FA Assignment 4

Q1.(a).

(i) $A = [5, 6, 4, 1, 9, 10]$

We do Reverse_Partition $(A, 1, 5, 6)$

$$A \rightarrow \begin{array}{cccccc} 5 & 6 & 4 & 1 & 9 & 10 \\ & & & & i & j \end{array}$$

$$\begin{array}{cccccc} 5 & 6 & 4 & 1 & 10 & 9 \\ & & & i & j & \end{array}$$

$$\begin{array}{cccccc} 5 & 6 & 4 & 10 & 1 & 9 \\ & & i & j & & \end{array}$$

$$\begin{array}{cccccc} 5 & 6 & 10 & 4 & 1 & 9 \\ & i & j & & & \end{array}$$

$$\begin{array}{cccccc} 5 & 10 & 6 & 4 & 1 & 9 \\ i & j & & & & \end{array}$$

$$\begin{array}{cccccc} 10 & 5 & 6 & 4 & 1 & 9 \end{array}$$

$A' = [10, 5, 6, 4, 1, 9]$

(ii) $A = [5, 6, 4, 1, 9, 10, 11, 12, 13]$

Reverse_Partition $(A, 1, 5, 9)$

$$A' \rightarrow \begin{array}{ccccccccc} 5 & 6 & 4 & 1 & 9 & 10 & 11 & 12 & 13 \\ p & & & & q & & & & r \end{array}$$

$$\rightarrow \begin{array}{ccccccccc} 5 & 6 & 4 & 1 & 13 & 10 & 11 & 12 & 9 \end{array}$$

$\rightarrow$ 5 6 4 12 13 10 11 1 9

$\rightarrow$ 5 6 11 12 13 10 4 1 9

$\rightarrow$ 5 10 11 12 13 6 4 1 9

$A' \rightarrow [13, 10, 11, 12, 5, 6, 4, 1, 9]$

(iii)

$$A = [5, 6, 4, 1, 9, 10, 11, 12, 13, 14]$$

p q r

$A' \rightarrow [5\ 6\ 4\ 1\ 14\ 10\ 11\ 12\ 13\ 9]$

$A' \rightarrow [5\ 6\ 4\ 13\ 14\ 10\ 11\ 12\ 1\ 9]$

⋮

$A' \rightarrow [14, 13, 12, 11, 5, 6, 4, 1, 9]$

Q1. b.

$A[1, —, n]$ is a sorted array.
$A(1, \lceil n/2 \rceil, n)$ can be used as an argument to the Reverse partition function as the sorted elements are partitioned around $\lceil n/2 \rceil$ with elements on the left being less than $\lceil n/2 \rceil$ and elements on the right being greater than $\lceil n/2 \rceil$. $\lceil n/2 \rceil$ here is the median.

if n is odd. we get the following structure,
Let,

$$A \rightarrow a_1 \text{ — } a_x \; \underset{\substack{| \\ \text{pivot}}}{\underline{b}} \; a_1' \text{ — } a_x'$$

Now, using the reverse partition function, we get, in place,

$$A' \rightarrow a_x' \; a_1' \text{ — } a_{x-1}' \; a_1 \text{ — } a_x \; b$$

if n is even,

$$A \rightarrow a_1 \text{ — } a_{x-1} \; b \; a_1' \text{ — } a_x'$$

after reverse partition, in place,

$$A' \rightarrow a_1' \text{ — } a_x' \; a_1 \text{ — } a_{x-1} \; b$$

Such a structure for A can be verified based on the application of the reverse partition function on our original array A.
If an array A is sorted and n is even,
The sub-arrays $a_1', \text{—}, a_x'$ and $a_1, \text{—}, a_{x-1}$ are sorted individually, this follows from our original A.
If n is odd, we can use rotation sorting for the sub array $a_x' \; a_1' \text{ — } a_{x-1}'$ in order to $a_1' \; a_2' \text{ — } a_{x-1}' \; a_x'$ in linear time.

Following this, for any sorted array A, after reverse partition and rotation adjustment, we obtain array $A_1 \; A_2 \; b$ in place, such that A_1 and

A_2 are sorted and b is the median of A.

For quick sort to give a balanced partition, the pivot must be the median. If we consider the pivot to be the last element for the partition function, $A_1 A_2 b$ satisfies the criterion. After the first call to the partition function, the next two calls will be on sorted arrays A_1, A_2 where we repeat the procedure recursively. We will prove this by induction with appropriate base cases.

```
Fast_QS (A, start, end)
{
    if ((end - start + 1) <= 2) {
        return
    }

    if ((end - start + 1) % 2 == 0) {

    Reverse_Partition (A, start, (start + end - 1)/2 , end)

    Fast_QS (A, start, (start + end - 1)/2 )

    Fast_QS (A, (start + end + 1)/2 , end - 1)

    }

    if ((end - start + 1) % 2 != 0) {
```

```
Reverse_Partition ( A, start, ⌈(start + end - 1)/2⌉, end)

Rotation_adjust ( A, start, ⌊(start + end - 1)/2⌋)

Fast_QS ( A, start, ⌊(start + end - 1)/2⌋)

Fast_QS ( A, ⌈(start + end - 1)/2⌉, end - 1)

}

}

Rotation_adjust (A, start, end)
{
    k = A[start]
    for i from start+1 to end :
        A[i-1] = A[i]

    A[end] = k
}
```

Proof:

Base case:

for $n = 2$, there are two elements in the array, with one of them being the pivot, the partition function will give us a balanced partition for $n = 1$, we will also get a balanced partition trivially.

for $n = 3$,

$$x_1 \quad \underset{\text{median}}{\underset{\uparrow}{x_2}} \quad x_3$$

After fast QS,

$x_3 \; x_1 \; x_2$ - 1st reverse partition

Next calls, Fast_QS($[x_3]$)

Fast_QS($[x_1, x_2]$)

Since median x_2 is at the last element, we will have a balanced result from the partition function.

Consider Fast_QS to give balanced partitions for arrays of size $n/2$ which are sorted.

We show that Fast_QS will give balanced partitions for arrays of size $n+1$. (The next level i in the induction ladder).

For array $A\left(\underbrace{1 \text{ --- } n/2}_{\substack{A_1 \\ \text{size } n/2 \\ (n/2 - 1 + 1)}}, \underbrace{\tfrac{n}{2}+1}_{\substack{b \\ \text{pivot}}}, \underbrace{\tfrac{n}{2}+2 \text{ --- } n+1}_{\substack{A_2 \\ \text{size } n/2 \\ \left(n+1-\left(\frac{n}{2}+2\right)+1\right)}}\right)$

Fast_QS(A) will yield, in place,

$$A \rightarrow [\, \text{Fast\_QS}(A_2),\ \text{Fast\_QS}(A_1),\ b \,]$$

In the first call to the partition function, during quick sort, as median b is the pivot element, we get a balanced partition resulting in

$$A \rightarrow [\,Fast\_QS(A_1),\ b,\ Fast\_QS(A_2)\,]$$

$Fast\_QS(A_1)$ is a permutation of A_1 and elements in A_1 are less than b. Similarly, elements in A_2 are greater than b.

The nexts calls to the partition function will be on $Fast\_QS(A_1)$ and $Fast\_QS(A_2)$
By our inductive assumption, these calls and subsequent ones will yield balanced partitions.

Hence $Fast\_QS(A)$ will give balanced partitions when quick sort is applied for A of size $(n+1)$.

Our proof by induction is complete.

For $Fast\_QS(A)$,

$$T(n) = 2T(n/2) + \theta(n)$$

We divide our problem into two sub problems, our reverse partition procedure and rotation adjustment procedure if required, take linear time.

Using master theorem,

$$T(n) = a\,T(n/b) + f(n)$$

$a = 2,\ b = 2,\ f(n) = cn$

$$g(n) = n^{\log_b a} = n^{\log_2 2} = n$$

$$f(n) = \theta(g(n))$$

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

Fast_QS will produce the array for balanced partitions in $\theta(n \log n)$ time complexity.

Q1.c

Consider the array A after Fast_QS(A),

$$A = [\underbrace{A_1}_{\frac{n-1}{2}}, \underbrace{b}_{1}, \underbrace{A_2}_{\frac{n-1}{2}}], \quad \text{size}(A) = n$$

$$\text{Fast\_QS}(A) = [\text{Fast\_QS}(A_2), \text{Fas\_QS}(A_1), b]$$

For every element in Fast_QS(A_2), which is a permutation of elements in A_2, every element in Fast_QS(A_1) and element b, will be lesser.

$$\therefore \text{ Number of inversions} \geq \underbrace{\left(\frac{n-1}{2}\right)}_{\text{no of elements in } A_2} \cdot \left(\underbrace{\frac{n-1}{2}}_{\text{no of elements in } A_1} + \overset{b}{1}\right)$$

$\therefore$ No of inversions $\geq \frac{1}{4}(n^2 - 1)$

While we divided our array into sizes $\left(\frac{n-1}{2}\right), 1, \left(\frac{n-1}{2}\right)$, we can do a similar analysis for when $n/2$ is an integer.
There will be other inversions with respect to two elements within Fast_QS (A_1) and Fast_QS (A_2) and b, but our analysis suffices to show the asymptotic bounds.

For insertion sort,

$$T(n) = \theta(n + I)$$

$$= \theta(n + c_1 n^2)$$

$$\therefore c_2(n + c_1 n^2) \leq T(n) \leq c_2(n + c_1 n^2)$$

for some c_1, c_2, for all $n > n_1$.
$\therefore$ For some constant c_0 for all $n > n_0$.

$$c_0 n^2 \leq T(n)$$

$$\therefore T(n) = \Omega(n^2)$$

Q1.d.

$$A = [A_1 \; b \; A_2] \rightarrow [A_2 \; A_1 \; b]$$

What we need the reverse partition function to do is that pivot b must go to the last element and for sub arrays A_1 and A_2

the median must be at the end. Such conditions suffice our Fast_QS analysis as we saw earlier. We can replace median b with the last element in element in A_2 and achieve the same conditions.

We make our fast partition function work in constant time.

We are essentially putting the median in the last position and we show that this is a valid reverse partition for our array A.

- For $A[1, \ldots, n]$, we want to preserve the structure after calling partition on the array output by reverse partition.

- $A[1, \ldots, n] \rightarrow$

$$a_1 - a_x \; b \; a'_1 - a'_x$$

We transform into,

$$a_1 - a_x \; a'_1 - a'_x \; b$$

On call to partition,

elements $a_1 - a_x < b$

elements $a'_1 - a'_x > b$

$\therefore$ Looking at the boundary, we get

$A \rightarrow \; a_1 - a_x \; b \; a'_1 - a'_x$ after partition.

In order to make the reverse partition in $O(1)$ time, we construct an array of pointers representation of A.

Initialize_Fast_Reverse_Partition (A)

{

$A^*[1, \ldots, n]$ - Array of pointers to nodes

```
for i from 1 to n
    {
        n = node()
        n.key = A[i]
        A*[i] = n

    }
}
```

```
Fast_Reverse_Partition (A*, p, q, r)
  {
     exchange ( A*[q], A*[r])

     return
  }
```

We call Fast_QS $(A^*, start, end)$

For A,

$A \to a_1 - a_x \; b \; a_1' - a_x'$

We get,

A'

$A' \to a_1 - a_x \; a_1' - a_x' \; b$

After partition,

$$A \rightarrow a_1 - a_x \; b \; a'_1 \; a'_2 - a'_x$$

$a_1 - a_x$	$a'_1 \; a'_2 - a'_x$
next argument to partition is FQS of this sub array.	next argument to partition is FQS of this sub array

In the next calls to the partition function, FQS will put the respective medians at the end of the sub arrays and hence the partitions will be balanced. We can proceed further similarly.

$$T(n) = 2\,T(n/2) + O(1)$$

$$n^{\log_b a} = n^{\log_2 2} = n$$

$$f(n) = O(1)$$

Using master theorem,

$$f(n) = O\left(n^{\log_2 2 - \epsilon}\right), \quad \epsilon > 0$$

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

Q1.e

We count the number of inversions for fast reverse partition. Without loss of generality, for simplicity we assume that $n = 2^k$

Consider the median at position $n/2$. It is now at the last position in the array. It has $\frac{n}{2}$ numbers greater than it behind it and thus has $\frac{n}{2}$ inversions.

Consider the next two medians for the next two sub arrays,
Original array element positions, $\left(\frac{n}{2} + \frac{n}{4}\right)$, $\left(\frac{n}{2} - \frac{n}{4}\right)$

The number of inversions for each of these medians with respect to those sub arrays will be $\frac{n}{4}$ each, which makes it $\frac{n}{2}$.

The next set of medians will be $\left(\frac{n}{2} + \frac{n}{4}\right) \pm \frac{n}{8}$ and $\left(\frac{n}{2} - \frac{n}{4}\right) \pm \frac{n}{8}$

For each of these four medians, with respect to that corresponding sub array, there will be $\frac{n}{8}$ inversions. Total becomes $4 \cdot \frac{n}{8} \sim \frac{n}{2}$

As we go down sets of medians, the number of such levels of medians will be $\log_2 n$ as $\frac{n}{2^{\log_2 n}} = 1$, will give us our last set of medians

Medians.

For each level, we have $\sim cn$ inversions.

$\therefore \; I = \Theta(n \log n)$

$\therefore$ For insertion sort,

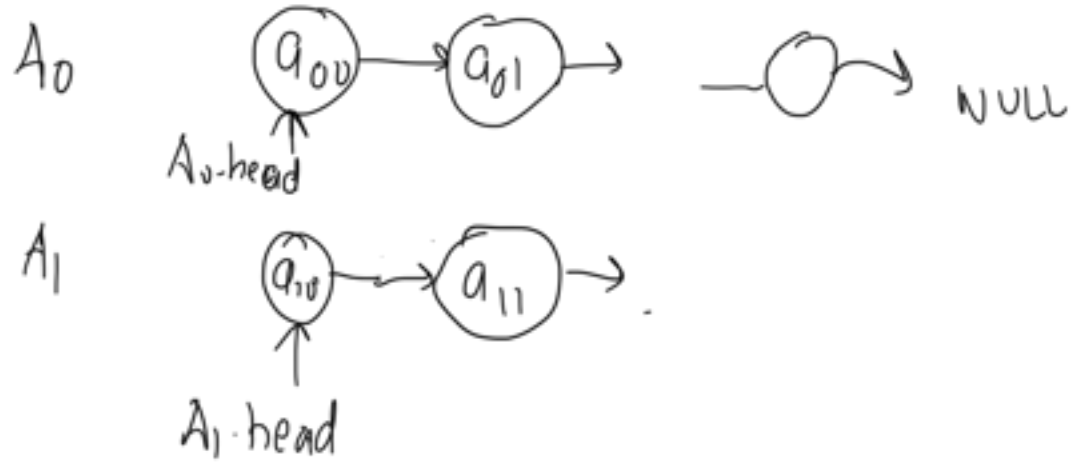
$$T(n) = \Theta(n + I) = \Theta(n \log n)$$

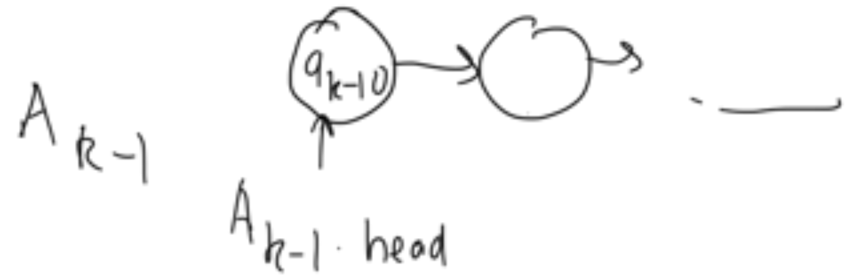
Q2.

(a)

We have been given linked lists $A_0, A_1, \ldots A_{k-1}$ of lengths $n_0, n_1, \ldots n_{k-1}$

We define Min Heap Q.
Elements in the min heap will be of the form $(x, list\_x)$ where x denotes the elements corresponding to which the Heap property is maintained.
$list\_x$ denotes the linked list from which element x is obtained.





$A_0, A_1, \ldots A_{k-1}$ are sorted.

We build a min heap with the keys of the elements pointed to by the heads of the linked list.

The next element in each list will be less than the head element and one of these head element keys will be minimum.

We then Extract min and go to the next element of that corresponding linked list from which the element was extracted.

We insert the new element in our heap.

We again extract min from our heap to find the second minimum element and iterate in that corresponding list. We repeat the process until completion. We present the algorithm and then the proof of correctness using the loop invariant.

We use the following Heap functions as an abstraction.

Q = Build_Min_Heap $([(x_1, l_1), (x_2, l_2) \ldots])$

- Takes in an array of Heap elements, builds a min heap based on the x values and returns an array. - $O(k)$ where k is the size of the Heap.

(x_{min}, l_{min}) = Extract_Min (Q)

- Returns (x_{min}, l_{min}), $O(\log k)$

Insert_Min_Heap $(Q, (x_i, l_i))$
- inserts (x_i, l_i) into the heap. - $O(\log k)$

We have, $n = n_0 + n_1 — n_{k-1}$
Let our new array be X.

Merge $(A_0, A_1, —, A_{k-1})$
{

Q = Build_Min_Heap([$(A_0.head.key, 0)$
$(A_1.head.key, 1)$
⋮
$(A_{k-1}.head.key, k-1)$])

$A_0^* = A_0.head$ # pointer to list A_0
$A_1^* = A_1.head$
⋮
$A_{k-1}^* = A_{k-1}.head$ # pointer to list A_{k-1}

for i from 1 to n
{
(a,b) = Extract_Min(Q)

X[i] = a

$A_b^* = A_b^*.next$

if $(A_b^* != NULL)$

```
        {
        Insert_Min_Heap(Q, (A_b.key, b))
        }
    }

    return X
}
```

Proof :

Loop invariant : At each iteration i, the i^{th} smallest element is selected and $(i-1)$ elements in sorted order from the smallest are present in X.

Initialization :

- In iteration 1 of the main for loop, we take the zeroth elements of each linked list to our heap. Since linked lists are sorted, the elements succeeding the elements succeeding them are larger than these elements. One of these elements is therefore the smallest and we obtain that element using extract min.

Maintainence :

Maintenance:

- In iteration i, we assume that $(i-1)$ smallest elements are present in X.
- We show that element i must be present in the current frontier from which the heap is made of.
- Suppose that element i was not present in the frontier. The heap will consist of k elements each of which will be greater than element i. Since the $(i-1)$ smallest elements are already in X and once we put an element in X, we iterate forward in that list and do not come across the element again.
- Since the lists are sorted, the elements ahead of the elements in the frontier are greater than the elements of the frontier.
- So the element i is not among elements not in our frontier. But element i must be present somewhere. Therefore our assumption is wrong. Hence, element i is present in the frontier.
- We obtain element i when we do extract min on the frontier elements which constitute the heap.
- Hence our loop invariant holds for iteration i.

Termination:

- In the final iteration, we obtain element n of X which completes our sorted array X.

Hence we have merged $A_0, A_1, \_\_ A_{k-1}$ to form

X.

for each of the n iterations of the loop, extraction from the heap and insertion into the heap take $O(\log k)$ time complexity.

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

where n is $(n_0 + n_1 - n_{k-1})$

Using the definition of Big Oh,

$$T(i) \leq c \log k$$

$$\sum_{i=1}^{n} T(i) \leq cn \log k$$

$$T(n) \leq cn \log k$$

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

Q2.b

We have,
Array $A[1, \_ , n]$ for n words.

distinct words range from $0, 1, \_ . k-1$
Correspondingly we will have linked lists $A_0, A_1, - A_{k-1}$.

$A[i] = j$, if word at position i is the j^{th} word in the vocabulary.

We iterate through array A and for each word we encounter in A, we add a corresponding element to the linked list for the word in the vocabulary.

```
GenerateSubArray (A[1, _, n], n, k)

{  A0 = LinkedList(), _ , Ak-1 = LinkedList()
                              # initialize
   # A0.head = NULL , A0.tail = NULL  # node
                                        pointers

    for i from 1 to n
    {
          x = node()
          x.key = A[i].
          j = A[i]        # vocabulary word
                            index

          if (Aj.head == NULL) {
              Aj.head = x
              Aj.tail = x
          }

          else {

            Aj.tail.next = x
            Aj.tail = x
          }
    }

   return  A0.head, A1.head, _ _, Ak-1.head
```

}

As we iterate through A once,

$T(n) = O(n)$

Q2.c

Let A [start, _ , end] be a minimum span.

let A(start) = x
let A(end) = y

A [start _ _ end]
x y

A [start + 1] cannot be x and A [end − 1] cannot be y as that would violate our definition of minimum span.

We run procedure (a) on our new lists A_0, A_1 — A_{k-1}.

- Consider the time when we pick start as the minimum element through A_x

- Each time, the heap will be consisted of the k frontier elements corresponding to the k linked lists.

- Since start is the minimum element, all of the elements in the frontier must be greater than start.

- A[start – end] is the minimum span.
- For every linked list apart from A_x and A_y, the ~~element~~ first element after start corresponding to that list will be present in the heap. Consider e_i to be present for A_i.

  If e_i is not the first element for A_i after start, that would mean that the first element after start has been iterated over. But start is our minimum element at our iteration timestep and therefore e_i must be the first element for A_i after start. This follows from the correctness of procedure (a).

- Let e_y be the element in A_y at our time step. e_y is the first element after start corresponding to A_y.

- If e_y is less than end, start – e_y – (end-1) will constitute a span thus violating our definition that start – end is the minimum span.

- Hence e_y must be end.
- Thus, for the time step in which we select start as the minimum element, end will also be present in the heap along with $k-2$ other elements corresponding to other lists.

Q2.d

- Since [start, end] will be part of the frontier constituting the heap; we will use a max Heap P.

- If there is a span and it occurs in the heap, the remaining k-2 elements must be between the limits of the span.

- For any frontier consisting of k elements from the k linked lists.
- let m_1 be the minimum element and m_2 be the maximum element, the other k-2 elements corresponding to other k-2 linked lists are therefore between m_1 and m_2.

- Hence, (m_1, m_2) constitutes a span, albeit not necessarily a minimum span.

- We iterate through all the possible frontiers as we use procedure (a). We generate the corresponding min and max for the span and compare through iterations

Find_Min_Span $(A_0, \_\_, A_{k-1})$

{

L

$P = \text{Build\_Max\_Heap}([(A_0.\text{head.key}, 0),$
$(A_1.\text{head.key}, 0),$
$\vdots$
$A_{k-1}.\text{head.key}, 0)])$

$Q = \text{Build\_Min\_Heap}([(A_0.\text{head.key}, 0)$
$(A_1.\text{head.key}, 1)$
$\vdots$
$(A_{k-1}.\text{head.key}, k-1)]$

$A_0^* = A_0.\text{head}$ # pointer to list A_0
$A_1^* = A_1.\text{head}$
$\vdots$
$A_{k-1}^* = A_{k-1}.\text{head}$ # pointer to list A_{k-1}

span_val = ∞

for i from 1 to n
{
(a, b) = Extract_Min (Q)
(c, d) = return_Max (P)

if (c - a < span_val) {

min_span = (a, c)
span_val = c - a

```
        }
        A*_b = A*_b.next
        if ( A*_b != NULL)
          {
            Insert_Min_Heap(Q, (A*_b.key, b))
            Insert_Max_Heap(P, A*_b.key, b))
          }

    }

}
```

- Essentially, min heap Q and max heap P will consist of the same configuration of frontier elements.
- In order to effectively find the maximum element for every configuration of the min heap, we use the max heap.
- The heap operations take $O(\log k)$ time for each iteration of the main for loop which is n times.
- $O(\log k)$ for Extract_Min(Q)
  $O(1)$ for return_Max(P)
  $O(\log k)$ for insertions in each of the heaps.

$\therefore \quad T(n) = O(n \log k)$ //

Problem 3

3(a)

```
k-Parent (i) {
        return ⌈(i-1)/k⌉
}

k_Child (i, j) {
        return (ki + 1) - (j - 1)
}
```

3-(b)

For element i, k children will go from -

$$\left\{ \begin{array}{ll} ki+1 & - \; c_1 \\ ki+0 & - \; c_2 \\ ki-1 & \end{array} \right.$$

$$i \longrightarrow \begin{cases} \vdots \\ ki-(k-2) - c_k \end{cases}$$

$$c_1 > c_2 — > c_k$$

for c_1,

$$\left\lceil \frac{ki+1-1}{k} \right\rceil = i$$

for c_k

$$\left\lceil \frac{ki-(k-2)-1}{k} \right\rceil = \left\lceil i - \frac{k-1}{k} \right\rceil$$

$$= i$$

for the element after c_1,

$$\left\lceil \frac{ki+2-1}{k} \right\rceil = \left\lceil i + \frac{1}{k} \right\rceil = i+1$$

This is correct as the element next after c_1 will be the child of $(i+1)$.

for the element before c_k,

$$\left\lceil \frac{(ki - (k-2) - 1) - 1}{k} \right\rceil = \left\lceil \frac{ki - k}{k} \right\rceil$$

$$= \left\lceil \frac{k(i-1)}{k} \right\rceil$$

$$= \lceil (i-1) \rceil$$

$$= i - 1$$

This is correct as the element before c_k will have $(i-1)$ as its parent.

if $\quad x_1 < x < x_2$

and $\quad \lceil x_1 \rceil = \lceil x_2 \rceil = 2$

then $\quad \lceil x \rceil = 2$

Therefore the elements between c_1 and c_k will give i correctly as the parent.

This completes our verification.

(c)

k_Max_Heapify (A, i, k)

{

largest = 0

```
for j from 1 to k {
  if (A[k_child(i,j)] > largest) and
     (k-Child (i,j) ≤ A.heap_size)) {
          largest = k_child (i,j)
      }
  }

  if (A[i] < A[largest])
     {
        exchange (A[i], A[largest])
        k_Max_Heapify (A, largest, k)
     }

}
```

(d)

k_Extract_Max (A)

{

{

$m = A[1]$
$A[1] = A[A.heap\_size]$
$A.heap\_size = A.heap\_size - 1$

k_Max_Heapify (A, 1, k)

return m

}

(e) let d be the depth in the tree.
level 1 will have 1 element.
level 2 will have k elements.
Each of those k elements will have k children,
level 3 will have k^2 elements

⋮

level d will have at max k^d elements in case the tree is full

$$n \leq 1 + k + k^2 - k^d$$

Also,
$$1 + k + \_ k^{d-1} < n$$

leaf nodes will have depth d, height 0
penultimate level nodes, depth d-1, height 1
|

root node $\rightarrow$ depth 1 height $d-1$
$= h$

$\therefore \quad 1+k - k^h < n \leq 1+k - k^{h+1}$

$$S_1 < n \leq S_2$$

$$S_1 = 1+ k - k^h$$
$$kS_1 = k - k^{h+1}$$

$$S_1 = \frac{k^{h+1}-1}{k-1} = \frac{k \cdot k^h}{k-1} - \frac{1}{k-1}$$

$$S_2 = \frac{k^{h+2}-1}{k-1} = \frac{k^2 \cdot k^h}{k-1} - \frac{1}{k-1}$$

There exist c_1, c_2 such that,

$$c_1 k^h < S_1$$
$$S_2 < c_2 k^h \quad \text{for all } h \geq h_0$$

$\therefore$ We can see that,

$$n = \Theta(k^h)$$

$$h = \Theta(\log_k n)$$

Each individual call to k_Max_Heapify will

be $\Theta(k)$
The number of calls to k_Max_Heapify will be $\Theta(h) \equiv O(\log_k n)$ for the completion of the recursion for the k_Max_Heapify function.

$\therefore$ For k_Max_Heapify,

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

We minimize $f(k) = k \log_k n$ wrt k

$$f(k) = k \log_k n$$

$$= \frac{k \ln n}{\ln k}$$

$$f(k) = \ln n \cdot \frac{k}{\ln k}$$

$$\frac{d\,f(k)}{dk} = \ln n \cdot \left(1 \cdot \frac{1}{\ln k} - \frac{1}{(\ln k)^2} \cdot \frac{1}{k} \cdot k \right)$$

$$= \ln n \left(\frac{\ln k - 1}{(\ln k)^2} \right)$$

$$\frac{d\,f(k)}{dk} = 0$$

$$\frac{\ln k - 1}{(\ln k)^2} = 0$$

$$\ln k = 1$$
$$k = e$$

The closest integers to e are 2 and 3

as $k \to 1$, $f(k) \to \infty$
as $k \to \infty$, $f(k) \to \infty$

$f(k)$ will have a minima at $k = e$

for our integer k purposes,

$$\frac{3}{\ln 3} < \frac{2}{\ln 2} \quad (2.73 < 2.88)$$

$\therefore$ We take $k = 3$ as our solution to the optimization.