Homework 2

Comp 3270

Haden Stuart / has0027

Problem 1.)

If $f(n) = O(g(n))$ then $f(n) \le c * (g(n))$ (Bounded from above)

If $f(n) = \Omega(g(n))$ then $f(n) \ge c * (g(n))$ (Bounded from below)

If $f(n) = \Theta(g(n))$ then $c_1 * (g(n)) \le f(n) \le c_2 * (g(n))$ (Tight bound from above and below)

a.) $f(n) = 100n + \log n \mid g(n) = n + (\log n)^2$

Let's assume this takes the upper bound form: $f(n) \le c * (g(n))$

⇨ $100n + \log n \le c * (n + (\log n)^2)$

When c = 100 and n = 1 we get:

⇨ $100(1) + \log(1) \le 100 * (1 + (\log(1))^2)$
⇨ $100 \le 100$

Since 100 = 100 is true, this holds making this an upper bound.

Now let's assume this takes the lower bound form $f(n) \ge c * (g(n))$.

⇨ $100n + \log n \ge c * (n + (\log n)^2)$

When c = 1 and n = 1 we get:

⇨ $100(1) + \log(1) \ge 1 * (1 + (\log(1))^2)$
⇨ $100 \ge 1$

Since 100 > 1 is true and there is no c value that can make this false, this holds making this a lower bound.

These two functions create the form $c_1 * (g(n)) \le f(n) \le c_2 * (g(n))$, meaning that

f(n) = Θ(g(n))

b.) $f(n) = \log n \mid g(n) = \log(n^2)$

Let's assume this takes the upper bound form: $f(n) \le c * (g(n))$

⇨ $\log n \le c * (\log(n^2))$

When c = 2 and n = 100 we get:

⇨ $\log(100) \leq 2 * (\log(100^2)$
⇨ $2 \leq 8$

Since $2 \leq 8$ is true, this holds making this an upper bound.

Now let's assume this takes the lower bound form $f(n) \geq c * (g(n))$.

⇨ $\log n \geq c * (\log(n^2))$

When c = 1 and n = 1 we get:

⇨ $\log(1) \geq 1 * (\log(1^2))$
⇨ $0 \geq 0$

Since 0 = 0 is true and there is no c value that can make this false, this holds making this a lower bound.

These two functions create the form $c_1 * (g(n)) \leq f(n) \leq c_2 * (g(n))$, meaning that

f(n) = Θ(g(n))

c.) $f(n) = n^2/\log n$ | $g(n) = n(\log n)^2$

Let's assume this takes the upper bound form: $f(n) \leq c * (g(n))$

⇨ $n^2/\log n \leq c * (n(\log n)^2)$

When c = 100 and n = 10 we get:

⇨ $10^2/\log(10) \leq 100 * (10(\log(10))^2)$
⇨ $100 \leq 100$

Since 100 = 100, this holds making this an upper bound.

Now let's assume this takes the lower bound form $f(n) \geq c * (g(n))$.

⇨ $n^2/\log n \geq c * (n(\log n)^2)$

When n = 10 we get:

⇨ $10^2/\log(10) \geq c * (10(\log(10))^2)$
⇨ $100 \geq c * 10$

Looking at this we see that if c is any value greater than 10, this will not hold.

Meaning that this is just ***f(n) = O(g(n))*** d.) $f(n) = n^{1/2}$ | $g(n) = (\log n)^5$

Let's assume this takes the upper bound form: $f(n) \leq c * (g(n))$

⇨ $n^{1/2} \leq c * (\log n)^5$

When n = 1 we get:

⇨ $1^{1/2} \leq c * (\log(1))^5$
⇨ $1 \leq c * 0$

Since $1 \leq 0$ is false, this does not hold.

Now let's assume this takes the lower bound form: $f(n) \geq c * (g(n))$

⇨ $n^{1/2} \leq c * (\log n)^5$

When n = 10 we get:

⇨ $10^{1/2} \leq c * (\log(10))^5$
⇨ $10^{1/2} \leq c * 1$

When c is greater than 4 this will hold making this a lower bound.

This leaves us with $\boldsymbol{f(n) = \Omega(g(n))}$

e.) $f(n) = n2^n$ | $g(n) = 3^n$

Problem 2.)

a.) This algorithm appears to be a divide and conquer algorithm that will recursively compare values to get the smallest value of the given array.

b.) For the base case we have: $T(n) = 7$ if $n \leq 1$
For the rest, we see that the algorithm does the following:
- Set the k variable (cost of 7)
- Make the first recursive call (cost of n/2)
- Make the second recursive call (cost of n/2)
- Compare and return (cost of 4)

Adding these up we get: $T(n) = 2T(n/2) + 11$

T(n) = 7 if n ≤ 1

T(n) = 2T(n/2) + 11 if n > 1

c.)

Level	Level number	Total number of recursive executions at this level	Input size to each recursive execution	Work done by each recursive execution, excluding the recursive calls	Total work done by the algorithm at this level
Root	0	2^0	n	c	cn
One level below root	1	2^1	n/2	c	cn/2
Two levels below root	2	2^2	n/4	c	cn/4
Just above the base case level	$\log_2 n - 1$	$2^{\log_2 n - 1}$	n/(n-1)	c	cn/(n-1)
Base case level	$\log_2 n$	$2^{\log_2 n}$	n/n	c	$c\log_2 n$

d.) Based on the findings above, this algorithm is **$\Theta(\log n)$**

Problem 3.)

Level	**Level number**	**Total number of recursive executions at this level**	**Input size to each recursive execution**	**Work done by each recursive execution, excluding the recursive calls**	**Total work done by the algorithm at this level**
Root	0	7^0	n	c	cn
One level below root	1	7^1	n/8	c	cn/8
Two levels below root	2	7^2	n/64	c	cn/64
Just above the base case level	$n^{\log_8 7}$ - 1	7^$n^{\log_8 n - 1}$)	n/($8^{\log_8 n - 1}$)	c	cn/($8^{\log_8 n-1}$)
Base case level	$n^{\log_8 7}$	7^($n^{\log_8 n}$)	n/n	c	cn/($7^{\log_8 n}$)

$T(n) = cn * \sum_{i = 0 \text{ to } n -1} (7/8)^i + cn$

Using the provided result $\sum_{i = 0 \text{ to } \infty} x^i = 1/(1 - x)$ while $x < 1$, we get:

$T(n) = cn * (1/(1-x)) + cn$

Problem 4.)

$T(n) = 3T(n/3) + 5$; $T(1) = 5$

<u>Statement of what you have to prove:</u>

Let's assume that this complexity is O(n), we will need to try and prove that $T(n) \leq cn$

<u>Base case proof:</u>

Plugging the base case into the equation above:

- ⇨ $T(1) \leq c * 1$
- ⇨ $5 \leq c$

<u>Inductive hypotheses:</u>

Now let's assume $T(n/3) \leq c * (n/3)$.

<u>Inductive step:</u>

Using the set inequality above we can find $T(n/3) = 3T((n/3)/3) + 5 \leq c * (n/3)$

- $\Rightarrow$ $3T(n) + 5 \leq c * (n/3)$

<u>Value of c:</u>

<u>Problem 5.)</u>

$f(n) = O(s(n))$ and $g(n) = O(r(n))$ imply $f(n) - g(n) = O(s(n) - r(n))$

Let's assume that $f(n) = n^2$, $g(n) = n^2$, $s(n) = n^2 + n$, and $r(n) = n^2$.

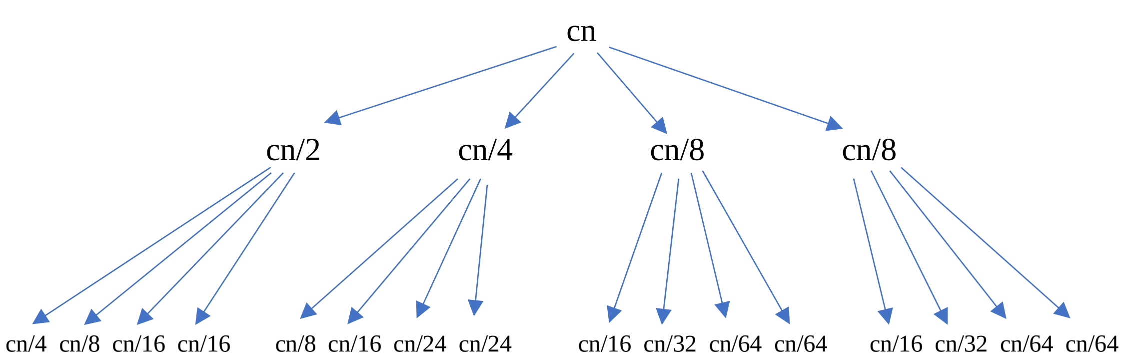
Plugging these values into the sets above we get:

- $\Rightarrow$ $n^2 = O(n^2 + n)$ and $n^2 = O(n^2 - n)$ (which works)
- $\Rightarrow$ $n^2 - n^2 = O((n^2 + n) - (n^2))$
- $\Rightarrow$ $0 = O(n^2 - n^2 + n)$
- $\Rightarrow$ $0 = O(n)$ (which does not hold)

Problem 6.)

$T(n) = T(n/2)+T(n/4)+T(n/8)+T(n/8)+n$; $T(1) = c$

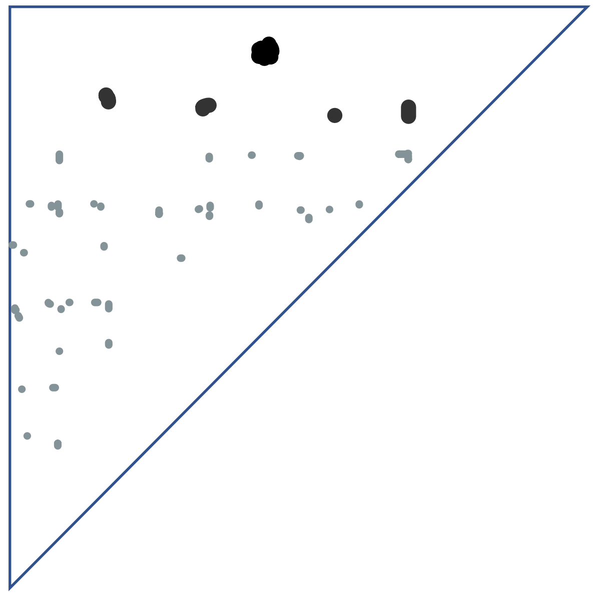
1.)



Work done at:
Level 0 = cn
Level 1 = cn
Level 2 = cn

2.) This tree should continue expanding further down the left side since the right continues to be divided by 8.

(Quite the masterful drawing)



3.) Depth of the tree at its shallowest part: $\log_8 n$

4.) Depth of the tree at its deepest part: $\log_2 n$

5.) Complexity guess: O(nlogn)

Problem 7.)

T(n) = T(n/2)+T(n/4)+T(n/8)+T(n/8)+n; T(1) = c

<u>Statement of what you have to prove:</u>

For this problem we will need to try and prove that T(n) ≤ cnlogn.

<u>Base case proof:</u>

Using T(1) = c we get:

- ⇨ T(1) ≤ c*1*log(1)
- ⇨ T(1) ≤ 0

<u>Inductive hypotheses:</u>

Let's expand by n/2 for each T(n):

- ⇨ T(n/2) = c*(n/2)*log(n/2)
- ⇨ T(n/4) = c*(n/4)*log(n/4)
- ⇨ T(n/8) = c*(n/8)*log(n/8)

Inductive step:

Now testing these for the original equation:

- ⇨ $T(n) \le c*(n/2)*\log(n/2) + c*(n/4)*\log(n/4) + c*(n/8)*\log(n/8) + n$
- ⇨ $T(n) \le c*[(n/2)*(\log n - \log 2)+(n/4)*(\log n - \log 4)+(n/8)*(\log n - \log 8)]+n$
- ⇨ $T(n) \le cn\log n * [(1/2)*(-1) + (1/4)*(-2) + (1/8)*(-4)] + n$
- ⇨ $T(n) \le cn\log n * [(-1/2) + (-2/4) + (-4/8)] + n$
- ⇨ $T(n) \le (-3/2)cn\log n + n$

Since $T(1) = c$, we get $T(1) \le (-3/2)c*(1)*\log(1) + 1$.

Which gives us: $c \le 1$

Problem 8.)

a.) $T(n) = 2T(99n/100) + 100n$
$a = 2$, $b = 100/99$, $f(n) = 100n$

Plugging these values into $\log_b(a)$ gives us: $\log_{100/99}(2) = 68.97$

Since $100n = O(n^{\log_{100/99}(2) - \varepsilon})$ for $\varepsilon > 0$, this is $T(n) = \Theta(n^{\log_{100/99}(2)}) = \mathbf{\Theta(n^{68.97})}$

b.) $T(n) = 16T(n/2) + n^3\lg n$
$a = 16$, $b = 2$, $f(n) = n^3\lg n$

Plugging these values into $\log_b(a)$ gives us: $\log_2(16) = 4$

Since $n^3\lg n = O(n^{\log_2(16) - \varepsilon})$ for $\varepsilon > 0$, this is $T(n) = \mathbf{\Theta(n^3\lg n)}$

c.) $T(n) = 16T(n/4) + n^2$
$a = 16$, $b = 4$, $f(n) = n^2$

Plugging these values into $\log_b(a)$ gives us: $\log_4(16) = 2$

Since $n^2 = O(n^{\log_2(16)})$, this is $T(n) = \mathbf{\Theta(n^2\lg n)}$

Problem 9.)

$T(n) = 2T(n-1) + 1$; $T(0) = 1$

Backwards substitution:

1.) $T(n-1) = 2T(n-1-1) + 1 = 2T(n-2) + 1$
$T(n) = 2(2T(n-2) + 1) + 1 = 4T(n-2) + 3$

$T(n-2) = 2T(n-2-1) + 1 = 2T(n-3) + 1$
$T(n) = 4(2T(n-3) + 1) + 3 = 8T(n-3) + 7$

$T(n-3) = 2T(n-3-1) + 1 = 2T(n-4) + 1$
$T(n) = 8(2T(n-4) + 1) + 7 = 16T(n-4) + 15$

The pattern created from the above equations is: $\mathbf{T(n) = 2^nT(n-n) + (2^n - 1)}$

2.) Simplifying the equation $T(n) = 2^nT(n-n) + (2^n - 1)$ we get:

- ⇨ $2^nT(0) + (2^n - 1)$
- ⇨ $2^n(1) + 2^n - 1$
- ⇨ $\mathbf{T(n) = 2^{n+1} - 1}$

3.) Plugging back in to the original relation:

LHS = $2^{n+1} - 1$
RHS = $2T(n-1) + 1$
LHS = RHS:

- ⇨ $2^{n+1} - 1 = 2T(n-1) + 1$
- ⇨ $2^{n+1} - 1 = 2(2^{n+1-1} - 1) + 1$
- ⇨ $2^{n+1} - 1 = 2^{n+1} - 1$

4.) The complexity of this algorithm is $\mathbf{O(2^n)}$

Forwards substitution:

$T(n) = 2T(n-1) + 1$, $T(0) = 1$

1.) $T(1) = 2T(1-1) + 1 = 2(1) + 1 = 3$
$T(2) = 2T(2-1) + 1 = 2T(1) + 1 = 7$
$T(3) = 2T(3-1) + 1 = 2T(2) + 1 = 15$

The pattern created from the above equations is $\mathbf{2^{n+1} - 1}$

2.) This is already simplified.

3.) LHS = $2^{n+1} - 1$
RHS = $2T(n - 1) + 1$
LHS = RHS:
⇨ $2^{n+1} - 1 = 2T(n - 1) + 1$
⇨ $2^{n+1} - 1 = 2(2^{n+1-1} - 1) + 1$
⇨ $2^{n+1} - 1 = 2^{n+1} - 1$

4.) The complexity of this algorithm is **$O(2^n)$**

Problem 10.)

$T(n) = T(n - 1) + n/2; T(1) = 1$

Using backward substitution:

$T(n - 1) = T(n - 2) + (n - 1)/2$

$T(n) = T(n - 2) + (n - 1)/2 + n/2$

$T(n - 2) = T(n - 3) + (n - 2)/2$

$T(n) = T(n - 3) + (n - 2)/2 + (n - 1)/2 + n/2$

$T(n - 3) = T(n - 4) + (n - 3)/2$

$T(n) = T(n - 4) + (n - 3)/2 + (n - 2)/2 + (n - 1)/2 + n/2$

This creates the pattern: $T(n) = T(n - m) + [(n - m + 1) + (n - m + 2) \ldots. + n]/2$

The pattern stops when $m = n - 1$ so we get: $T(n) = 1 + (1 + 2 + 3 + 4 \ldots. + n - 1)/2$

Using the arithmetic series pattern we get: $T(n) = 1 + [(n*(n + 1)/2) - 1]/2$

⇨ $T(n) = [(n*(n + 1)/2) + 1]/2$
⇨ **$T(n) = (n*(n + 1)/4) + ½$**

Problem 11.)

$T(n) = 2T(n/2) + 2n\log_2 n$ | $T(2) = 4$ | Guess is $O(n\log^2 n$

Using the master theorem:

$a = 2$, $b = 2$, $f(n) = 2n\log_2 n$

Plugging in a and b we get: $n^{\log_2 2} = n$

Since $2n\log_2 n = \Omega(n^{\log_2 2 + \varepsilon})$ for $\varepsilon > 0$, we get $T(n) = \Theta(f(n) = \Theta(2n\log_2 n)$

Problem 12.)

The Big Oh notation is specifically for setting an algorithm's upper bound, meaning it will be at most this complexity. Saying that an algorithm is at least $O(n^2)$ is like saying that an algorithm is at least, at most n^2 which is completely redundant. The phrase at least would be more fitting to the lower bound notation $\Omega()$ instead of the upper bound notation $O()$.