Nisharg Gosai (002273353)
Algorithms Assignment 2

1. (20pts) Solve the following recurrences using the substitution method:

a. T(n) = T(n/2)+T(n/4)+T(n/8)+n.

Our guess is T(n) = O(n).

Show that T(n) <= cn for some constant c > 0

$T(n) = T(n/2) + T(n/4) + T(n/8) + n$

$T(n) <= c(n/2) + c(n/4) + c(n/8) + n$

$= n[c/2 + c/4 + c/8 + 1]$

$= n[7c/8 + 1]$

=n($7c/8$)+n

Divide $\frac{7}{8}cn$ into $cn - \frac{cn}{8}$

=$cn - \frac{cn}{8} + n$

<=cn if c=8

Therefore T(n) = O(n)

b. T(n)= 4T(n/2)+n2.

Our guess is T(n) = O(n2)

Show that T(n) <= cn2 for some constant c > 0

$T(n) = 4T(n/2) + n^2$

$T(n) <= 4(c\frac{n^2}{4}) + n^2$

$= cn^2 + n^2$

Which is not O(n^2), so we take a new guess,

T(n)<=$cn^2 lgn$ for some constant c>0

$T(n) = 4T(n/2) + n^2$

$T(n) <= 4(c\frac{n^2}{4} lg\frac{n}{2}) + n^2$

$= cn^2 lg\frac{n}{4} + n^2$

$= cn^2(lgn - lg2) + n^2$

$= cn^2(lgn - 1) + n^2$

$= cn^2 lgn - cn^2 + n^2$

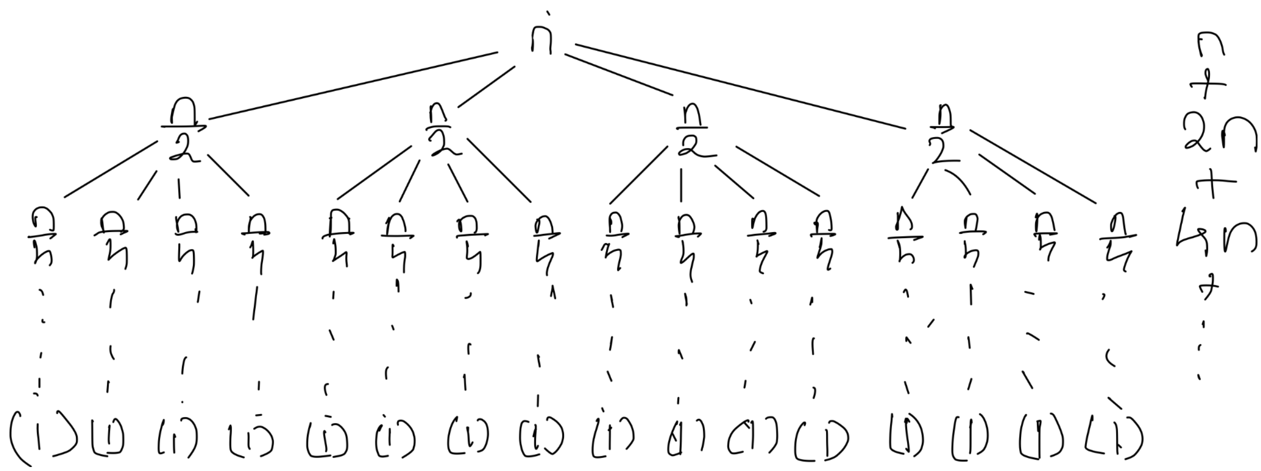
<=cn^2lgn if -cn^2+n^2=0 which is c=1

Therefore, T(n)=O(n^2lgn)

2. For the following recurrence, sketch its recursion tree and use the tree to guess a good asymptotic upper bound on its solution. Then use the substitution method to verify your answer.

T(n)= 4T(n/2)+n

The recursion tree is as follows: we assume that the base case costs constant time $\Theta(1)$



The height of tree is lgn and we have lgn+1 levels

Adding the cost of each level n+2n+4n+...+$2^k n$ = n $\sum_{i=0}^{k} 2^i$ = n*$\left(\frac{2^{k+1}-1}{2-1}\right)$ = n*(2^{k+1}-1)

Total no. of leaves = $4^{height} = 4^{lgn} = n^{lg4} = n^2$

Therefore the total cost is n^2 * n * (2^{k+1}-1)

Therefore the upper bound is O(n^2)

Verifying using substitution,

T(n)= 4T(n/2)+n

T(n)<=4(c$\frac{n^2}{4}$)+n

= c$n^2 + n$

We need a better guess which is T(n)<= $cn^2 - dn$

T(n)= 4T(n/2)+n

$T(n) <= 4(cn^2/4 - dn/2)+n$

$= cn^2 - 2dn + n$

$= cn^2 - dn - dn + n$

$<= cn^2 - dn$ if $d>1$

Therefore it is O(n^2)

3. Use the master method to give tight asymptotic bounds for the following recurrences:
a. T(n) = 2T(n/4)+1
b. T(n) = 2T(n/4)+n
c. T(n) = 2T(n/4)+n2
d. T(n) = 2T(n/4)+n1/2

We can use the master method to solve recurrences of the form
T(n)=aT(n/b)+f(n)

Where a>=1 and b>1, here f(n) is our driving function

We have 3 cases,

Case 1: f(n) = O($n^{log_b a-\varepsilon}$) for some constant $\varepsilon > 0$

(f(n) is polynomially smaller than $n^{log_b a}$)

Solution: T(n) = $\Theta(n^{log_b a})$

(Cost is dominated by leaves)

Case 2: f(n) = $\Theta(n^{log_b a} lg^k n)$, where k>=0 is a constant

(f(n) is within a polylog factor of $n^{log_b a}$, but no smaller)

Solution: T(n)=$\Theta(n^{log_b a} lg^{k+1} n)$

(Cost is $n^{log_b a} lg^k n$ at each level, and there are Θ(lgn) levels

Simple case: k=0, f(n) = $\Theta(n^{log_b a})$, T(n) = $\Theta(n^{log_b a} lg\ n)$

Case 3: f(n) = $\Omega(n^{log_b a+\varepsilon})$ for some constant $\varepsilon > 0$ and f(n) additionally satisfies the regularity condition $af(n/b) <= cf(n)$ for some constant c<1 and all sufficiently large n,

(f(n) is polynomially larger than $n^{log_b a}$)

Solution: T(n) = $\Theta(f(n))$

(cost is dominated by root)

a. T(n) = 2T(n/4)+1

Here a=2 and b=4, which implies $n^{log_b a}$ =$n^{log_4 2}$ =$n^{log_4 4^{\frac{1}{2}}}$ =$n^{\frac{1}{2} log_4 4}$=$\sqrt{n}$,
Our f(n) = 1

$n^{log_b a}$>f(n), we can use Case 1,
f(n) = O($n^{log_b a-\varepsilon}$) for some constant $\varepsilon > 0$
f(n)=O($n^{log_4 2-\epsilon}$), for ε>0,
If we take ε=½ then f(n)=O(1)

Therefore T(n)= $\Theta(n^{log_b a})$ **=** $\Theta(\sqrt{n})$

b. T(n) = 2T(n/4)+n

Here a=2 and b=4, which implies $n^{log_4 2}$= $\sqrt{n}$
Our f(n) = n

$n^{log_b a}$<f(n), we can use Case 3,
f(n) = $\Omega(n^{log_b a+\varepsilon})$ for some constant $\varepsilon > 0$
f(n)=$\Omega(n^{log_4 2+\varepsilon})$,
If we take ε=½ then,
=$\Omega(n^{1/2 log_4 4+1/2})$=$\Omega(n^{1/2+1/2}) = f(n)$

It also satisfies the regularity condition,

Therefore T(n)= $\Theta(f(n))$ **=** $\Theta(n)$

c. T(n) = 2T(n/4)+n^2

Here a=2 and b=4, which implies $n^{log_4 2}$= $\sqrt{n}$
Our f(n) = n^2

$n^{log_b a}$<f(n), we can use Case 3,
f(n) = $\Omega(n^{log_b a+\varepsilon})$ for some constant $\varepsilon > 0$
f(n)=$\Omega(n^{log_4 2+\varepsilon})$,
If we take ε=3/2 then,

$=\Omega(n^{1/2log_4 4+3/2})=\Omega(n^{1/2+3/2}) = f(n^2)$

It also satisfies the regularity condition,

Therefore T(n)= $\Theta(f(n))$ **=** $\Theta(n^2)$

d. T(n) = 2T(n/4)+$n^{1/2}$

Here a=2 and b=4, which implies $n^{log_4 2}$= $\sqrt{n}$
Our f(n) = $n^{1/2}$

$n^{log_b a}$= f(n), we can use Case 2,
f(n) = $\Theta(n^{log_b a} lg^k n)$ where k>=0 is a constant,
If we take k=0(simple case) then f(n) = $\Theta(n^{log_b a})$,
f(n)=$\Theta(n^{1/2})$=f(n),
Therefore T(n) = $\Theta(n^{log_b a} lg\ n)$ **=** $\Theta(\sqrt{n} lg\ n)$

4.

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

The smallest value element should be a node which is not a parent of any nodes since in max-heap parent node is larger than child node, therefore the smallest element is a leaf node.

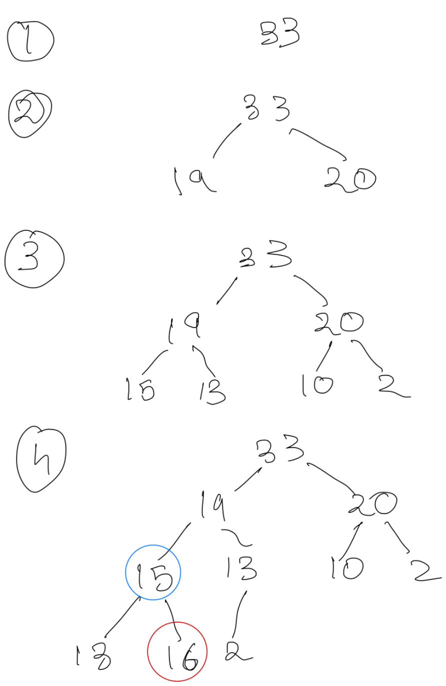
b. Is the array with values (33, 19, 20, 15, 13, 10, 2, 13, 16, 12} a max heap?

We build the max heap using the array, we need to remember that in a max heap the root is the largest value and for all nodes *i* except the root, A[Parent(i)]>=A[i]

We also have formulas,

Root of tree is A[1]
Parent of A[i] = A[[i/2]]
Left child of A[i] = A[2i]
Right child of A[i]=A[2i+1]

[33, 19, 20, 15, 13, 10, 2, 13, 16, 12]



Here the property of max heap is violated, since 15 is not greater than 16, therefore this is not a max heap.

5. Write an efficient MAX-HEAPIFY that uses an iterative control construct (a loop) instead of Recursion.

Assumptions: 1) there is a i^{th} node which needs to be rearranged to maintain heap property
2) Except i^{th} node, entire tree is a heap

```
MAXHEAPIFYINS(A,i)

    If A.heap-size=1 // check for heap array of size 1
        Return
    Else
        While A.heap-size>1
            l=LEFT(i)   //get the left child of i
            r=RIGHT(i)   //get the right child of i

            If l <= A.heap-size and A[l]>A[i]
                largest=l
            Else largest=i

            If r <= A.heap-size and A[r]>A[largest]
                largest=r

            If largest=i
                Print "heafiy done"
                Break

            SWAP(A[i],A[largest])
            i = largest
        end
```

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

To solve we need to

1) Build a min heap using first element of the lists
2) Extract minimum element and add it to final list
3) Take the next element from the same list and add it to final list

We have n steps and insertion into heap is lg k

```
MERGELIST(lists)

    For i from 1 to length(lists)

        L = FirstElement(lists) // L is an array with first elements of lists

    A = MIN-HEAP(L)   //Build a min heap A from L
    While MIN-HEAP not empty
        M = Heap-extract-min(A) //Add element to final list which is M

    return M
```