CS 624 - ANALYSIS OF ALGORITHMS

Assignment 1

Aravind Haridas – 02071139

1.

A. Loop Invariant: The loop invariant for the while loop is:

we already know that a loop variant is a property that holds before and after each iteration of the loop. for this algorithm, a suitable loop variant involves the variables x, y and p and it maintains the relationship that needs to hold true for the correctness of the power calculation, x^n.

$$r * y^p = x^n$$

(b) Proof of Correctness:

Initialization: Before the loop starts, r =1, y= x, and p =n.

Thus $r * y^p$

$1 * x^n$

$= x^n$

the loop invariant holds from start.

Maintenance: we need to show that the loop invariant holds at the beginning of each iteration of the loop. we need to show that it still holds after one iteration.

in each iteration we need to check if p is odd, we multiply *r* by *y.* this step corresponds to including than extra factor of y in the product on the left side of the loop invariant. Since *p* is odd, we can express *p* as 2*k*+1 for some integer *k*.

$$r \times y^{2k+1} = r \times y^{2k} \times y$$

Now, from the loop invariant, we know that,

$$r \times y^{2k} = x^{p/2}$$

$$r \times y^{2k} \times y = x^{p/2} \times y = x^p$$

So, the loop invariant still holds after the iteration.

Termination: in this algorithm the loop terminates, *p* becomes 0. At this point, the loop invariant becomes $r \times y^0 = r = x^n$

which is the desired result therefore by induction the algorithm is correct.

(c) Running Time: The running time of the algorithm is $O(\log n)$. In each iteration of the while loop, *p* is halved, and the loop continues until *p* becomes O . The number of iterations is determined by how many times *p* can be halved before reaching O , which is log *n.*

Hence the running time is logarithmic in terms of n.

CS 624 - ANALYSIS OF ALGORITHMS

2. (a)

A loop invariant is a condition that holds true before and after each iteration of the loop. for Cumulative Sums algorithm the loop invariant can be stated.

the loop invariant for the loop is:

$$R[j] = \sum_{i=1}^{j} A[i] \text{ for all } 2 \leq j \leq \text{length}[A]$$

b. proof of correctness: initialization Before the loop starts (*j*=2), the invariant holds trivially since

$$R[1] = A[1] \text{ and } \sum_{i=1}^{1} A[i] = A[1].$$

maintenance: Assume the loop invariant holds for 2<= j <=k for some k. we want that it holds for j = k+1.

$$R[k+1] = R[k] + A[k+1]$$

By the loop invariant we assume

$$R[k] = \sum_{i=1}^{k} A[i]$$

and hence we have this.

$$R[k+1] = \sum_{i=1}^{k} A[i] + A[k+1] = \sum_{i=1}^{k+1} A[i]$$

Hence, the loop invariant holds after the *k*+1-th iteration

Termination: After the loop terminates

$$R[\text{length}[A]] = \sum_{i=1}^{\text{length}[A]} A[i]$$

This matches the desired correctness property $R[n] = \sum_{i=1}^{n} A[i]$

By induction the algorithm is correct.

(c) Running Time: The running time of this algorithm Is $O(n)$ because it has a single for loop that iterates through the array of length *n* Each iteration performs constant time operations, so the overall time complexity is linear in terms of the array length.

CS 624 - ANALYSIS OF ALGORITHMS

3. (a) f(n) = O(g(n)) implies g(n) = O(f(n)).

This statement is false. If *f(n) = O(g(n)),* it means there exists a constant *c* such that for all n, *f(n) <= c. (g(n)).* any way that does not necessarily imply that g(n) is bounded above by a multiple of f(n), which is required for *g(n) = O(f(n)).*

(b). this statement is false.

$f(n) + g(n) = \Theta(\min(f(n), g(n)))$.

if we consider *f(n)* is significantly larger than *g(n).* then sum of *f(n) + g(n)* will be dominated by *f(n),* and thus it cannot be tightly bounded by the minimum of two.

this statement might be true but not always true.

(c). It's true if f*(n) = O(g(n)),* there exists constants c and n_0 such that for all n >= n_0, *f(n) <= c. (g(n)).* we can take log on both sides we will have *log(f(n)) <= log (c. (g(n))) = log(c) + log(g(n))*. since log (c) is constant *log*(f(n)) = *O(log(g(n))).*

(d)

It's true, if *f(n) = O(g(n)), then there* exits constants c and n_0 such that for all n >=n_0, f(n) <= c.g(n). raising 2 to the power of each side,

$2^{f(n)} <= 2^{c.g(n)}$

since we know 2^x

is an increasing function, and *c. g(n) can* be considered as g'(n) where

g'(n)= O(g(n)), we have

$2^{f(n)} = O(2^{g(n)})$.

CS 624 - ANALYSIS OF ALGORITHMS

4. we have four recurrences we need to solve by using master theorem or others.

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

This is in the form of $T(n) = aT(n/b) + f(n)$, where a=2 , b= 2 and f(n) =n4.

now comparing $n^{\log_b a}$ with f(n), we have $n^{\log_2 2} = n$.

since f(n) = n4 $= O(n^{log}2^{2-}\varepsilon)$ *for any* $\varepsilon > 0$

where ε is a positive constant, the Master theorem's case 1 applies.

Therefore, the upper bound is $O(n^4)$.

For the lower bound, we use the Omega notation. It's clear that f(n) = n^4 = $\Omega(n^{\log_2 2+\varepsilon})$ for $\varepsilon>0$. So, the lower bound is $\Omega(n^4)$.

Thus the tight bound for T(n) is $\Theta(n^4)$.

(b) $T(n) = T(7n/10) + n$ this recurrence doesn't fit the form of master theorem, so well use the substitution method.

let's assume above statement as these then we will have

$$T(n) = O(n).$$

$$T(n) \leq c \cdot \tfrac{7n}{10} + n$$
$$= (\tfrac{7c}{10} + 1) \cdot n$$

if we choose c>= 10/3, then *T(n) = O(n).*

for the lower bound T(n) >= n trivially so lower bound is $\Omega(n)$

thus , $T(n)=\Theta(n)$.

(f) $T(n) = 2T(n/4) + \sqrt{n}.$

this recurrence doesn't fir the form of the master theorem directly, so we'll use the substitution method.

now we can assume $T(n) = O(n^{\log_b a})$ then we have

$$T(n) \leq c \cdot \sqrt{n} + \sqrt{n}$$
$$= (c+1) \cdot \sqrt{n}$$

if we choose c>=1 then

$T(n)=O(\sqrt{n})$

For the lower bound $T(n) >= \sqrt{n}$ so $\Omega(n)$

is the lower bound then we have $T(n)=\Theta(\sqrt{n})$.

(g) $T(n) = T(n-2) + n^2$

These recurrences doesn't fit in the form of the master theorem directly so we will use the substitution method.

Let's assume $T(n)=O(n^2)$ then we will have

$T(n) \leq c \cdot (n-2)^2 + n^2$

Expanding $(n-2)^2$ and simplifying, we get

$T(n) \leq c \cdot (n^2 - 4n + 4) + n^2$

$= cn^2 - 4cn + 4c + n^2$

$= (c+1)n^2 - 4cn + 4c$

If we choose $c \geq 1$, then $T(n)=O(n^2)$.

For the lower bound, $T(n) \geq n^2$ Trivial so $\Omega(n^2)$ is the lower bound.

Thus $T(n)=\Theta(n^2)$.

5.

To prove that for a binary tree t, if mindepth(*t*) >= *n*, then countnil(*t*) >=2^n, we can use the induction on *n.*

Base case(n=0): for n=0, the tree t with mindepth(*t*) *>= 0, can be either Nil or a non-Nil node. If it is Nil, countnil(t) =1, which is >= 2^0. If t is not nil, countnil(t counts the nil nodes,* which are at least two childrens of t hence countnil(t) >=2, which also >=2^0.

Inductive step:

Assume the statement is true for some n =k, if mindepth(t) >= k, then countnil(t) >= 2^k.

Now, consider n= k+1 .a tree t with mindepth(t) >= k+1 must have a minimum depth of at least k in both subtrees. By the inductive hypothesis, each subtree has at least 2^k nil nodes.

Then t has at least $2^k + 2^k = 2^{k+1}$ Nil nodes.

Thus by induction if mindepth (t) >=n, then countnil(t)>= 2^n holds for all n>=0.