6.1.1 ***What are the minimum and maximum number of elements in a heap of height h?***

We assume the question is asking about a binary heap. Since the height is h there is a number k of elements such that if there is one more element the height is $h+1$ *and there is a number j such that if we take one elment of this heap our height will be* $h-1$. *We know that each node has at most* 2 *children and this level indicate the the height a heap has the following property, at height* $=1$, *we consider the node at height* $=1$, *it has* 1 *element. height* 2 *has at most* 3 *elements and at least* 2 *elements height* 3 *has at most* 9 *elements and minimum of* 6 *elements.*

So it has at most 2^h-1 *elements and at minimun* 2^{h-1}

6.1.2 ***Show that an*** $n-$ ***element heap has height*** $\lfloor lgn \rfloor$

We showed that in a heap of height h there are at least 2^{h-1} *elements and* 2^h-1 *elements at most. So:*

$2^{h-1} \leq n \leq 2^h-1 \rightarrow \mathrm{h}-1 \leq \log(n) \leq \log\left(2^h-1\right) \leq h \rightarrow$

$h-1 \leq \log(n) \leq h \rightarrow$ *adjust for convetion in which we begin from height* 0

$h \leq \lg(n) < h+1 \rightarrow h$ *is an integer so it can only be* $\lfloor lgn \rfloor$

6.1.3 ***Show that in any subtree of a max*** $-$ ***heap, the root of the subtree contains the largest value occurring anywhere in that subtree***

This is clearly what guarantees the max $-$ *heap property since the parents are greater than the childs, by the transitivity property of* $>$ *we know this is true. Suppose that its not, so exists an element in that subtree, let denote it by* $S[j]$ *such that* $S[j] > S[1]$, $S[j]$ *have a parent of atleast* 1 *order, and the index of its parent is lower Than his index, and we have* $A[parentNode(S[j])] < S[j]$ *but this is a* max *heap and we need to have* $A[parentNode(S[j])] \geq S[j]$

6.1.4 ***Where in a*** **max** ***heap the smallest element reside, assuming that all elements are distinct?***

The miminum element must reside in a leaf, so without children, if it would have children This would imply that their values are $\leq$ *this min, since they are different, it would imply that the children have a lower value than the min, what is a contradiction. We cannot, although, get any more information based on that, it can be in any leaf since we don't have any relation between brothers*

6.1.5 ***At which levels in a max*** $-$ ***heap the*** $k-$ ***th largest element reside, for*** $2 \leq k \leq \left\lfloor \frac{n}{2} \right\rfloor$, ***assuming that all elements are distinct?***

The largest element is clearly in 0 *level, the second is clearly i the first level, the third can be at the first level or third. The deepest we can go is to achieve the the* $k-1$ *level. Furthermore in need to be in a level that have atleast k elements*

$2^h \leq k$. *if* $k=2$ *we can go from atleast level* 1 *to level* 1.

$if\ k = \left\lfloor \frac{n}{2} \right\rfloor \rightarrow 2^h \leq \left\lfloor \frac{n}{2} \right\rfloor \rightarrow h \leq \log\left(\left\lfloor \frac{n}{2} \right\rfloor\right) \rightarrow since\ it\ must\ be\ an\ integer$
$and\ it\ could\ go\ to\ \left\lfloor \frac{n}{2} \right\rfloor - 1\ of\ height.since\ there\ are\ n\ elements\ in\ our\ tree$
$The\ height\ of\ our\ tree\ is\ \lfloor lgn \rfloor\ which\ is < \left\lfloor \frac{n}{2} \right\rfloor - 1\ so\ it\ can\ go\ everyhwere\ except$
$the\ root$

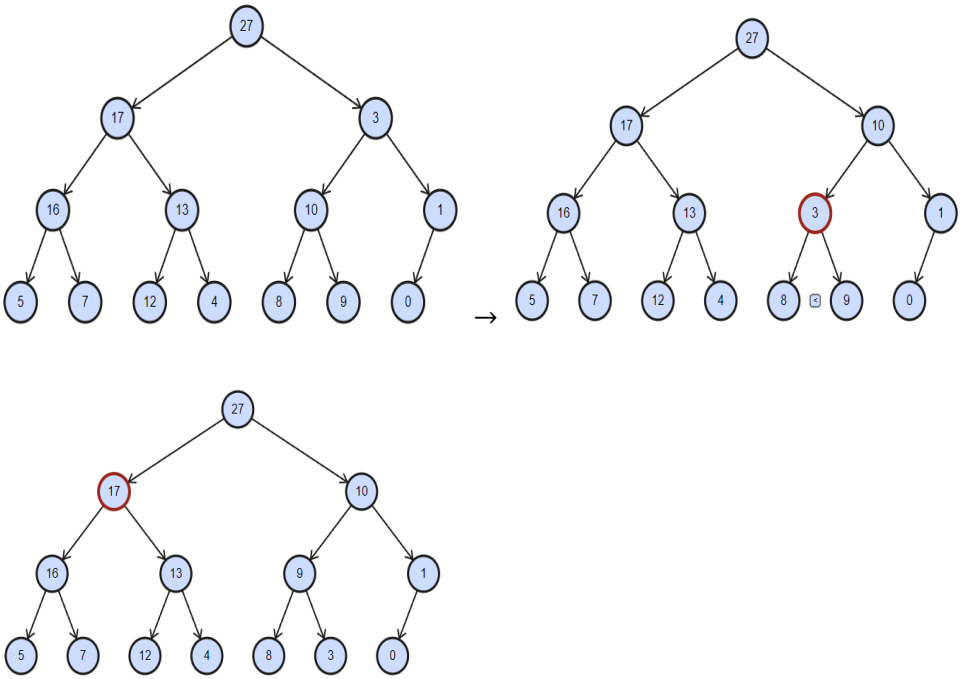
6.1.6 ***Is an array that is in sorted order a*** **min** ***heap***?
Yes, it clearly respect the condition to be a min *heap*

6.1.7 ***Is the array with values*** $(\mathbf{33, 19, 20, 15, 13, 10, 2, 13, 16, 12})$***a*** **max** ***heap***?
We can verify just by drawing, get the index beginning from 0
To be a max *heap we need that* $array[i] \geq array[2i+1], array[2i+2]$
Element $15, i = 3, has\ element\ i = 7\ as\ child, 13, and\ element\ i = 8\ as\ child$
$16, which\ is\ larger\ than\ 15, so\ its\ not\ a\ max\ heap$

6.1.8 ***Show that, with the array representation for storing an*** $\boldsymbol{n - element}$ ***heap, the leaves are the nodes indexed by*** $\boldsymbol{\left\lfloor \frac{n}{2} \right\rfloor + 1, \left\lfloor \frac{n}{2} \right\rfloor + 2 \ldots n}$

$Clearly\ after\ we\ find\ the\ exact\ separation, point\ where\ k\ is\ a\ leaf\ and\ k-1$
its not the nexts elements are all leaves.
$In\ this\ situation\ lets\ show\ that\ \left\lfloor \frac{n}{2} \right\rfloor\ is\ a\ node.\ since\ it's\ a\ node\ it\ must\ have\ a\ left\ child$
$all\ nodes\ have\ a\ left\ node\ whose\ index\ is\ equal\ the\ index\ of\ the\ node\ times\ 2$
$since\ the\ node\ is\ \left\lfloor \frac{n}{2} \right\rfloor < \frac{n}{2} \rightarrow\ 2\left\lfloor \frac{n}{2} \right\rfloor \leq n, it's\ a\ node.\ Now\ let's\ show\ that$
$\left\lfloor \frac{n}{2} \right\rfloor + 1\ has\ not\ a\ left\ child \rightarrow\ 2\left\lfloor \frac{n}{2} \right\rfloor + 2 = 2 * \left(\left\lfloor \frac{n}{2} \right\rfloor + 1\right) > 2\left(\frac{n}{2}\right) = n$
$So\ the\ index\ of\ the\ left\ child\ exceeds\ the\ actual\ number\ of\ elements$

6.2.1 ***Using figure 6.2 as a model, illustrate the operation of*** $\boldsymbol{MAX-HEAPIFY\ (A,3)}$ ***on the array***
$\boldsymbol{A = 27; 17; 3; 16; 13; 10; 1; 5; 7; 12; 4; 8; 9; 0}$



6.2.2 ***Show that each child of the root of an*** $\boldsymbol{n-node}$ ***heap is the root of a subtree containing at most*** $\boldsymbol{\frac{2n}{3}}$ ***nodes. What is the smallest constant*** $\boldsymbol{\alpha}$ ***such that each subtree hast at most*** $\boldsymbol{\alpha n}$ ***nodes? How does that affect the recurrence 6.1 and its solution?***

We always create the children of the left node first, so the left nodes are the only ones who can get the maximum number. this always happen when we achieve a new height and we add only the necessary child to complete the first subtree of the left from the left child of the root. Lets assume we have a height H. *the number of maximum elements are* $2^{h+1}-1$ *, and we have the maximum leaves as* $2^h, 2^{h-1}$ *for the left subtree and* 2^{h-1} *for the right subtree, above of that, if we take out all the leaves of this level our height go to* $h-1$ *and the number of total elements is* $2^h-1. \rightarrow 2^h-2$ *the subtree. So we have the following:* $\frac{\left(\frac{2^h-2}{2}+2^{h-1}\right)}{2^h-1+2^{h-1}}$ *ratio of left subtree elements for height h. This function monotonic increasing, and the limit goes to* $\frac{2}{3}$ *So the ratio is always* $<\frac{2}{3}$.

In the situation above we forced the left subtree have more weight. Now we force them to be equal so in a height h, each one have $(2^{h+1}-2)/2$ *in a total of* $2^{h+1}-1 \rightarrow$ $\frac{2^{h+1}-2}{2\cdot(2^{h+1}-1)}$. *The smallest constant is for* $n=3\ \alpha=\frac{1}{3}$.
It doesn't affect it all since for $\alpha<1$ *the solution is still* $O(\lg(n))$

6.2.3 ***Starting with the procedure*** $Max - Heapify$, ***write pseudocode for the procedure*** $MIN - HEAPIFY(A, i)$ ***which performs the corresponding manipulation on a*** $min - heap$. ***How does the running time of*** $MIN - Heapify$ ***compare with that of*** $MAX - heapify$?

Its the same thing, just swap $>$ *and* $<$ *the runing time is the same*

6.2.4 ***What is the effect of calling*** $MAX - HEAPIFY(A, i)$ ***when the element is larger than its children***?

Nothing

6.2.5 ***What is the effect of calling*** $MAX - HEAPIFY(A, i)$ ***for*** $i > A.heap - size/2$

We are at the leaves, so nothing

6.2.6 ***The code for*** $MAX - HEAPIFY$ ***is quite efficient in terms of constant factors, except possibly for the recursive call in line*** **10**, ***for which some compilers might produce inefficient code. Write an efficient*** $MAX - HEAPIFY$ ***that used an iterative control construct instead of a recursion***:

$MAX - HEAPIFY(A, i)$:
while $i < \left\lfloor \frac{n}{2} \right\rfloor$:
$l = LEFT(i)$
$r = RIGHT(i)$
$if (l \leq A.heap - size\ and\ A[l] > A[i])$
  $largest = l$
$else\ largest = i$
$if\ (r \leq A.heap - size\ and\ A[r] > A[largest])$:
  $largest = r$
$if\ largest\ !\ A[i]$
  $swap\ A[i]\ with\ A[largest]$
  $i = largest$
else:
  break

6.2.7 ***Show that the worst case running time of*** $MAX - HEAPIFY$ ***on a heap of size*** n ***is*** $\Omega(lg(n))$. (***Hint***: ***For a heap with*** n ***nodes, give node values that cause*** $MAX - HEAPIFY$ ***to be called recursivelly at every node on a simple path from the root down to a leaf***

Well. the hint gives everything, in the worst case you call it from the root and consecutively in all levels, until the highest height possible given n elements

With n elements the height is $\lfloor lgn \rfloor \rightarrow$ *we have then* $\lfloor lgn \rfloor + 1$ *calls on heapify*
In this case $MAX - HEAPIFY$ *run exactly in* $\lfloor lgn \rfloor + 1 \rightarrow \Theta(\lg(n)) \rightarrow \Omega(\lg(n))$

6.3.1 ***Using figure* 6.3** ***as a model, illustrate the operation*** $BUILDMAX - HEAP$ ***on the array*** $A = (5;\ 3;\ 17;\ 10;\ 84;\ 19;\ 6;\ 22;\ 9)$

Use the following Site with the array above as input

http://btv.melezinek.cz/binary − heap.html

6.3.2 ***Show that*** $\left\lceil \frac{n}{2^{h+1}} \right\rceil \geq \frac{1}{2}$ ***for*** $0 \leq h \leq \lfloor lg(n) \rfloor$

For a complete binary tree we have that
$2^h \leq n \leq 2^{h+1} - 1 \rightarrow 2^{h+1} \leq 2n \rightarrow 2^{h+1} > 2^{h+1} \rightarrow$
$2^h \leq n \leq 2^{h+1} - 1 < 2^{h+1} \leq 2n \rightarrow \frac{2^h}{2^{h+1}} \leq \frac{n}{2^{h+1}} \rightarrow \frac{1}{2} \leq \frac{n}{2^{h+1}} \leq \left\lceil \frac{n}{2^{h+1}} \right\rceil$

6.3.3 ***Why does the loop index i in line 2 of*** $BUILD - MAX - HEAP$ ***decrease from*** $\left\lfloor \frac{n}{2} \right\rfloor$

We want to make a bottom − up approach so we want to go from the leaves to the root

---

6.3.4 ***Show that there are at most*** $\left\lceil \frac{n}{2^{h+1}} \right\rceil$ ***nodes of height h in any n −element heap***

we can show it by induction, we know that a heap with only a root has 1 *element*
A heap with two levels has at most 3 *elements and at least* 2 *elements*
A heap with three levels has at most 7 *elevents and at least* 4 *elements*
in these cases we verify that we have 4, 2 *and* 1 *as maximun number of leaves*
Suppose for a $n - heap$, *the number of leaves is atmost* $2^{\lfloor \lg n \rfloor = h} \rightarrow$

We showed its valid for two, suppose its valid for h, now in the next level
So there are at most $2^{\lfloor \lg n \rfloor}$, *if all these leaves have children*
then we have $2 \cdot 2^{\lfloor \lg n \rfloor}$ *new leaves and height* $+ 1$. *so* $2^{\lfloor \lg n \rfloor + 1}$ *that is* 2^{h+1}
Now, since there are atmost $2^{\lfloor \lg(n) \rfloor}$ *leaves , this occurs at the last level.*
Then the previous level, $h - 1$, *there must be a maximun of* $2^{\lfloor \lg n \rfloor - 1}$ *nodes*
This is equivalent to the statement above, but we chosed h as 0 *from the root till h in the leaves*

6.4.1 ***Using figure* 6.4** ***as a model, ilustrate the operation Heapsort on the array*** $A = 5;\ 13;\ 2;\ 25;\ 7;\ 17;\ 20;\ 8;\ 4$

Use the following Site with the array above as input

http://btv.melezinek.cz/binary − heap.html

6.4.2 ***Argue the correctness of HeapSort using the following loop invariant***
At the start of each iteration of the foor loop of lines $\mathbf{2-5}$ ***the subarray*** $\boldsymbol{A[1:i]}$ ***is a max − heap containing the i smallest elements of*** $\boldsymbol{A[1:n]}$, ***and the subarray*** $\boldsymbol{A[i+1;n]}$ ***contains the*** $\boldsymbol{n-i}$ ***largest elements of*** $\boldsymbol{A[1:n]}$, ***sorted***

Lets verify the base case, where we are just before the first iteration, $i = n$
$A[1:n]$ *is a max − heap since we just called Build − Max − Heap on it.* $A[1:n]$ *obviously contain all elements and therefore it contains the n smallest elements of* $A[1:n]$ *and the subarray* $A[n+1;n]$ *is empty and contains the* 0 *largest element of* $A[1:n]$
Now suppose it works for i
Then $A[1:i]$ *is a max − heap containing the i smallest elements of* $A[1:n]$

$A[0]$ *is* $\geq$ *all other elements from* $A[1:i]$ *so it makes him the* $n-i+1$ *largest element*
We then make a swap, what guarantess that $A[i:n]$ *is properly sorted, Now if it was a Max heap then* $A[2]$ *and* $A[3]$ *were Max − Heaps two, because we just removed a child from one of them, what maintains the property so if we apply Max − Heapify in the first element transform* $A[1:i-1]$ *in a heap, so it guarantees*
For the termination when i woud be 1 *so* $A[1:1]$ *is clearly a max − heap which is the smallest element of the array by the previous iteration the following properties also holds*

6.4.3 ***What is the running time of HEAPSORT on an array A of length n that is already sorted in increasing order?*** ***How about if the array is already sorted in decreasing order?***

Even though its in an increasing order we will re
necessity so the algorithm still is $\mathrm{O}(nlg(n))$

6.4.4 ***Show that the worst case running time of HEAPSORT is*** $\boldsymbol{\Omega(nlgn)}$

The build part is $O(n)$ *so we won't bother with it. At the worst case for each iteration of the loop We will call* $MAX - HEAPIFY$ h *times so we have*:

$$\sum_{i=1}^{n-1} \lg(n-i) - 1 \leq \sum_{i=1}^{n-1} \lfloor \lg(n-i) \rfloor \leq \sum_{i=1}^{n-1} \lg(n-i)$$

$$\Theta(nlgn) = \log(n-1!) - n + 1 \leq \sum_{i=1}^{n-1} \lfloor \lg(n-i) \rfloor \leq \log(n-1!) = \Theta(nlgn)$$

So in the worst case it's still $\Omega(nlgn)$

6.4.5 ***Show that when all elements of A distinct, the best − case runnig time of HeapSort is*** $\boldsymbol{\Omega(nlg(n))}$

This showed to be a very tricky question and will be done later

---

6.5.1 ***Suppose that the objects in a max − priority queue are just keys.***
Illustrate the operation of MAX − HEAP − EXTRACT − MAX on the heap
15; 13; 9; 5; 12; 8; 7; 4; 0; 6; 2; 1

We basically copy the $A[0]$, *swap it with* $A[size]$ *and apply a* $max - heap$ *on it since we removed the last element, it swapped to be the first, we can still apply* $max - heapify$ *and it will position in the right position again*

6.5.2 Suppose that the objects in a max − priority queue are just keys.
Illustrate the operation of MAX − HEAP − INSERT(A, 10) on the heap A
$= \{15, 13, 9, 5, 12, 8, 7, 4, 0, 6\}$

We basically insert 10 *as the last element, all the other subtrees that aren't a direct parental lineage of this element is still a* $max - heap$, *so we can use a bottom up approach to rearragen the elements as needed*

6.5.4 ***Write pseudocode for the procedure***
$MAX - HEAP - DECREASEKEY(A, x, k)$
in a $max - heap$. ***What is the running time of your procedure?***

Since we are decreasing the value of the key we just need to apply a $max - heapify$ *on the element. Its still* $\log(n)$

6.5.5 ***Why does*** $MAX - HEAP - INSERT$ ***bother setting the key of the inserted object to*** $-\infty$ ***in line*** **5** ***given that line*** **8** ***will set the object's key to the desired value***

This is just an work round to assert that the increase method verify if its in fact increasing the value. Since the Insertion uses it as subroutine and doesn't change the value we needed to do this .

6.5.6 ***Professor Uriah suggests replacing the while loop of lines*** $5 - 7$ ***in*** $MAX - HEAP - INCREASE - KEY$ ***by a call to*** $MAX - HEAPIFY$.
Explain the flaw in the professor idea

Here we are doing a bottom up approach, $max - heapify$ *uses a top − down approach Since the value were increased the object maintain itself being a* $max - heap$, *so the heapify method won't do anything*

6.5.7 ***Argue the correctness of*** $MAX - HEAP - INCREASE - KEY$ ***using the following loop invariant:***
At the start of each iteration of the while loop of lines $5 - 7$:
a. If both nodes $PARENT(i)$ ***and*** $LEFT(i)$ ***exist,***
then $A[PARENT].key \geq i\ A[LEFT(i)].key$.

b. If both nodes $PARENT(i)$ and $RIGHT(i)$ exist,
then $A[PARENT(i)].key \geq A[RIGHT(i)].key$.

c. The subarray $A[1 : A.heap - size]$ satisfies the max
− heap property, except that there may be one violation, which is that $A[i].key$
may be greater than $A[Parent(i)].key$
You may assume that the subarray $A[1: A.heap - size]$ satisfies the
max − heap property at the time $MAX - HEAP - INCREASE - KEY$ is called.

Initialization: since it passed from the catch erro k is greater than the actual key
Since it was max − heap just before the update the parent of
i were greater than him that were greater than the left child. Since they didn't changed
values it still holds that parent > left. The same thing for the right child

Since we increased the value of i it still is bigger than left and right, if they exist.
In this situation we could have increased the value of its parent, so there might be one violation
that this cell is nowgreater than the parent

Maintenance, Since we do the swap and previously we already showed that the parent,
that now is in i position, were greater than LEft and Right, this still holds
Since the swap ocurred this violation was solved, but now there is possibly a new
violatio between the new i and its new parent, this is guaranteed by the I
= Parent(i) assignment

Termination: if the node is the root it has no parent or we found a parent(i) greater than the
actual i, if this case never occurs we will stop at the root because
we have a clearly convergent sequence

6.5.8 ***Each exchange operation on line 6 of $MAX - HEAP - INCREASE$***
−KEY typically requires three assignments, not counting the updating
of the mapping from the objects to array indices. Show how to use the idea of the
inner loop of $INSERTION - SORT$ to reduce the three assignments to just one

The idea is reuse $x.key$ already saved in a variable, so durgin the while loop we just
use the assignment $A[i] = A[parent(i)]$ and after the loop $A[i] = x$

6.5.9 ***Show how to implement a $first - in, first - out$ queue with a priority***
queue. Show how to implement a stack with a priority queue

The idea is that always that you insert a new element it must have a high priority
for the stack case, in this sense the priority can have a counter indicating
how many objects there are and this will be associated with the priority
So, just call insert with the actual value of this counter and increment it
For the queue the idea is the opposite, just decrement it, or use a
min − priority queue with the same strategy as said above

6.5.10 ***The Operation $MAX-HEAP-DELETE(A,x)$ deletes the object x from $max-heap$ A. Give an implementation of $MAX-HEAP-DELETE$ for an $n-element$ $max-heap$ that runs ins $O(lg(n))$ timeplus the overhead for mapping priority queue objects to array indices***

We can swap the desired element with the last one , based on their values we decide if we will do a $max-heapify$ or a $bottom-up$ approach to fix the heap
So we make $A[i]=A[A.size]$. if $A[i]$were lower we do the bottom up approach
So we swap the $A[i]$with its parent if necessary until the heap property be okay
In the case $A[i]$were larger we just call $max-heapify$. We just do swaps equ
al the height

6.5.11 ***Give an $O(nlg(k))-$ time algorithm to merge k sorted lists into one sorted list, where n is the total number of elements in all input lists. Hint(use a $min-heap$ for $k-way$ merging)***

We can maintain a $min-heap$ of size k, we begin from the smallest elements of each list. We then proceed to extract this element what takes $\lg(k)$
We insert another from the same list the one was extracted, what takes still $\lg(k)$
We do this for each element, what leads us to deduce, without a proof although that the algorithm is $nlg(k)$

6.1 ***Building a heap using insertion***
One way to build a heap is by repeatedly calling $MAX-HEAP-INSERT$ to insert the elements into the heap. Consider the procedure $BUILD-MAX-HEAP'$ on the facing page. It assumes that the objects inserted are just the heap elements

```
BUILD-MAX-HEAP'(A, n)
1  A.heap-size = 1
2  for i = 2 to n
3      MAX-HEAP-INSERT(A, A[i], n)
```

a. Do the procedures $BUILD-MAX-HEAP$ and $BUILD-MAX-HEAP'$ always create the same heap when run on the same input array? Prove that they do or provide a counter example

b. Show that in the wors case $BUILD-MAX-HEAP'$ requires $\Theta(nlgn)$ time to build an $n-element$ heap

No, both procedures may failt to produce the same structure, but they do produce heaps
a counter example would be an array 1,2,3 → *In this case* 2 *and* 3 *are leaves so build* max *heap will call $max-heapify$ on* 1 *and switch with* 3 *resulting in* 321
Now we will insert 2 *and* 3 *in the heap. two will begin as the root so* 213 *then we insert* 3
Who will take the place from 2, *so we finish with* 312

We will realize n inserts. when we do an insert we add at the end of the heap, at this time we can make at maximun h swaps. so we have

$$\Theta(nlgn) = \sum_{i=2}^{n} c_i \lg(i) - c_i \leq \sum_{i=2}^{n} c_i \lfloor \lg(i) \rfloor \leq \sum_{i=2}^{n} c_i \lg(i) \leq \sum_{i=2}^{n} c_{max} \lg(i) = c_{max} \lg(n!)$$
$$= \Theta(nlogn)$$

6.2 *Analysis of a d ary heapsa $d-ary$ heap is like a binary heap, but (with one possible exception)nonleaf nodes have d chieldren instead of two children. In all parts of this problem, assume that the time to maintain the mapping between objects and heap elements i $O(1)$per operation*

***a*) Describe how to represent a $d-ary$ heap in an array**
***b*) Using Θ notation, express the height of a d ary heap of n elements in terms of n and d**
***c*) Given an efficient implementation of $EXTRACT-MAX$ in a d ary max heap Analyze its running time in terms of d and n**
***d*) Given an efficient implementation of $INCREASE-KEY$ in a d $-ary$max heap. Analyze its running time in terms of d and n**
***e*) Given an efficient implementation of $INSERT$ in a $d-ary$ maxheap. Analyze its runnning time in terms of d and n**

a) Let an array of Index 1,2,3,4 *... n. We would have the root as index* 1 *the k children from* $1+1$, *until* $1+k$. *Then the k children of the first would have* $2+1 \ldots 2+k$. *So we have their children going from* $(k+\mathbf{2})$
$A[k^2+k+1]$

We could represent it like the $D-ary$ *parent of* (i):
$\left\lfloor \frac{i-2}{d} \right\rfloor + 1$
And the j element of the element i as:
$d(i-1)+j+1$

b)The first level have 1 *element, the second have k, the third have* k^2, *the fourth have* k^3 *at maximun, the height then is obviously* $\lfloor \lg_k n \rfloor$

c) So extractt MAX will do the same thing as in the $max-heap$, *we will swap it with the last element and perform a* $max-heapify$. *In this case we perform d comparisons at each level and we have a* $\lfloor d \log_d n \rfloor$ *for the height*
$\Theta(d \log_d n)$

d) So we will perform the same, given that we want to increase the value of the lowest to the larger, then we realize just a comparison between the new value and the value of the parent. if your new is bigger than swap, repeat until you are the root or the parent be bigger than you. In the worst case we have a $\Theta(\log_d n)$

e)$MAX-HEAP$ *insert works the same and have the* $\Theta(\log_d n)$

6.3 *Young tableaus*

An m x n Young tableaus is an mx n matrix suych that the entries of each roware in sorted order from left to right and the entries of each column are in sorted order from top to bootom. Some of the entries of a Young tableaucan be used to hold r $\leq mn$ ***finite numbers***

a) Draw a 4x4 Young tableau containing the elements $9; 16; 3; 2; 4; 8; 5; 14; 12$

b) Argue that an mx n Young tableau Y is empty if $Y[1:1] = \infty$. ***Argue that Y is full(contains mn elements) if*** $Y[m,n] < \infty$

c) Give an algorithm to implement EXTRACT $-$ ***MIN on a nonempty mxn Young tableau that runs in*** $O(m+n)$ ***time. Your algorithm should use a recursive subroutine that solves mx n problem by recursively solving either an*** $m-1$ ***x n or an mxn*** -1 ***subproblem.*** (***Hint: think about MAX*** $-$***HEAPIFY***). ***Explain why your implementation of EXTACT*** $-$ ***MIN runs in*** $O(m+n)$

d) Show how to insert a new element into a nonfull mx n Young tableau in $O(m+n)$

e) Using no other sorting method as a subroutine, showhow to use an nxn Young tableau to sort n^2 ***numbers in*** $O(n^3)$***time***

f) Given an $O(m+n)$ $-$ ***time algorithm to determine wheter a given number is stored in a given mxn Young tableau***

a) Many ways, one of them:

$$\begin{matrix} 2 & 3 & 9 & 12 \\ 4 & 8 & 14 & \infty \\ 5 & \infty & \infty & \infty \\ 16 & \infty & \infty & \infty \end{matrix}$$

b)Well we have that $A_{11} \leq A_{1n}$ *for all n and* $A_{11} \leq A_{m1}$ *for all m] and we know that* $A_{mn} \leq A_{jk}$ *for all* $m \leq j$ *given* $n = k$ *or* $n \leq k$ *for* $m = j$*. The argument, hat could be proved by induction is that the rest of line* 1 *is* ∞
So all other elements below these must be ∞ *. if it less than*∞ *its finite, so its a number*

c)Ok the minimun element is obivously the first, while the largest is the last A_{mn}
Since we will do a extract movement we need to change the value of the cell for ∞*, but* ∞ *is not sitable in that position. In this situation we will begin to make a similar* $MAX - HEAPIFY$ *STRATEGY, we will compare the* ∞ *position with its neighbours(the right and below one)and we will choose the lowest of them since the neighbours (the right and below one)must be greater than the actual position. So when we take the lowest one we guarantee that the position obeys the rules of the Young tableau. If we do n swaps in the columns and m swaps in the rows we put the* ∞ *in the last position and all cells are obeying the rule*
This takes obviously $O(m+n)$*, m swaps in the rows, n swaps in the column*

D) If it's a non full tableau then the last element is for sure the ∞ *we then place this new element in this position we now realize the comparison with its neighbours, the left and top one and we choose with the highest, we proceed it until all cells obey the rule*
This is $O(m+n)$

e) To insert a n element in a n^2 *Tableau we use* $O(2n)$*time, we insert* n^2 *so* $n^2 O(n) = O(n^3)$*. We then use EXTRACT* $\min$ *for each n, so* $n^2 O(n) = O(n^3)$

$f)$ *Since the tableau is sorted by row and column we can use this information in our favor*
We can begin from the top right position or from the bottom left. For the top right:
In each iteration, if the element is lower than the one in the cell we reduce by 1 *the column*
If is higher we sum 1 *in the row so the Complexity is* $O(n+m)$. *Why this is true*?
If its lower thanthe cell no other element in that column will be useful since they are greater
If its greater than, no other element in that row will be useful, since they are all lower
We verify that we can eliminate a row or column at a time, so we reduce the problem
To a $m-1$ *x n or mx* $n-1$ *subproblem that is guaranteed to have the value, if*
initially the tableau already had the value