

Session 1

Implementation of Simple Algorithms

This session is to implement small problems such as Euclid's Algorithm for GCD, polynomial expression evaluation through Horner's method, algorithms for evaluation of exponentiation and simple sorting algorithms. Selection sort begins by finding the least element in the array, which is moved to the first position. Then the second least element is searched for and moved to the second position in the array. This process continues until the entire array is sorted.

Objective

× Do not copy. Read for understanding and the viva

- Implement Euclid's Algorithm to find GCD
- Implement Horner's method for polynomial expression
- Compare performance of Horner's method with brute-force method
- Implement simple sorting algorithms
- Implement multiplication of two matrices

1. Euclid's Algorithm

Q1: Implement Euclid's Algorithm to find GCD(15265, 15) and calculate the number of times mod and assignment operations will be required.

What is Euclid's algorithm?

Euclid's algorithm is an efficient method for computing the Greatest Common Divisor (GCD) of two integers—the largest number that divides both of them without leaving a remainder.

It is based on the principle that the GCD of two numbers also divides their difference.

Specifically, the GCD of a and b is the same as the GCD of b and $a \bmod(b)$.

Algorithm

■ Write in lab record

The algorithm follows a simple recursive process:

- Given two numbers a and b (where $a > b$).
- Divide a by b and find the remainder r .
- Replace a with b and b with r .
- Repeat the process until the remainder is 0.
- The last non-zero remainder is the GCD.

Example

GCD of 48 and 18

- $48/18 = 2$ with a remainder of 12
- $18/12 = 1$ with a remainder of 6
- $12/6 = 2$ with a remainder of 0

The GCD is 6.



Lab record: write one language only. Pick yours once and every page opens on it; the other tabs are the same solution for comparison.

Python

Euclid.py

```
1 def gcd_with_stats(a, b):
2     """
3     Computes GCD and tracks modulo and assignment operations.
4     """
5     mod_count = 0
6     assign_count = 0
7
8     # Absolute values for mathematical consistency
9     a, b = abs(a), abs(b)
10    assign_count += 2
11
12    while b:
13        # Modulo operation
14        remainder = a % b
15        mod_count += 1
16
17        # Assignment operations (a = b, b = remainder)
18        a = b
19        b = remainder
20        assign_count += 2
21
22    return a, mod_count, assign_count
23
24 # Input values
25 val_a, val_b = 15265, 15
26 result, mods, assigns = gcd_with_stats(val_a, val_b)
27
28 print(f"GCD({val_a}, {val_b}) = {result}")
29 print(f"Modulo Operations: {mods}")
30 print(f"Assignment Operations: {assigns}")
```

2. (i) Horner's Rule

Q2 (i): Implement Horner's rule for evaluating polynomials $P(x) =$

$6x^6 + 5x^5 + 4x^4 - 3x^3 + 2x^2 + 8x - 7$ at $x=3$. Calculate how many times

- (a) Multiplications and addition operations will take
- (b) How many times the loop will iterate

What is Horner's Rule?

Horner's Rule is an efficient algorithm for evaluating polynomials that reduces the number of multiplications required by nesting the terms. Instead of calculating each power of x individually, it rewrites the polynomial in a recursive format.

Mathematical Logic

✖ Do not copy. Read for understanding and the viva

Note

To learn more about horner's rule, watch:

- [Horner's Method by Oscar Veliz](#)
- [Source code](#)

- For a polynomial $P(x) = a_n x^n + a_{(n-1)} x^{(n-1)} + \dots + a_1 x + a_0$, Horner's Rule transforms it into:

$$1 \quad P(x) = (\dots ((a_n x + a_{(n-1)})x + a_{(n-2)})x + \dots + a_1)x + a_0$$

LATEX

- Given our polynomial:

$$1 \quad P(x) = 6x^6 + 5x^5 + 4x^4 - 3x^3 + 2x^2 + 8x - 7$$

LATEX

- The nested form at $x = 3$ is:

$$1 \quad P(3) = ((((((6 * 3 + 5) * 3 + 4) * 3 - 3) * 3 + 2) * 3 + 8) * 3 - 7$$

LATEX

Algorithm

Write in lab record

- Initialize result as the coefficient of the highest degree term a_n
- For each subsequent coefficient from $a_{(n-1)}$ down to a_0 :
 - Multiply the current `result` by x .

- Add the next coefficient to the `result` .
- The final `result` is the value of the polynomial.

Example

✖ Do not copy. Read for understanding and the viva

Step	Operation	Current Result
Start	Initial value (an)	6
1	$(6 \times 2) + 2$	14
2	$(14 \times 2) + 5$	33
3	$(33 \times 2) + (-7)$	59

■
Lab record: write one language only. Pick yours once and every page opens on it; the other tabs are the same solution for comparison.

Python

horner-method.py

```
1 def horner_rule_analysis(coeffs, x):
2     # n is the degree (length of coeffs - 1)
3     # result starts as the first coefficient (6 in this case)
4     result = coeffs[0]
5
6     mult_count = 0
7     add_count = 0
8     loop_count = 0
9
10    # The loop starts from the second coefficient to the last
11    for i in range(1, len(coeffs)):
12        loop_count += 1
13
14        # Every step involves 1 multiplication and 1 addition
15        result = (result * x) + coeffs[i]
16
17        mult_count += 1
18        add_count += 1
19
20    return result, mult_count, add_count, loop_count
21
22 # P(x) = 6x^6 + 5x^5 + 4x^4 - 3x^3 + 2x^2 + 8x - 7
23 coeffs = [6, 5, 4, -3, 2, 8, -7]
24 x_val = 3
25
26 res, m, a, lp = horner_rule_analysis(coeffs, x_val)
27
28 print(f"Result: {res}")
29 print(f"(a) Multiplications: {m}, Additions: {a}")
30 print(f"(b) Loop Iterations: {lp}")
```

2. (ii) Brute Force Method

Q2 (ii): Apply a brute force method to implement the above polynomial expression from 2(i) and compare it with Horner's method in terms of number of multiplication operations.

What is Brute Force Method?

In Brute Force, for each term $a_k * x^k$, you calculate x multiplied by itself k times, then multiply by the coefficient a_k .

Algorithm

Write in lab record

- Initialize $total_sum = 0$.
- For each term $a_k x^k$:
 - Calculate $power_val = x^k$ (using a loop or power function).
 - Calculate $term_val = a_k * power_val$.
 - Add $term_val$ to $total_sum$.
- Return $total_sum$.

Example

✗ Do not copy. Read for understanding and the viva

Polynomial: $3x^2 + 2x + 5$ at $x=4$

- Term 1 ($3x^2$): $4 * 4 = 16$; $16 * 3 = 48$ (2 mults)
- Term 2 ($2x^1$): $2 * 4 = 8$ (1 mult)
- Term 3 (5): 5 (0 mults)
- Sum: $48 + 8 + 5 = 61$.

Total Multiplications: $2 + 1 = 3$. (Horner would only use 2).



Lab record: write one language only. Pick yours once and every page opens on it; the other tabs are the same solution for comparison.

Python

horner-method.py

```
1 def brute_force_analysis(coeffs, x):
2     n = len(coeffs) - 1 # The degree of the polynomial (6 in this case)
3     total_sum = 0
4     mult_count = 0
5     add_count = 0
6
7     # Loop through each coefficient and its index
8     for i, coeff in enumerate(coeffs):
9         # Calculate current power: e.g., for the first term (index 0), power
10         power = n - i
11
12         # Calculate x^power by multiplying x, 'power' number of times
13         current_pow_val = 1
14         for _ in range(power):
15             current_pow_val *= x
16             mult_count += 1 # Tracking multiplications for powers
17
18         # Multiply coefficient by the calculated power
19         if power > 0:
20             total_sum += coeff * current_pow_val
21             mult_count += 1 # Tracking multiplication by coefficient
22         else:
23             # Constant term (x^0) has no power or coefficient multiplication
24             total_sum += coeff
25
26         # Every coefficient after the first one involves an addition operation
27         if i > 0:
28             add_count += 1
29
30     return total_sum, mult_count, add_count
31
32 # P(x) = 6x^6 + 5x^5 + 4x^4 - 3x^3 + 2x^2 + 8x - 7
33 coeffs = [6, 5, 4, -3, 2, 8, -7]
34 res, m, a = brute_force_analysis(coeffs, 3)
35
36 print(f"Brute Force Result: {res}")
37 print(f"Total Multiplications: {m}")
38 print(f"Total Additions: {a}")
```

Complexity Comparison

Metric	Horner's Method	Brute Force Method
Multiplications	n (6)	$2n(n+1)+n$ (21)
Additions	n (6)	n (6)
Time Complexity	$O(n)$	$O(n^2)$

Brute Force method is significantly more expensive because the inner loop recalculates powers from scratch for every single term, whereas **Horner's Rule** reuses the result of the previous multiplication.

Comparison (Honer vs Brute)

Brute Force Multiplications:

- To calculate x^6 , you do 5 multiplications.
- Then 1 more to multiply by the coefficient 6.
- Totaling k multiplications for a term of degree k .
- For degree 6: $6+5+4+3+2+1 = 21$ multiplications.

Horner's Method Multiplications:

- As shown previously, this only requires 6 multiplications.

3. Matrix Multiplication

Q3: Implement matrix multiplication of two matrices $A[4,4]$ & $B[4,4]$ and calculate:

- how many times the inner-most and outer-most loops will run
- total number of multiplications and additions in computing the multiplication

Explanation

✗ Do not copy. Read for understanding and the viva

Matrix multiplication computes a resultant matrix C from two input matrices A and B such that:

$$1 \quad C[i][j] = \sum (A[i][k] \times B[k][j]) \quad \text{for } k = 0 \text{ to } n-1$$

LATEX

For $n \times n$ matrices, this requires **three nested loops**:

- **Outer loop (i)**: Iterates over rows of A
- **Middle loop (j)**: Iterates over columns of B
- **Inner-most loop (k)**: Computes the dot product between the i -th row of A and j -th column of B

Each inner iteration performs **one multiplication** ($A[i][k] * B[k][j]$) and one addition ($+=$ to accumulate into $C[i][j]$). The total **time complexity** is $O(n^3)$ and **space complexity** is $O(n^2)$ for the result matrix.

Algorithm & Pseudocode

```
1 INPUT: A[4][4], B[4][4]
2 OUTPUT: C[4][4]
3
4 Initialize C[4][4] with zeros
5 For i from 0 to 3:
6     For j from 0 to 3:
7         For k from 0 to 3:
8             C[i][j] += A[i][k] * B[k][j]
9 Return C
```

PY

Answer

■ Write in lab record

(i) Loop Execution Count

Loop Position	Iteration Count	Reason
Outer-most (i)	4 times	Runs once per row of A
Inner-most (k)	64 times	Executes for every (i, j) pair: $4 \times 4 \times 4 = 64$

(ii) Total Arithmetic Operations

Operation	Count	Explanation
Multiplications	64	1 per inner iteration → $n^3 = 4^3 = 64$
Additions	64	Each <code>c[i][j] += ...</code> performs 1 accumulation addition per inner step

Execution Trace (Conceptual)

For `n=4`:

- `i=0`: `j` runs 0..3 → `k` runs 4 times each → 16 inner executions
- `i=1`: same → 16 inner executions
- `i=2`: same → 16 inner executions
- `i=3`: same → 16 inner executions
- Total inner executions** = $16 \times 4 = 64$
- Each inner step: 1 mul + 1 add → 64 mul + 64 add



Lab record: write one language only. Pick yours once and every page opens on it; the other tabs are the same solution for comparison.

Python

matrix-multiplication.py

```
1 def matrix_multiply_4x4(A, B):
2     C = [[0]*4 for _ in range(4)]
3     outer_count = inner_count = mul_count = add_count = 0
4
5     for i in range(4):
6         outer_count += 1
7         for j in range(4):
8             for k in range(4):
9                 C[i][j] += A[i][k] * B[k][j]
10                inner_count += 1
11                mul_count += 1
12                add_count += 1
13
14     return C, outer_count, inner_count, mul_count, add_count
15
16 A = [[1,2,3,4], [5,6,7,8], [9,10,11,12], [13,14,15,16]]
17 B = [[1,0,0,1], [0,1,0,1], [0,0,1,1], [1,1,1,0]]
18
19 C, outer, inner, mul, add = matrix_multiply_4x4(A, B)
20
21 print("Result Matrix C:")
22 for row in C:
23     print([f"{x:4}" for x in row])
24 print(f"\nOuter-most loop runs: {outer} times")
25 print(f"Inner-most loop runs: {inner} times")
26 print(f"Total Multiplications: {mul}")
27 print(f"Total Additions: {add}")
```

4. Binary Exponentiation

Q4: Implement left to right and right to left binary exponentiation methods for the following problems:

(i) 4^{512}

(ii) 3^{31}

Calculate the followings for problems (i) and(ii)

- How many times the loops will be executed?
- How many times the multiplication and additions operations will be executed?

Explanation

✖ Do not copy. Read for understanding and the viva

Binary exponentiation, also known as exponentiation by squaring, is an efficient algorithm for computing large powers of a number. It reduces the number of multiplications needed by using the properties of exponents.

The algorithm can be implemented in two ways: left-to-right and right-to-left.

- **Left-to-Right:** Start from the highest bit of the exponent and work downwards.
- **Right-to-Left:** Start from the lowest bit of the exponent and work upwards.
- Both methods have a time complexity of $O(\log n)$ and require $O(1)$ space.
- The number of multiplications depends on the number of bits in the exponent and the number of 1s in the binary representation of the exponent.
- For example, for 4^{512} , the exponent 512 in binary is 1000000000, which has 1 bit set to 1. For 3^{31} , the exponent 31 in binary is 11111, which has 5 bits set to 1.



Lab record: write one language only. Pick yours once and every page opens on it; the other tabs are the same solution for comparison.

Python

binary-exponentiation.py

```
1 def right_to_left(base, exp):
2     """Computes base^exp using Right-to-Left binary exponentiation."""
3     result = 1
4     running_square = base
5     loops, multiplications = 0, 0
6
7     temp_exp = exp
8     while temp_exp > 0:
9         loops += 1
10        # If the bit is 1, multiply the result by current square
11        if temp_exp % 2 == 1:
12            result *= running_square
13            multiplications += 1
14
15        # Square the base for the next bit
16        if temp_exp > 1:
17            running_square *= running_square
18            multiplications += 1
19
20        temp_exp //= 2
21    print(f"After processing bit: result={result}")
22    return loops, multiplications
23
24
25 def left_to_right(base, exp):
26     """Computes base^exp using Left-to-Right binary exponentiation."""
27     result = 1
28     loops, multiplications = 0, 0
29
30     # Process bits from the Most Significant Bit to the Least
31     binary_str = bin(exp)[2:]
32     for bit in binary_str:
33         loops += 1
34         # Always square the result
35         result *= result
36         multiplications += 1
37
38         # If the bit is 1, multiply by the original base
39         if bit == "1":
40             result *= base
41             multiplications += 1
42    print(f"After processing bit: result={result}")
43    return loops, multiplications
```

```
44
45
46 def run_all():
47     problems = [(4, 512), (3, 31)]
48     for b, e in problems:
49         print(f"--- Problem: {b}^{e} ---")
50
51         r_loops, r_mults = right_to_left(b, e)
52         print(f"R2L Method: Loops={r_loops}, Multiplications={r_mults}")
53
54         l_loops, l_mults = left_to_right(b, e)
55         print(f"L2R Method: Loops={l_loops}, Multiplications={l_mults}\n")
56
57
58 if __name__ == "__main__":
59     run_all()
```

5. Bubble Sort

Q5: Implement Bubble Sort algorithm for the following list of numbers:

55 25 15 40 60 35 17 65 75 10

Calculate

- (i) a number of exchange operations
- (ii) a number of times comparison operations
- (iii) a number of times the inner and outer loops will iterate?

Explanation

✗ Do not copy. Read for understanding and the viva

Bubble Sort is a simple sorting algorithm that repeatedly steps through the list, compares adjacent elements and swaps them if they are in the wrong order. The process is repeated until the list is sorted. The algorithm gets its name because smaller elements “bubble” to the top of the list.

- The outer loop runs $n-1$ times, where n is the number of elements in the list.
- The inner loop runs $n-i-1$ times for each iteration of the outer loop, where i is the current iteration of the outer loop.

- The number of comparisons is the sum of the inner loop iterations, which is $n(n-1)/2$ in the worst case.
- The number of exchanges depends on the initial order of the elements. In the worst case (when the list is in reverse order), the number of exchanges is also $n(n-1)/2$.



Lab record: write one language only. Pick yours once and every page opens on it; the other tabs are the same solution for comparison.

Python

bubble-sort.py

```
1 def bubble_sort(numbers):
2     n = len(numbers)
3     # Counters for the requested metrics
4     exchanges = 0
5     comparisons = 0
6     outer_iterations = 0
7     inner_iterations = 0
8
9     # The outer loop runs 'n' times (once for each element)
10    for i in range(n):
11        outer_iterations += 1
12
13        # The inner loop runs from the start to the 'unsorted' portion.
14        # We subtract 'i' because the last 'i' elements are already sorted.
15        for j in range(0, n - i - 1):
16            inner_iterations += 1
17            comparisons += 1
18
19            # Comparison: Is the current element larger than the next?
20            if numbers[j] > numbers[j + 1]:
21                # Exchange: Swap the elements
22                numbers[j], numbers[j + 1] = numbers[j + 1], numbers[j]
23                exchanges += 1
24
25    return exchanges, comparisons, outer_iterations, inner_iterations
26
27
28 # Input data
29 data = [55, 25, 15, 40, 60, 35, 17, 65, 75, 10]
30 ex, comp, out, inn = bubble_sort(data)
31
32 print(f"Sorted: {data}")
33 print(f"Exchanges: {ex}, Comparisons: {comp}, Outer: {out}, Inner: {inn}")
```

6. Selection Sort

Q6: Implement Selection Sort algorithm on a similar set of data as in Q5 and

(i)perform similar operations on the above example

(ii) make a comparison between the two algorithms in terms of best case and worst case complexities.

Explanation

✖ Do not copy. Read for understanding and the viva

Selection Sort is a simple comparison-based sorting algorithm. It divides the input list into two parts: the sorted part at the left end and the unsorted part at the right end. Initially, the sorted part is empty and the unsorted part is the entire list. The algorithm repeatedly selects the smallest (or largest, depending on sorting order) element from the unsorted part and moves it to the end of the sorted part.

- The outer loop runs $n-1$ times, where n is the number of elements in the list.
- The inner loop runs $n-i-1$ times for each iteration of the outer loop, where i is the current iteration of the outer loop.
- The number of comparisons is the same as Bubble Sort, which is $n(n-1)/2$ in the worst case.
- The number of exchanges is $n-1$ in the worst case, as each element is moved at most once.



Lab record: write one language only. Pick yours once and every page opens on it; the other tabs are the same solution for comparison.

Python

selection-sort.py

```
1 def selection_sort_detailed(arr):
2     n = len(arr)
3     # Metric trackers
4     exchanges, comparisons = 0, 0
5     outer_iters, inner_iters = 0, 0
6
7     # Outer loop: Moves the boundary of the unsorted subarray
8     for i in range(n):
9         outer_iters += 1
10
11         # Step 1: Assume the first unsorted element is the smallest
12         min_index = i
13
14         # Step 2: Inner loop to find the actual minimum in the remaining unsorted
15         for j in range(i + 1, n):
16             inner_iters += 1
17             comparisons += 1
18             if arr[j] < arr[min_index]:
19                 # We found a new minimum; just update the index (no swap yet)
20                 min_index = j
21
22         # Step 3: Swap the found minimum with the first unsorted element
23         # This happens ONLY once per outer loop iteration.
24         if min_index != i:
25             arr[i], arr[min_index] = arr[min_index], arr[i]
26             exchanges += 1
27
28     return exchanges, comparisons, outer_iters, inner_iters
29
30
31 # Run the analysis
32 data = [55, 25, 15, 40, 60, 35, 17, 65, 75, 10]
33 ex, comp, out, inn = selection_sort_detailed(data)
34 print(f"Exchanges: {ex}, Comparisons: {comp}")
```

Comparison of Bubble Sort and Selection Sort

Metric	Bubble Sort	Selection Sort
Best Case Time Complexity	$O(n)$	$O(n^2)$
Worst Case Time Complexity	$O(n^2)$	$O(n^2)$
Number of Comparisons	$O(n^2)$	$O(n^2)$
Number of Exchanges	$O(n^2)$	$O(n)$

- Bubble Sort can be optimized to stop if no swaps are made, which gives it a best case time complexity of $O(n)$ when the list is already sorted. Selection Sort, on the other hand, always requires $O(n^2)$ comparisons regardless of the initial order of the elements. However, Selection Sort performs fewer exchanges than Bubble Sort, making it more efficient in terms of the number of swaps, especially for large lists.
- In summary, Bubble Sort is generally more efficient for small or nearly sorted lists, while Selection Sort can be more efficient for larger lists where the cost of swapping is high.
- For the given list of numbers, both algorithms will perform the same number of comparisons, but Bubble Sort may perform more exchanges than Selection Sort depending on the initial order of the elements.

[✎ Edit this page](#)

[< Previous](#)
Section 1 · Design and Analysis of Algorithms

Session 2 [Next >](#)