2. (a) Parent \([d,i] = \left\lceil \frac{(i-2)}{d} + 1 \right\rceil \) 

Child \([d,i,K] = d \cdot (i-1) + k + 1, 1 \leq K \leq d\) 
(Note that there are \(d\) children)

(b) \(\Theta(\log d n)\) 

(c) Extract-Max is similar 
Max-heapify runs through a loop (all \(d\) children, instead of just two, \(l\) & \(r\)) 
Runtime \(\Theta(d \log d n)\)

(d) Similar to ones for regular heap in book, 
\(\Theta(\log d n)\) running times.
Consider the following heap tree. We can always know that the parent of a node at level \( k \) is at level \( k-1 \). The root node is at level 0, and its index is 1. If we remove an element from the heap, we also remove nodes from the lower layers. Assume now nodes in layer \( \log n \) have already been removed from the heap. We only consider the job done on layer \( \log n - 1 \).

Before the job:

\[
\begin{array}{c}
\text{A} \\
\text{B} \\
\text{C} \\
\text{D} \\
\text{E} \\
\text{F} \\
\text{G} \\
\end{array}
\]

After the job:

\[
\begin{array}{c}
\text{A} \\
\text{B} \\
\text{C} \\
\text{D} \\
\text{E} \\
\text{F} \\
\text{G} \\
\end{array}
\]

That is: All nodes in layer \( \log n \) have been removed.

Suppose before the job, the values in nodes from 1 to \( J \) are:

\( x_1, x_2, \ldots, x_m \). Each of theses values will first be moved to node A, then be exchanged to one of the nodes from layer 1 to layer \( \log n - 1 \). In order to have minimum running time, let's assume these nodes first occupy layer 1, then layer 2, \ldots, layer \( \log n - 2 \), layer \( \log n - 1 \). It is obvious that the total number of nodes from layer 1 to layer \( \log n - 2 \) is less than the number of nodes in layer \( \log n - 1 \). So in this case, all nodes in layer \( \log n - 2 \) will be occupied by one of the old value from 1 to \( J \). That is, \( m/2 \) nodes will be moved from layer 0 to layer \( \log n - 2 \). Since \( m > n/4 \), so for this job, the running time:

\[
T(n) > \frac{m}{2} (\log n - 2) = \frac{n}{8} (\log n - 2) = \frac{n}{8} (\log n - \frac{n}{4}).
\]

Since it is only part of the total running time of heapsort, so running time for heapsort must be:

\[
T(n) > \frac{n}{8} (\log n - \frac{n}{4}).
\]

For a large enough \( n \), we get:

\[
T(n) > \frac{n}{8} (\log n - \frac{n}{4}) > \frac{1}{8} n \log n = \log n.
\]

That is to say, even for best case, running time:

\[
T(n) = \log n.
\]
b) A tableau is Young if
\[ Y[i,j] < Y[i+1,j] \]
\[ Y[i,j] < Y[i,j+1] \]

Therefore, if \( Y[1,1] = \infty \) all elements in the table have to be greater than \( Y[1,1] \), therefore all elements in this tableau must be \( \infty \)!! \( \Rightarrow \) Empty tableau.

Therefore, if \( Y[m,n] < \infty \), all elements in the tableau must be smaller than \( Y[m,n] \). This is only possible if they are \( < \infty \). \( \Rightarrow \) Tableau is full.
c) EXTRACT_MIN(Y)
    min \leftarrow Y[1,1]
    Y[1,1] \leftarrow Y[m,n]
    Y[m,n] \leftarrow \infty
    YOUNGITY(Y, 1, 1)
    return \text{min}

YOUNGITY(Y, i, j)
    smallest \leftarrow Y[i, j]
    if \ j+1 \leq n \text{ and } Y[i, j+1] < \text{smallest}
        smallest \leftarrow Y[i, j+1]
        new_i \leftarrow i
        new_j \leftarrow j+1
    if \ i+1 \leq m \text{ and } Y[i+1, j] < \text{smallest}
        smallest \leftarrow Y[i+1, j]
        new_i \leftarrow i+1
        new_j \leftarrow j
    if \ Y[i, j] = \text{smallest}
        exchange \ Y[i, j] \leftrightarrow Y[\text{new}_i, \text{new}_j]
        YOUNGITY(Y, \text{new}_i, \text{new}_j)

Runtime

\[ T(p) = T(n+m) = T(n+m-1) + \Theta(1) \]
\[ T(p-1) = T(p-2) + \Theta(1) \]
\[ T(p-2) = T(p-3) + \Theta(1) \]
\[ T(2) = T(1) + \Theta(1) \]
\[ T(1) = (m-n) \cdot \Theta(1) \]

\[ T(p) = \Theta(m+n) \]
INSERT (Y, key)

if Y[m,n] = 0
    then error "tableau is already full"

Y[m,n] ← key
Fix_tableau(Y,m,n)

Fix_tableau(Y,i,j)
largest ← Y[i,j]
if j-1 > 0 and Y[i,j-1] > largest
    largest ← Y[i,j-1]
new_i ← i
new_j ← j-1
if i-1 > 0 and Y[i-1,j] > largest
    largest ← Y[i-1,j]
new_i ← i-1
new_j ← j
if Y[i,j] = largest
    exchange Y[i,j] ← Y[new_i,new_j]
Fix_tableau(Y,new_i,new_j)

Rerun

n+m = p
T(p) = T(p-1) + \Theta(1)
T(p-1) = T(p-2) + \Theta(1)
\forall p

T(2) = T(1) + \Theta(1)

T(n) = p \cdot \Theta(1) \quad \forall n = m+n

T(n) = \Theta(m+n)
Sort \( A \)

for \( i \leftarrow 1 \) to \( \text{length} \left( A \right) \)

\[ n^2 \]

Insert \((Y, A[i])\) \( O(n) \)

\[ \text{min} \leftarrow A[1, 1], \text{output min} \]

while \( \text{min} \neq \infty \)

\[ n^2 \]

\[ \text{min} \leftarrow \text{Extract-Min}(Y) \] \( O(n) \)

\[ \text{output min} \]
6.3 Description. We start in the lower left corner. We compare the value there with the value we are looking for. If it is the requested value, we are done. If it is greater than the value we are looking for, we can eliminate this row (since it can not be in this row). If the value is smaller than the requested value, we can eliminate the column, since it is not possible that the value is in this column.

\[
\text{IS IN-TABLEAU} (Y, \text{value})
\]

\[
\begin{array}{c}
j \leftarrow m \\
i \leftarrow 1
\end{array}
\]

\[
\text{found} \leftarrow \text{false}
\]

\[
\text{while} (\text{found} = \text{false}) \text{ and } (j \geq 1) \text{ and } (i \geq 1)
\]

\[
\text{if } Y[j,i] = \text{value}
\]

\[
\text{then } \text{found} \leftarrow \text{true}
\]

\[
\text{else if } Y[j,i] > \text{value}
\]

\[
\text{then } j = j - 1 \quad \text{"eliminate } j^{th} \text{ row}"
\]

\[
\text{else}
\]

\[
i = i + 1 \quad \text{"eliminate } i^{th} \text{ column}"
\]

* If either i or j gets lower than 1 then the value can not be in the tableau.

Runtime: It is obvious, that the while loop is executed maximized \((m+n)\) times. Each run is \(O(1)\).

Therefore \(T(p) = O(m+n)\). With recursion \(T(p) = T(p-1) + O(1)\)

\(T(1) = O(m+n)\)

We could start either in the corner \(Y[1,1]\) or \(Y[m,n]\) but not in corner \(Y[1,n]\) or \(Y[m,1]\)!
(7.2-1) \[ T(n) = T(n-1) + \Theta(n) \]

Show that \( T(n) = \Theta(n^2) \)

\[
\begin{align*}
T(n) &= T(n-1) + \Theta(n) \\
T(n-1) &= T(n-2) + \Theta(n-1) \\
T(n-2) &= T(n-3) + \Theta(n-2) \\
& \vdots \\
T(2) &= T(1) + \Theta(2) \\
\end{align*}
\]

where \( T(2), T(1), \) and \( \Theta(2) \) are constants.

\[
T(n) = T(1) + \sum_{i=2}^{n} \Theta(i) = T(1) + \Theta\left(\frac{n(n+1)}{2}\right)
\]

\[ T(n) = \Theta(n^2) \]
\[ T(n) = \max_{0 < q \leq n-1} \left( T(q) + T(n-q-1) \right) + \Theta(n) \]

Assume \( T(n) \geq cn^2 \)

\[ T(n) \geq c \cdot \max \left\{ q^2 + (n-q-1)^2 \right\} + d \cdot n \]

\[ \max \max \text{ can be at critical points} \]
\[ \text{(end pts, derw = 0)} \Rightarrow \]
\[ \text{above function has max \text{ at} q = 0, n-1} \]

\[ T(n) \geq c \cdot n^2 - 2cn + c + d \cdot n \]
\[ \geq c \cdot n^2 + (d-2c) \cdot n + c \]

True \( \Rightarrow (d-2c) \geq 0 \Rightarrow c \leq \frac{d}{2} \)

\[ T(n) \geq c \cdot n^2 \]

\[ T(n) = \Omega(n^2) \]
Problem Set 2
4/16/02

5) Problem 7-6

I  A  B  C  or no overlap

II  A  B  C  or overlap

III  A  C  C  or all overlap

I  \( c_a < c_b < c_c \)  just 1 case  what is \( c \)?

II  \( c_a \leq c_c \leq c_b \)
   \( c_c \leq c_a \leq c_b \)  3 cases
   \( c_a \leq c_b \leq c_c \)

III  \( c_a \leq c_b \leq c_c \)
   \( c_a \leq c_c \leq c_b \)
   \( c_b \leq c_c \leq c_a \)
   \( c_b \leq c_a \leq c_c \)
   \( c_b \leq c_a \leq c_c \)  6 cases
   \( c_b \leq c_b \leq c_b \)

Idea: when input array contains overlapping intervals
- these intervals don't need to be sorted - are already sorted due to "fuzzy behavior"

\[ \begin{align*}
&\text{5 cases of overlapping between pivot and other element} \\
1. & \text{no overlap at all} \quad a) \quad c_a \leq c_c \leq c_b \\
2. & \text{end of pivot overlaps start of elem} \quad \text{or} \quad b) \quad c_b \leq c_c \leq c_a \\
3. & \text{elem overlaps part of pivot} \quad \text{or} \quad \text{pivot is contained in elem} \\
4. & \text{pivot is contained in elem} \\
\end{align*} \]
My solution to that problem is not memory-efficient at all, but
that's not what we're asked to do, so I hope it's fine.

Every recursive call of the fuzzy-quick-sort has to create
4 additional arrays of at most size n of the input array.

1. Pick pivot out of original array (deterministic or randomly)

2. Create arrays:
   I  
   | contains all elements < pivot like \[ a < \text{ pivot} \]
   case 1 b
   | has to be sorted recursively

   II  
   | contains all elements > pivot like \[ \text{ pivot} > a \]
   case 1 a
   | has to be sorted recursively

   III case 2, 3, and 5
   | all overlaps

fell all in case 2 \( (\text{ pivot } \quad \text{ elem}) \) from right to left
fell all in case 3 \( (\text{ elem } \quad \text{ pivot}) \) from left to right

all in case 5 \( (\text{ elem } \quad \text{ pivot}) \) can be inserted from left to right or
right to left

> keep track of two indexes that move towards each other
> III is then already sorted

IV  

contains all case 4 overlaps \( (\text{ pivot } \quad \text{ elem}) \)
| has to be sorted as well because it could
be \( (\text{ pivot } \quad \text{ pivot}) \) 
where elements have to sorted as well

> keep track of smallest/shortest elem and sort IV
by using this smallest elem go through II and
separate elements again
Problem 2-6 (continued)

Now QuickSort III recursively by giving smallest interval as pivot

- case 4 can't happen anymore!

- merge all sub arrays back into A as:

\[ I = \text{lower half} + \text{pivot} + \text{upper half} + \bar{I} \]

- Note: lowercase the words like array[1], 0 and array[1].b

1. since algorithm uses QuickSort shuffle/technique
   it runs in expected \( \Theta(n \log n) \) time if the pivot
   positions the input array into 2 arrays (the arrays I and II)
   of about equal size.

   If no interval overlaps then Fuzzy-QuickSort (continue)
   just return because certain is empty

\[ \Theta(n \log n) \]

Right now the pivot is selected deterministically but it would
be easy to randomly choose a pivot.

If all intervals overlap like
then my algorithm runs in \( \Theta(n) \) time

\text{If} I \text{and} II \text{will be empty since all intervals overlap}
\text{away III will be sorted already and only } \Rightarrow \Theta(n)
\text{array I if not empty has to be sorted recursively}
second run: with array IV and smallest item in there as pivot

→ since all intervals overlap arrays I and II will be empty as well as array III because the pivot is the smallest and therefore can't contain another.

The only array that is filled is array IV which takes $\Theta(n)$

→ there would be 2 calls to Fuzzy-Quicksort and each call requires $2 \cdot \Theta(n)$ (to sort all elements into array and to put them back into array A passed as param)

$\Rightarrow 4 \cdot \Theta(n) = \Theta(n)$