, \_\_ \; a_{n/2}^1$
$A_1^1: b_1^1, \_\_ \; b_{n/2}^1$

$A_0^2: a_1^2, \_\_ \; a_{n/2}^2$
$A_1^2: b_1^2, \_\_, b_{n/2}^2$

In the selection of C_k, let us compare a_i and b_j.

if $a_i^1 < b_j^1$ then $C_k = a_i^1$

This implies, $a_i^2 < b_j^2$ and $C_k = a_i^2$

next we compare a_{i+1}^1 and b_j^1, based on our assumption, the comparisions in order are the same,

$a_{i+1}^1 = a_{i+1}^2$

$b^{1}_{i+1} = b^{2}_{i+1}$, for the selection of element C_{k+1} for the sorting order to be preserved.

This will be valid for any k, i and j.

This would mean that A^1_0, A^1_1 and A^2_0, A^2_1 are equivalent and are the result of the same grouping on A. This completes our proof by contradiction.

Every branch in the decision tree will correspond to a unique order of comparisions and thus a unique A_0, A_1 derived from A.

\# A.size = n

$$N(\text{leaves}) = {}^{n}C_{n/2}$$

Using Stirling's approximation,

$$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^{n}\left(1 + O\left(\frac{1}{n}\right)\right)$$

$$N(\text{leaves}) = \frac{n!}{\left(\frac{n}{2}\right)!\left(\frac{n}{2}\right)!} = \frac{n!}{\left(\left(\frac{n}{2}\right)!\right)^{2}}$$

$$\approx \frac{\sqrt{2\pi n}\;\frac{n}{e^{n}}}{\left(\sqrt{2\pi\frac{n}{2}}\;\frac{\left(\frac{n}{2}\right)^{n/2}}{e^{n/2}}\right)^{2}}$$

$$\approx \frac{\sqrt{2\pi n}\cdot\frac{n^{n}}{e^{n}}\cdot 2^{n}}{2\pi\frac{n}{2}\cdot\frac{n^{n}}{e^{n}}}$$

$$N(\text{leaves}) \approx \left(\frac{2}{\pi n}\right)^{1/2}\cdot 2^{n}$$

Following from Sterlings approximation,

$$N(\text{comparisions}) \equiv \text{Height of decision tree}$$

$$\equiv \log_2\left(2^{n}\cdot\left(\frac{2}{\pi n}\right)^{1/2}\right)$$

$$N(\text{comparisions}) \geq n - \frac{1}{2}\log\left(\pi n/2\right)$$

$$\log\left(\frac{\pi n}{2}\right) = \sigma(n)$$

This will act as a lower bound for the number of comparisions.

1.(d).

$$A = [a_1, \_\_, a_{n/2}]$$

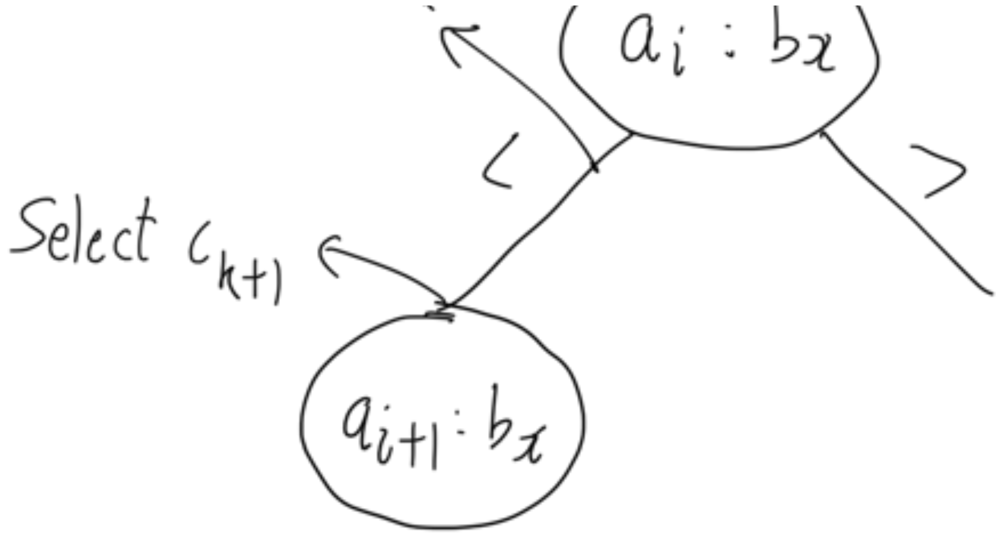
$$B = [b_1, \_\_, b_{n/2}]$$

- We have seen that every node in the decision tree will give us a corresponding element for the final array.
  let two consecutive elements in final array C be c_k and c_{k+1}.

- Mapping to our sub arrays, let c_k correspond to a_i and b_j correspond to c_{k+1}.

- For our level k where we select c_k, let the decision node be the following, with the comparision with some arbitrary b_x

Select $c_k = a_i$



We know that c_{k+1} is selected from B as it corresponds to b_j.
$\therefore$ At the decision node where we select for c_{k+1}, we must select an element from B which is b_x. Hence, b_x equals b_j.
We have compared a_i and b_x in the previous node.
$\therefore$ Our path will compare a_i and b_j for some a_i and b_j such that a_i and b_j are consecutive in C with $a_i < b_j$.

Consider the arrangements of A and B in C.
- For the case where we have alternating a's and b's such that $a_i < b_j$, as we traverse the decision tree and select an element at a time, we will have $(n-1)$ comparisions.
- If a_i is selected at some level k, b_j is selected at level $(k+1)$ on comparision with a_{i+1}. a_{i+1} is selected next on comparision with b_{j+1}.
- The process follows as k is going from 1 to $(n-1)$.
- If Array A has consecutive elements in C which are between two elements in B. These elements

will be compared to the same element in B as we move the iterator over A. If the rest of the configuration remains as discussed before, A will be exhausted first and the last few elements of B will be compared with infinity.

- Importantly, each comparision in the decision tree yields an element in the array C and C has n elements.
- After $(n-1)$ comparisions in the decision tree, we obtain $(n-1)$ corresponding elements for C. The positioning of the last element is trivial given the size.

$$\therefore N(\text{comparisions}) = (n-1)$$

1.(e).

We assume that for the multi way merge procedure for k sorted lists, time taken is $O(n \log \log k)$

Consider the sorting problem as follows, we divide the array of n elements into n/k arrays of k elements each.

We then sort each of the arrays consisting of k elements and merge the sorted n/k lists.

For our sorting procedure, we get,

$$T(n) = \frac{n}{k} T(k) + O\left(n \cdot \log \log \frac{n}{k}\right)$$

For some sufficiently small k, the complexity analysis with respect to n becomes,

$$T(n) = O(n \log \log n)$$

But we know that for comparision sorting,

$$T(n) = \Omega(n \log n)$$

But as $n \log n \geq n \log \log n$, therefore our complexity analysis breaks down. Hence our assumption is false.

$$T(n) \geq c_1 n \log n \quad \forall\ n > n_0$$

$$T(n) \leq c_2\, n \log \log n \quad \forall\ n > n_0^*$$

These are mutually inconsistent statements.

Hence, the claim of merging k lists with time complexity $O(n \log \log k)$ is disproved.

Problem 2.

2.a)

1. We use the recurisive partition based Select (A, i) procedure to select the i^{th} smallest element in $O(n)$ time. $i = \lceil n/2 \rceil$ and we find the median first.

2. We used the procedure to count the number of occurences of every unique element in our Array A. We can use an array C which stores the counts of elements in A. We consider the number of elements in A (n). The maximum M of those will define the range of $C[1, \_, M]$, if $M >> n$, we could use a Hash map C instead of an array. The procedure to develop the mapping will require one pass over array A and will be linear in n.

3. We use the 3-way Partition function to Partition around the median as pivot.

4. For the elements on the left of the pivot elements, we construct a max heap

from the unique ones. We find unique elements in linear time.

5. For the elements on the right of the pivot elements, we construct a min heap.

6. We maintain the record of the median, the range of positions which the median occupies and the size of A.

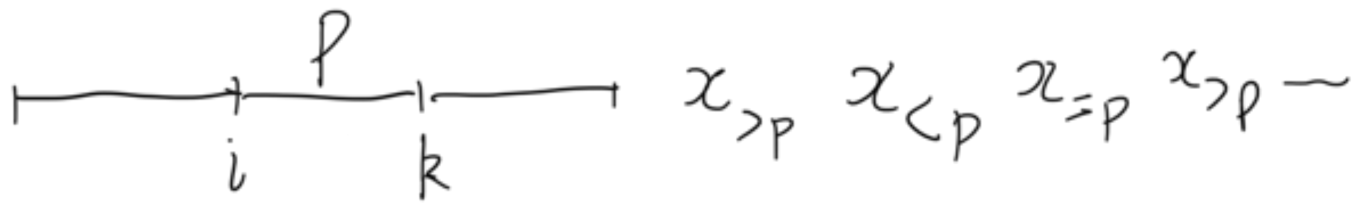
2.(b).

3-Way-Partition (A, p, r)

```
{
    x = A[r]
    i = p-1
    k = p-1

    for j from p to r-1 :

        if A[j] < x :
            i = i+1
            k = k+1
            exchange (A[j], A[i])
            exchange (A[i], A[k])
```

```
exchange (A[j], A[k])

if A[j] == x:
    k = k+1
    exchange (A[j], A[k])


exchange (A[k+1], A[r])

return i+1, k+1

}
```

- We have two iterators i and k which mark the boundary between elements smaller than the pivot elements and the pivot elements, the pivot elements and the elements larger than the pivot elements.



- If we encounter an element less than the pivot, as we iterate through the array (A[j]), we increment i, increment k, swap A[j] and A[i], then swap A[j] and A[k].
- Hence, when we encounter $x_{<p}$, i will move to the first pivot element p, k will move

to the first pivot element p_1, k will move to the first element ahead of the pivot elements, let that be l_1, we swap $x_{<p}$ ($A[j]$) and p_1. ($A[i]$).

- p_1 will now be in $A[j]$, we then swap $A[k]$ (l_1) and $A[j]$ (p_1), in order to put p_1 in the correct position with the pivot elements P. $x_{<p}$ is at a position where it is behind the pivot elements and elements like $x_{>p}$ are ahead of the pivot elements.
- if we encounter an element equal to the pivot, increment k and swap that element with $A[k]$.

- Elements from p to i (inclusive) will be less than the pivot. Elements from $(k+1)$ to r will be greater than the pivot. $(i+1)$ to k will contain the pivot elements. Hence we return $(i+1)$ and $(k+1)$.
- We do this over one pass over $A(p, \_, r)$. This is linear in the size of A.

$$T(r-p+1) \equiv \Theta(r-p+1)$$

$$T(n) = \Theta(n)$$

2. (c)

- For the analysis of time complexity with respect to n, the max_size of an element is a constant.
- We will assume that the range of elements in our array is proportional to the number of elements in our array A as we prepare an array to store the counts in linear time. Else we could use the dictionary data structure based on Hash Tables or binary search trees.

Build (A[1, __, n])
{
 # Initializations
 - M = Max_Size
 - Count [1, __, M], elements initialized to 0. Where M ≡ order of n
 - L - array from [1, __, n/2] for left heap
 - R - array from [1, __, n/2] for right heap.
 - Placed [1, __, M] array of booleans used for unique element identification.

 1. median = Select (A, i = $\lceil n/2 \rceil$)

 # O(n) time

2. I, J = 3_Way_ Partition (A, 1, n , pivot = median)

3. for i from 1 to n :

count [A[i]] = count [A[i]] + 1

O(n) time for this loop , as array indexing is constant time due to pointer arithmetic.

4. # find unique elements for max heap on the left.
k = 0
for i from 1 to I-1 :
if placed (A[i]] == False :
placed [A[i]] = True
k = k+1
L[k] = A[i]

5. # find unique elements for min heap on the right.
j=0
for i from J to n :
if placed [A[j]] == False :
placed [A [j]] = True
j = j + 1

$j = j+1$
$R[j] = A[i]$

6. H_1 = Build_Max_Heap (L [1, _ , j])

H_2 = Build_Min_Heap (R [1, _ , k])

start = I
end = J - 1
}

- Each of the steps in the algorithm is linear in n. The same would be the case if count was based on a hash table based dictionary. We would iterate over the keys to find the unique elements.

- The Build Heap procedures we know are linear in n. We have seen the partition procedure used to be linear in n, so is the select procedure for finding the median.

- We maintain variables median, count, start, end, H_1, H_2 for other procedures.

For Build (A [1, _ , n])

$T(n) = O(n)$

$\Rightarrow \quad T(n) = O(n)$

$\#$ if $\quad f(n) = O(n) \rightarrow f(n) \leq c_1 n \quad \forall n > n_0$

$\quad\quad g(n) = O(n) \rightarrow g(n) \leq c_2 n \quad \forall n > n_*$

let $n_* > n_0$

$\therefore \quad f(n) + g(n) \leq c_1 n + c_2 n \quad \forall n > n_*$

$\therefore \quad f(n) + g(n) = O(n)$

2.d

```
Insert (x)
{

  if x == median :
     count [median] = count [median] + 1
     n = n+1
     end = end +1

  if (x < median) :

     if (count [x] == 0) :
        Insert_Max_Heap ( R , x)
```

```
count[x] = count[x] + 1

n = n+1

if (start ≤ ⌈n/2⌉ and end ≥ ⌈end/2⌉):

        # for else case which corresponds to
          new median
else :

    new_median = Extract_Max(H1)

    end = start - 1
    start = end - count[new_median] + 1

    Insert_Min_Heap(H2, Median)

    median = new_median

if (x > median):

    if (count[x] == 0):
          Insert_Min_Heap(x)

    count[x] = count[x] + 1
```

$n = n + 1$

if $\left(start \le \lceil n/2 \rceil \text{ and } end \ge \lceil n/2 \rceil \right)$:

else :

new_median = Extract_Min(H_1)

start = end + 1

end = start + count[new_median] − 1

Insert_Max_Heap (H_1, median)

median = new_median

}

- We have a current median, it's start and end positions.
- After every insertion, n becomes $(n+1)$, we check if our current median occupies $\lceil (n+1)/2 \rceil$
- If an element smaller than the median is inserted, the original median element might shift right, and the new median will be the one just less than that, as extracted

from the max heap for the left sub array.

- Our condition checks for that and finds the beginning and the end the new median.
- Our old median is now greater than the new median and will be inserted in the heap on the right.
- Similarly for $x >$ median.

Extractions and Inertions in the heap have $O(\log n)$ time, rest of the individual operations taking constant time, for our Insert procedure,

$$T(n) = O(\log n)$$

2.e

```
Median ( )
{
    return median
}
```

We have always maintained the correct median during our Build and Insert procedures.

Problem 3.

3(a)

- Given that all elements are integer numbers from 1 to 10^6.
- We use the counting procedure to count the number of instances for each element.
- We then use the cumulative count to find the range.
- In the build procedure, we find the median along with developing the Count array. This is done in one pass over A.
- In the insert procedure, we increment the count, we run a loop over all unique elements in order to update the cumulative counts. We then update the median as suitable.
- Complexity analysis is relative to the size of the Array n, 10^6 here is independent and can be treated as a constant. As $A[1 \ldots n]$ grows arbitrarily large, the number of distinct elements is still 1 to 10^6.

We use an array $[1 \ldots 10^6]$ for the counts. We store the median after building and after every insertion.

3.(b)

Build (A[1, —, n])

{ median is stored in the variable median.
Count [1, —, 10^6] initialized to 0.
C_count [1, —, 10^6] initialized to 0.

for i from 1 to n :

Count [A[i]] = Count [A[i]] + 1

C_count [1] = Count [1]

for j from 2 to 10^6 :
C_count [j] = C_count [j-1] + count [j]

let c_count [0] = 0

for k from 1 to 10^6 :

if (c_count [k-1] + 1 $\leq \lceil n/2 \rceil$

and

$c\_count[k] \geq \lceil n/2 \rceil)$:

median = k

break

}

Insert (x)

{

count [x] = count [x] + 1

for i from x to 10^6 :

c count [i] = c count [i] + 1

n = n + 1

for i from 1 to 10^6 :

if ($c\_count[i-1] \leq \lceil n/2 \rceil$ and

$c\_count[i] \geq \lceil n/2 \rceil)$:

median = i

break

3.(c)

```
Median () {
    return median
}
```