DAA HW 1

Problem 1

1. Purpose: Learn about Horner's rule and practice how loop invariants are used to prove the correctness of an algorithm. Please re-read Section 2.1 in our textbook and solve problem 2-3 Correctness of Horner's rule.
   The following code fragment implements Horner's rule for evaluating a polynomial

   $$P(x) = \sum_{k=0}^{n} a_k x^k$$
   $$= a_0 + x(a_1 + x(a_2 + \ldots + x(a_{n-1} + x a_n)\ldots))$$

   Given the coefficients $a_0, a_1 \ldots a_n$ and a value for x:

   y=0
   for i = n downto 0
   $\quad y = a_i + x.y$

   a. In terms of $\boldsymbol{\Theta}$ notation, what is the running time of this code fragment for Horner's rule?

Ans: The for loop is being executed n times so running time of above code fragment is $\boldsymbol{\Theta}$(n)

   b. Write pseudocode to implement the naive polynomial-evaluation algorithm that computes each term of the polynomial from scratch. What is the running time of this algorithm? How does it compare to Horner's rule?

Ans:

```
Naive-Horner()
      y = 0
      for i = 0 to n
            temp = 1
            for j = 1 to i
                  temp = temp * x
            y = y + a[i] * temp
      return y
```

Running time of this algorithm - $\boldsymbol{\Theta}(n^2)$
It is slower than Horner's rule because of the two for nested loops.

c. Consider the following loop invariant: At the start of each iteration of the for loop of lines 2–3,

$$y = \sum_{k=0}^{n-(i+1)} a_{k+i+1} x^k$$

Interpret a summation with no terms as equaling 0. Following the structure of the loop invariant proof presented in this chapter, use this loop invariant to show that, at termination,

$$y = \sum_{k=0}^{n} a_k x^k$$

Ans:

By considering the for loop -

for i = n downto 0

$$y = a_i + x.y$$

Initialization:

Initially i = n so upper bound of the summation is -1, Hence sum is 0 which implies that y = 0

Maintenance:

By using loop invariant, in the end of the iteration,

$$y = a_i + x \sum_{k=0}^{n-(i+1)} a_{k+i+1} x^k$$

$$y = a_i x^0 + x \sum_{k=0}^{n-(i+1)} a_{k+i+1} x^k$$

$$y = a_i x^0 + \sum_{k=0}^{n-(i+1)} a_{k+i+1} x^{k+1}$$

$$y = a_i x^0 + \sum_{k=0}^{n-i-1} a_{k+i+1} x^{k+1}$$

$$y = a_i x^0 + \sum_{k=1}^{n-i} a_{k+i} x^k$$

We get, $y = \sum_{k=0}^{n-i} a_{k+i} x^k$

Termination:

In the end we have i = 0,

We get, $y = \sum_{k=0}^{n} a_k x^k$

d. Conclude by arguing that the given code fragment correctly evaluates a polynomial characterized by the coefficients a0; a1, … ,an

Ans: By using the invariant of the loop, we have shown that a given code fragment is a sum that equals a polynomial of x with the coefficients a0, a1, … an.

Problem 2

2. Purpose: Practice counting basic operations and analyzing algorithms. Assume n > 0 and consider the following algorithm.

   (1) l ← 1
   (2) k ← 0
   (3) for i ← 1 to n do
   (4) l ← i - l
   (5) if k < n do
   (6) k ← i * i - 1
   (7) return(k, l)

   a. For n = 1,2,3,4,5 what values for k and l are returned in line 7? How many multiplications (“*”) does the algorithm perform for computing these values? How many subtractions (“-”) does the algorithm perform for computing these values?

n	1	2	3	4	5
Return value (k)	0	3	3	8	8
Return value (l)	0	2	1	3	2
# multiplications (“*”)	1	2	2	3	3
# subtractions (“-”)	1	2	3	4	5

   b. As a function of n, what is the value of k returned in line 7? Justify your results.

Ans: The value of k as a function of n is as follows -

$k = 0$ if n = 1

$(L(\sqrt{n}) + 1)^2 - 1$ for n > 1 Where L(x) = lower bound of x

If n = 2,

$k = (1 + 1)^2 - 1$

$k = 4 - 1$
$k = 3$

If n = 15,

$k = (3 + 1)^2 - 1$
$k = 16 - 1$
$k = 15$

c. As a function of n, what is the value of l returned in line 7? Justify your results.

Ans: The value of l as a function of n is as follows -

$l = f(n) = n/2 + 1 \quad if\ n\ is\ divisible\ by\ 2$
$\quad (n - 1)/2 \quad otherwise$

Let l = T(n),

From the code,

T(0) = 1
T(n) = n - T(n-1)

By substituting the values of n,

T(1) = 1 - 1
T(2) = (2 - 1) + 1
T(3) = (3 - 2) + 1 - 1
T(4) = (4 - 3) + (2 - 1) + 1
T(5) = (5 - 4) + (3 - 2) + 1 - 1
.
.
.
.

By induction,

T(n) = 1 + 1 + … (n/2) times + 1 if n is divisible by 2
1 + 1 + … (n-1)/2 times otherwise

Hence,

$l = f(n) = n/2 + 1 \quad if\ n\ is\ divisible\ by\ 2$
$\quad (n - 1)/2 \quad otherwise$

d. As a function of n, how many multiplications ("*") does the algorithm perform? Justify your results.

Ans: let numberOfMultiplications(n) = nMul(n)

$nMul(n) = 1$ if n = 1
$L(\sqrt{n}) + 1$ if n > 1 Where L(x) = Lower bound of x

If n = 2,

nMul(2) = 1 + 1
= 2

If n = 5,

nMul(2) = 2 + 1
= 3

e. As a function of n, how many subtractions ("-") does the algorithm perform? Justify your results.

Ans: let numberOfSubtractions(n) = nSub(n)

$nSub(n) = n$ for all n >= 1

As the for loop executes for n times and there is no restriction on subtraction so subtraction will also be executed n times.

If n = 2,

nSub(2) = 2

Problem 3

3. Rank the following functions by order of asymptotic growth; that is, find an arrangement g1 , g2 , … of the below functions with g1∈Ω(g2), g2∈Ω(g3) …. Mark asymptotically equivalent functions, i.e., gk ∈Θ(gk+1) by a "*". Justify your solution

$f1 = n^{0.91} - lg(n),\ f2 = 99n^{0.8},\ f3 = nlg(n) - 9n,\ f4 = lg(n!) + n^{0.9},\ f5 = n + 9/n^{0.9},$
$f6 = n^{0.9} lg(n^{0.8}),\ f7 = 9n^{0.9}$

Ans: $f3 = f4 > f5 > f6 > f1 > f7 > f2$

Explanation -

I. f3 = f4

This is because $lg(n!) = \Theta(nlg(n))$ and $n > n^{0.9}$

II. f3 > f5

This is because $nlg(n) \in \Omega(n)$

By using L'Hospital's rule,

$$\lim_{n \to \infty} \frac{f3}{f5} = \lim_{n \to \infty} \frac{f3'(n)}{f5'(n)}$$

$$L = \lim_{n \to \infty} \frac{nlg(n) - 9n}{n + 9/n^{0.9}}$$

By differentiating further,

$$L = \infty$$

Therefore, f3 = ω(f5)

III. f5 > f6

This is because $n^{0.9} lg(n^{0.8}) \in O(n)$

IV. f6 > f1,

By using L'Hospital's rule,

$$\lim_{n \to \infty} \frac{f6}{f1} = \lim_{n \to \infty} \frac{f6'(n)}{f1'(n)}$$

L $= \lim_{n \to \infty} \frac{0.8n^{0.9} lg(n)}{n^{0.91} - lg(n)}$

By differentiating further,

L $= \infty$

Therefore, f6 = ω(f1)

V. f1 > f7,

This is because $n^{0.91} > n^{0.9}$

VI. f7 > f2,

This is because $n^{0.9} > n^{0.8}$

Problem 4

4. Using the c, n0 definitions of Θ and ω, prove or disprove the following statements rigorously. Give values for c and n0 that will make your argument work.
   a. nlg(n)+lg(n)∈ω(n) (little-omega)
   b. nlg(n)+sqrt(n)+n∈Θ(n)

Ans:

a. nlg(n) + lg(n) ∈ ω(n) (little-omega)

Let, f(n) = nlg(n) + lg(n),

g(n) = n

Method 1 -

As lg(n) is always positive for n > 1,

$0 <= n <= nlg(n)$ for n >= 1

$0 <= n <= nlg(n) + lg(n)$ for n >= 1

$0 <= n < nlg(n) + lg(n)$ for n >= 2

$0 <= 1 * n < nlg(n) + lg(n)$

This is of the form $0 <= c.g(n) < f(n)$

Therefore, for c = 1 and n0 = 2, nlg(n) + lg(n) ∈ ω(n) is proved.

Method 2 -

As lg(n) is monotonically increasing function,

By using L'Hospital's rule,

$\lim_{n \to \infty} \frac{f(n)}{g(n)} = \lim_{n \to \infty} \frac{f'(n)}{g'(n)}$ if f(n) = ∞ and g(n) = ∞

Or f(n) = 0 and g(n) = 0

Here, f(n) = ∞ and g(n) = ∞,

Hence, by using L'Hospital's rule,

$$\lim_{n \to \infty} \frac{nlg(n) + lg(n)}{n} = \lim_{n \to \infty} \frac{f'(n)}{g'(n)}$$

$$L = \lim_{n \to \infty} \frac{lg(n) + \frac{n}{nln(2)} + \frac{1}{nln(2)}}{1}$$

$$L = \lim_{n \to \infty} \frac{nln(2)lg(n) + n + 1}{nln(2)}$$ (differentiating again)

$$L = \lim_{n \to \infty} \frac{ln(2)lg(n) + \frac{nln(2)}{nln(2)} + 1}{ln(2)}$$ by $\lim_{n \to \infty} \frac{f''(n)}{g''(n)}$

$$L = \lim_{n \to \infty} \frac{ln(2)lg(n) + 2}{ln(2)}$$

$$L = \lim_{n \to \infty} lg(n) + \frac{2}{ln(2)}$$

$$L = \infty$$ By substituting n = ∞

∴ Since L = ∞, nlg(n) + lg(n) ∈ ω(n) is proved

b. nlg(n) + sqrt(n) + n ∈ Θ(n)

Let, f(n) = nlg(n) + sqrt(n) + n,

g(n) = n

Method 1 -

As lg(n) is always positive for n > 1,

$0 <= n <= nlg(n)$ for n >= 1

$0 <= n < nlg(n) + sqrt(n) + n$ for n >= 1

This is of the form $0 <= c.g(n) <= f(n)$

Therefore, $for\ c = 1\ and\ n0 = 1,\ nlg(n) + sqrt(n) + n \in \omega(n)$ …………1

If we consider $nlg(n) + sqrt(n) + n \in O(n)$

then,

$$nlg(n) + sqrt(n) + n <= n$$
$$nlg(n) <= n$$
$$lg(n) <= 1$$

However, there is no constant c such that lg(n) < c for all n >= n0

Therefore, $nlg(n) + sqrt(n) + n \notin O(n)$ …………2

From 1 and 2,

$nlg(n) + sqrt(n) + n \notin \Theta(n)$ is proved.

Method 2 -

By using L'Hospital's rule,

$\lim_{n \to \infty} \frac{f(n)}{g(n)} = \lim_{n \to \infty} \frac{f'(n)}{g'(n)}$ if f(n) = ∞ and g(n) = ∞

Or f(n) = 0 and g(n) = 0

Here, f(n) = ∞ and g(n) = ∞,

Hence, by using L'Hospital's rule,

$$\lim_{n \to \infty} \frac{nlg(n) + n^{1/2} + n}{n} = \lim_{n \to \infty} \frac{f'(n)}{g'(n)}$$

$$L = \lim_{n \to \infty} \frac{lg(n) + \frac{n}{nln(2)} + \frac{1}{2n^{1/2}} + 1}{1}$$ (by differentiating again)

$$L = \lim_{n \to \infty} \frac{2n^{1/2}ln(2)lg(n) + 2n^{1/2} + ln(2) + 2ln(2)n^{1/2}}{2ln(2)n^{1/2}}$$

$$L = \lim_{n \to \infty} \frac{\frac{ln(2)lg(n)}{n^{1/2}} + \frac{2}{n^{1/2}} + \frac{1}{n^{1/2}} + \frac{ln(2)}{n^{1/2}}}{\frac{ln(2)}{n^{1/2}}}$$ by $\lim_{n \to \infty} \frac{f''(n)}{g''(n)}$

$$L = \lim_{n \to \infty} lg(n) + 1 + \frac{3}{ln(2)}$$

$$L = \infty$$ By substituting n = ∞

∴ Since L = ∞, nlg(n) + lg(n) ∈ ω(n) .

Hence nlg(n) + sqrt(n) + n ∉ Θ(n)

Problem 5

5. Describe a recursive O(lg(n)) algorithm which computes a 3n , given a and n. You may assume that a is a positive real number, and n is a positive integer, but do not assume that n is a power of 2. To avoid deductions, please provide (1) a textual description of the algorithm and, if helpful, flowcharts and pseudocode; (2) one worked example or diagram to illustrate how your algorithm works; (3) proof of the correctness of the algorithm; and (4) an analysis of the time complexity of the algorithm.

Ans:

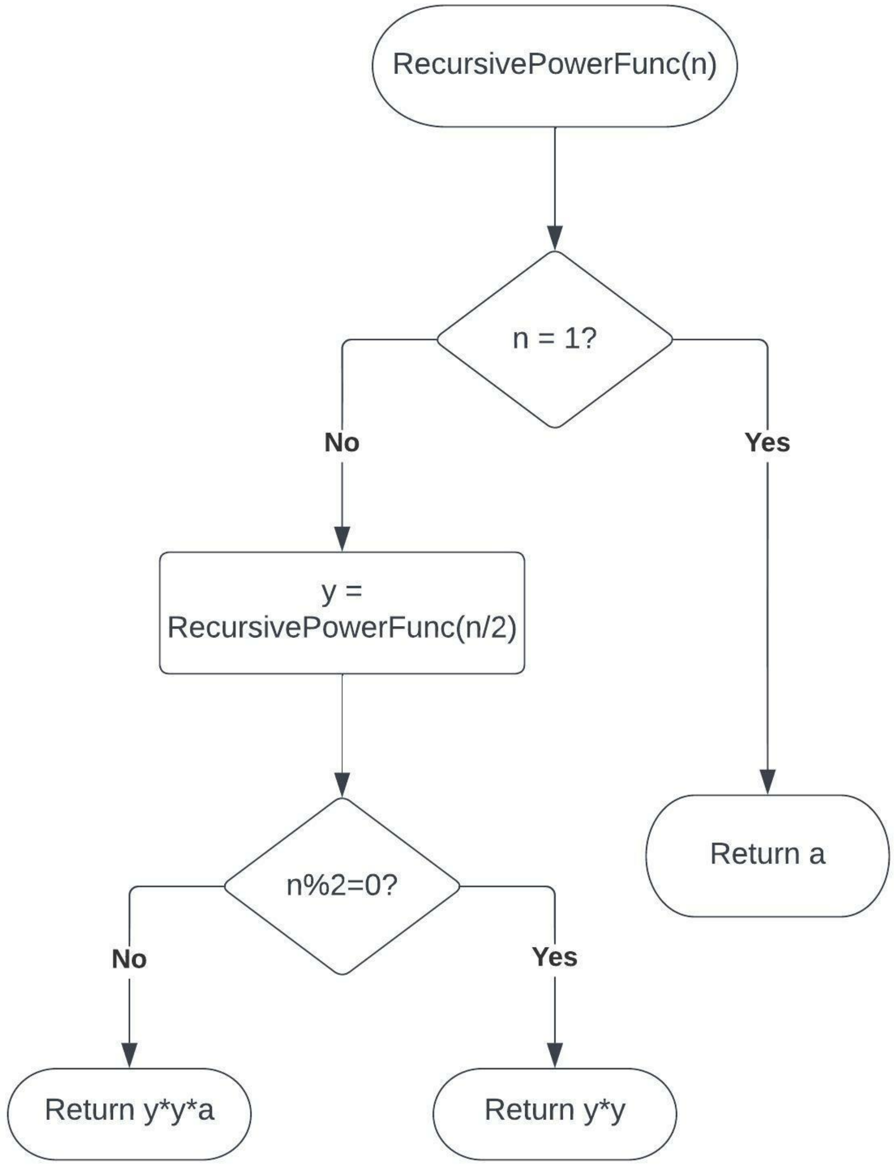
a. a textual description of the algorithm -
   i. By using the divide and conquer algorithm which uses a recursion, the time complexity will be O(lg(n)).
   ii. In this algorithm, every recursion call divides the number n to n/2 until we get n as 1.
   iii. Base case: If n==1, we return a
   iv. After recursion call, if n is divisible by 2 then return square of the result otherwise multiply ‘a’ with square of the result
   v. Pseudo code -

```
RecursivePowerFunc(a, n)
      if n == 1
            return a
      y = RecursivePowerFunc(a, n/2)
```

```
    if n%2 == 0
        return y * y
    else
        return y * y * a

CalculatePowerFunc(a, n)
    aToPowerN = RecursivePowerFunc(a, n)
    aToPower3N  = aToPowerN * aToPowerN * aToPowerN
    return aToPower3N
```

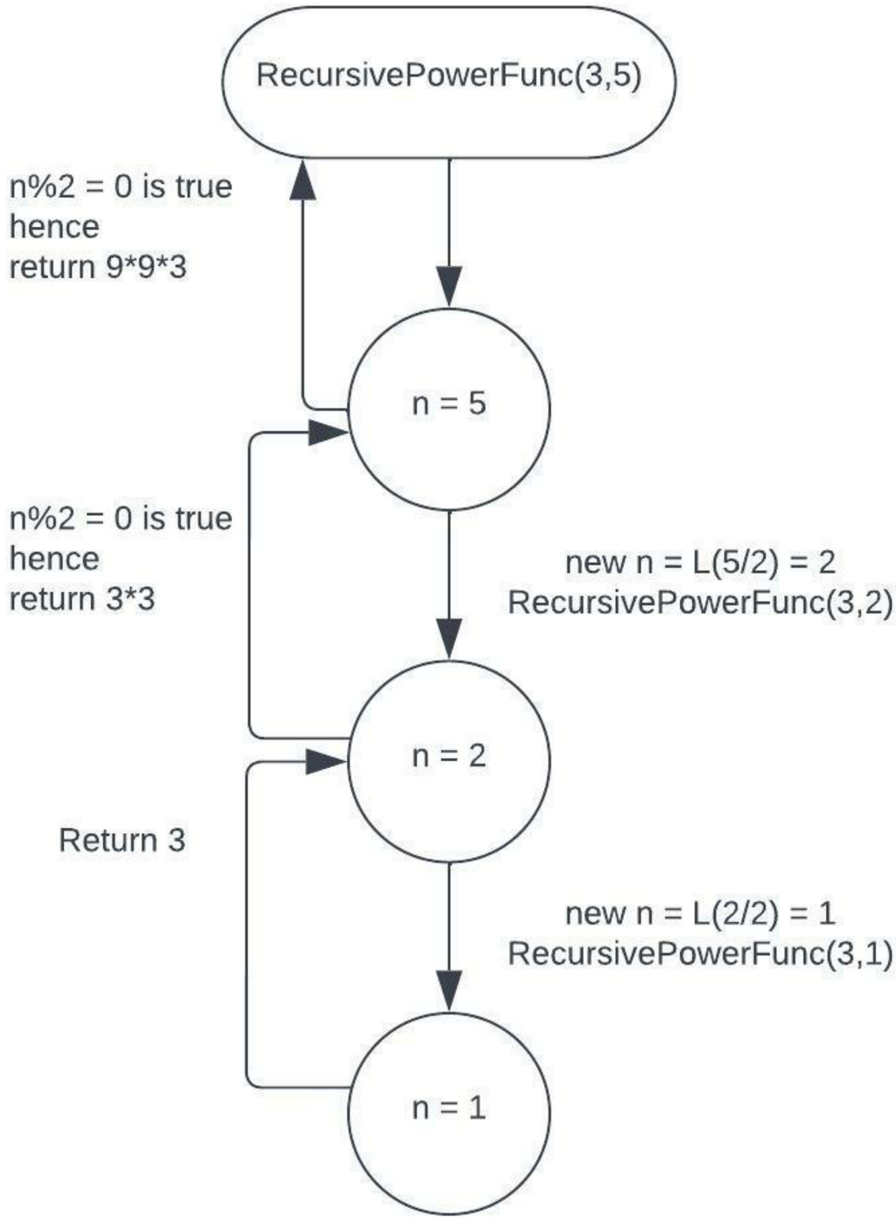
vi. Flowchart -



b. Example to illustrate the algorithm -

Let a = 3, n = 5

To calculate $a^{3n} = 3^{3*5}$,



Result of RecursivePowerFunc() = 9*9*3 = $a^{n} = 3^{5}$

Therefore, CalculatePowerFunc() returns the result = $a^{n} * a^{n} * a^{n}$

$= 3^{5} * 3^{5} * 3^{5}$

$= 3^{15} = a^{3n}$

c. Proof of correctness of the algorithm -

Precondition -

For each power p, $1 <= p <= n$

Postcondition -

RecursivePowerFunc terminates and return a^n for value n.

Initialization :

Initially, p = n, so if n != 1 then it calls a recursive function maintaining relation $1 <= p <= n$ and calculates a^n after recursive calls

Maintenance :

At any power p, as n is divided by 2 at every recursive function call, the relation $1 <= p <= n$ is maintained and calculates a^p

Termination :

At p =1, it returns a, so no further recursive function calls.
Hence maintaining the relation $1 <= p <= n$

d. Analysis of the time complexity of the algorithm -

By using divide and conquer algorithm,

Recursive calls divide n to n/2 hence following a tree structure

Time complexity = height of a tree
= lg(n)