**AIR University, Islamabad**

**Islamabad Campus**

**Department of Creative Technologies**

|  |  |
| --- | --- |
| **Assignment No. 1:Mathematical Analysis of Iterative and Recursive Algorithms**  **Complexity Functions as Asymptotic NOTATION** | |
| **Subject:** Design and Analysis of Algorithm | **Instructor**: Dr Zulfiqar Ali |
| Class: | BS(SE)- 5A, BS(SE)- 5B |
| **Assigned Date:** 18th Oct, 2025 | **Due Date:** 30th Oct , 2025 |
| **Objective:** | |
| The objective of this assignment is to give you some practice regarding:   * **Growth Rate of a function** * **Asymptotic Notations( O, , )** * **Properties of Asymptotic Notations** * **Limits Rule for comparing Order of growth)** * **Mathematical Analysis of iterative and recursive algorithms** | |
| **CLO Mapping** | |
| CLO-4 : Analyze best, average, and worst-case behaviors of an algorithm. | |
| **Instructions** | |
| Read the following instructions before attempting to solve this assignment.   1. **Submission**: This is an individual assignment. Submit handwritten soft copy. (Word or PDF format). 2. **Understand the Assignment Thoroughly:**  * Make sure you fully grasp the requirements and objectives of the assignment. * Clarify any doubts with your instructor if necessary. * Read recommended books to gain a deeper understanding of various concepts in this assignment. * Try to consolidate key concepts and ideas from these questions.  1. **Ensure academic integrity and prevent plagiarism:**  **Try to make the solution by yourself and protect your work from other students. If I found the solution files of some students are same then I will reward zero marks to all those students.**  |  |  | | --- | --- | | **Category** | **Criteria** | | **Correctness of Complexity Analysis (30%)** | * 30%: Correctly identifies the time and space complexity in Big-O notation for all sections. * 20%: Mostly correct, but with some minor errors or omissions in complexity analysis. * 10%: Partially correct complexity analysis but contains significant errors. * 0%: Incorrect or missing complexity analysis. | | **Justification of Complexity (25%)** | * - 25%: Provides thorough and well-explained reasoning for time and space complexity calculations. * 15%: Adequate reasoning but lacks depth or completeness. * 10%: Some justification provided, but it is weak or incorrect. * 0%: No justification for the complexity analysis. | | **Comparison of Algorithms (20%)** | * 20%: Clear and accurate comparison of multiple algorithms in terms of efficiency, scalability, and trade-offs. * 15%: Comparison is mostly correct but lacks detail or misses key points. * 10%: Limited comparison with some inaccuracies or omissions. * 0%: No comparison or completely incorrect. | | **Edge Case and Worst-Case Analysis (15%)** | * 15%: Thorough analysis of best-case, average-case, and worst-case complexities, with edge case examples. * 10%: Provides analysis but lacks completeness (e.g., missing worst-case or edge cases). * 5%: Only one scenario (e.g., best-case) is analyzed. * 0%: No consideration of edge or worst cases. | | **Clarity and Structure of Explanation (10%)** | * 10%: Well-structured, with clear language and supporting diagrams or examples where needed. * 7%: Generally clear but could use more structure or examples. * 3%: Unclear or confusing, with minimal supporting details. * 0%: Poorly structured, hard to understand, or no explanations provided. |   In order to discourage copying/cheating, 30% marks may be awarded based on the marks obtained either in the corresponding quiz or a question(s) in the Midterm exam that will be used as multiplying factor. Hence, the final grades (out of 100) will be calculated as follows:  Obtain Marks = 40% of (marks obtain using  ***Submission marks (given by the letter grades shown above) + 30 \* multiplying factor*** | |

|  |
| --- |
| **Question # 1 (Visualization of notation)** |
| Let functions *f* and *g* are defined from set of Natural number N to set of positive real number R+. In each case draw the graphs of *f* and 2*g* on the same set of axes and find a number *n*0 so that *f (n)* ≤ 2*g(n)* for all *n > n*0.  **1.** *f (n)* = *n*2 + 10*n* + 11 and *g(n)* = *n*2  **2.** *f (n)* = *n*2 + 125*n* + 254 and *g(n)* = *n*2 |
| **Question # 2 (Use of definition)** |
| 1. Establish a big-O relationship, find witnesses c and *n0* such that |f (n)| ≤ c|g(n)| whenever *n > n*0. 2. *f* (n) = n log n 3. *f* (n) = n4 + 9n3 + 4n + 7 4. *f* (n) = n4 + 5 log n + 3n 5. *f* (n) = 2n + 17 is O(3n ). 6. Show that n log n is O(n2) but that n2 is not O(n log n). |
| **Question # 3 ( Use of Theorems)** |
| Give a big-O estimate for each of these functions. For the function g in your estimate f (n) is O(g(n)), use a simple function g of smallest order.  a) (n3+n2 log n)(log n+1) + (17 log n+19)(n3+2)  b) (2n + n2)(n3 + 3n)  c) (nn + n2n + 5n)(n! + 5n) |
| **Question # 4 ( Requires calculus)** |
| For each of these pairs of functions, determine whether *f* and g are asymptotic.   1. *f* (n) = n4 + log(3n8 + 7),   g(n) = (n2 + 17n + 3)2   1. *f* (n) = (n3 + n2 + n + 1)4,   g(n) = (n4 + n3 + n2 + n + 1)3.   1. f (n) = log(n2 + 1),   g(n) = log n   1. f (n) = 2n+3,   g(n) = 2n+7   1. f (n) =   g(n) =   1. f (n) =   g(n) =  Use limit rule to determine the dominating term in the given expression.  L’Hospital’s Rule |
| **Question # 5** |
| Find time complexity of the following algorithms. Use the most suitable notation among O, Ω, and to specify the time efficiency class of the given algorithm. |
| **Question # 6** |
| Use most appropriate notation among O, Ω, and Ɵ to compute the time complexity of the given program segments. |
| **Question # 7** |
| Fill in the following table   |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | **Problem** | **metric size for its**  **inputs** | **its basic operation** | **whether the basic operation count can**  **be different for inputs of the same size** | | | | **Its best case** | **Its average case** | **Its worst case** | | Computing the sum of n numbers |  |  |  |  |  | | Word processing |  |  |  |  |  | | Finding the largest element in a list of n numbers |  |  |  |  |  | | Euclid’s algorithm |  |  |  |  |  | | Solving linear equations |  |  |  |  |  | | Displaying a scene graphically |  |  |  |  |  | |
| **Question # 8** |
|  |
| **Question # 9** |
| Consider the following algorithm and answer the following questions against each algorithm.   1. What is its basic operation/Primitive Operation? 2. Whether the basic operation count can be different for inputs of the same size. 3. Setup the summation? How many times is the basic operation executed? 4. Use most appropriate notation among O, Ω, and Ɵ to compute the time complexity of the given algorithm/segments.       Where Merge function has time complexity O(n)    Consider modifying Algorithm selection sort as shown in Algorithm modselectionsort.     1. What is the minimum number of element assignments performed by Algorithm modselectionsort? When is this minimum achieved? 2. What is the maximum number of element assignments performed by Algorithm modselectionsort? Note that each interchange is implemented using three element assignments. When is this maximum achieved?      1. What is the minimum number of element comparisons performed by the algorithm? When is this minimum achieved? 2. What is the maximum number of element comparisons performed by the algorithm? When is this maximum achieved? 3. What is the minimum number of element assignments performed by the algorithm? When is this minimum achieved? 4. What is the maximum number of element assignments performed by the algorithm? When is this maximum achieved? 5. Express the running time of Algorithm Bubblesort in terms of the O and Ω notations. 6. Can the running time of the algorithm be expressed in terms of the Θ-notation? Explain.   Consider the following insertion sort algorithm.    Computing the average number of comparisons performed by Algorithm Insertionsort. |
| **Question # 10** |
| Consider the following Algorithm Test(*n*). Provide a line-by-line analysis and construct function T(*n*) that gives the runtime of this algorithm as a function of “*n*”. Also determine the Big-Oh of this algorithm.  Algorithm Test(*n*)  // Input:  // Output:   1. sum 2. **for** *i* to *n* 3. **if** ( *i*%2 ≠ 0) 4. sum sum + Test1( *i* ) + Test1( *i*+1) 5. **else**   sum sum + Test1 ( *i* ) + Test2( *i* )   1. **end for** 2. **return** sum  |  |  | | --- | --- | | Algorithm Test1(*a*)  // Input:  // Output:   1. **for** *i* to *n2* 2. a a \* i 3. **end for** 4. **return** a | Algorithm Test2(*b*)  // Input:  // Output:   1. **for** *i* to *n* 2. **for** *j* to *i* 3. b b \* (i +j) 4. **end for** 5. **end for** 6. **return** b | |
| **Question # 9** |
| Consider the following Algorithm Test(*n*). Provide a line-by-line analysis and construct function T(*n*) that gives the runtime of this algorithm as a function of “*n*”. Also determine the Big-Oh of this algorithm.  Algorithm Test(*n*)  // Input:  // Output:   1. **for** *i* to *n* 2. **if** ( *i*%2 = 0) 3. Test1( *i* )) 4. **else** 5. **for** *j* to *i* 6. Print(Test2( *i* )) 7. Print ( ) 8. **end for** 9. **end for**  |  |  | | --- | --- | | Algorithm Test1(*n*)  // Input:  // Output:   1. **for** *i* to *n* 2. **for** *j* to *i* 3. **Print (j)** 4. **end for** 5. **for** *k* to *3n* 6. **Print (j)** 7. **end for** 8. **end for** | Algorithm Test2(*a*)  // Input:  // Output:   1. sum 2. **for** *i* to *n* 3. sum sum + a \* *i* 4. **end for** 5. **return** sum | |
| **Question # 11** |
| Suppose you have algorithms with the five running times listed below. (Assume these are the exact running times.) How much slower does each of these algorithms get when you   1. Double the input size 2. Increase the input size by one?   Solution:  (a)   |  |  |  | | --- | --- | --- | | T(n) | T(2n) | Time Complexity increased by | |  |  |  | |  |  |  | |  |  |  | |  |  |  | |  |  |  |   (b)   |  |  |  | | --- | --- | --- | | T(n) | T(n+1) | Time Complexity increased by | |  |  |  | |  |  |  | |  |  |  | |  |  |  | |  |  |  | |
| **Question # 11** |
| Answer the following questions:  for (i= 1; i<= n; i++)  for(j=1; j<=n; j++)  for(k =1; k<=n; k++)  //five assignment statement  Suppose n = 3, how many times are the five assignments executed? \_\_\_\_\_\_  i. Does one loop depend on another?\_\_\_\_\_  ii. What is the complexity of the segment?  for (i=10; i<=n; i++)  {    for(j = 15; j<=n; j++)  for(k = 1; k <= n; k++)  //five assignments  }  Determine the complexity of the segment.  for(i = 1; i<= n; i++)  {  for(j = 1; j <= n; j++)  {  //20 assignments  }  for(j = 1; j<=n; j++)  if( j%2 == 1)  for (k = 1; k <=n; k++)  {  // five instructions  }  }  Notice the 20 assignments are in a doubly nested loop and the five instructions are in a triple loop.  Suppose n = 10  i. How many times will the 20 assignments be executed?\_\_\_\_\_\_\_\_\_\_\_\_\_  ii. How many times will the five instructions be executed? \_\_\_\_\_\_\_\_\_\_\_\_\_ iii. Now determine the complexity of the segment  for (i=1; I <= n. i++)  {  for (j = I; j <= n , j++)  {  *//*Six assignment statement  }  if (i % 2 == 1)  {  *//* Four assignment statement  }  }  What is the complexity?  for (i=1; i <= n; i++)  for (j = 1; j <= m: j++)  for (k =1; k <= n: k++)  *//* Two assignment instructions  If n = 3 and m = 4 how many times are the two assignment statements executed?  i = 1;  while (i <= n)  {  *//* Three assignment statements  j = 17;  while (j <= 100)  {  *//* Two assignments  j++;  }  i++;  }  i. How many times are the three assignments executed? \_\_\_\_\_\_\_\_\_  ii. How many times are the two assignments executed? \_\_\_\_\_\_\_\_\_\_  iii. What is the order (complexity) of the loop?  i = n;  do  {  *//* Three assignments  j =1;  while (j <= n)  {  *//*Two assignments  j++;  }  i--;  }while (i >= 1);  i. How many times are the three assignments executed? \_\_\_\_\_\_\_\_\_  ii. How many times are the two assignments executed? \_\_\_\_\_\_\_\_\_\_  iii. What is the order (complexity) of the loop?  i = 1;  while (i <= n)  {  //Three assignments  j = 1;  while (j <= n)  {  //Two assignments  j = j\*2;  }  i++;  }  Let n = 8.  i. How many times are the three assignments executed? \_\_\_\_\_\_  ii. How many times will the inner loop execute? \_\_\_\_\_\_  iii.What will the values of j be for each execution of the inner loop\_\_\_\_\_\_\_\_\_\_\_\_  iv. Now determine the complexity for arbitrary n?  i =n;  while(i≥1)   {  //Three assignments  j = 1;  while (j <= n)  {  *II*Two assignments  j = j + 2;  }  i = i / 3;  }  Determine the complexity for arbitrary n? |
| **Question # 12** |
| Set up a recurrence relation for the given functions. Solve recurrence relations using Recursion Tree Method.  Use appropriate asymptotic notation among O, Ω, and  to specify the time efficiency class of the given function.   |  |  | | --- | --- | | Test(n)  {      if(n == 1)  return  else  {  **for** (i = 1; i < n; i++)       {          Print(i)      }  Test(n/2)  Test(n/4)  Test(n/8)  }  } | Test(n)  {      if(n = 0)  return  else  {  **for** (i = 1; i < n; i= i\*2)      {          Print(i)     }  Test(n-1)  }  } | | Test(n)  {      if(n = 0)  return  else  {  **for** (i = 1; i <= n; i++)      {  **for** (j = 1; j < n; j += i)          {              // Some O(1) task          }      }  **for** (k = 1; k < 5; k++)  {  Test(n/2)  }  }  } | Test(n)  {  if(n == 0)  return  else  {  for( i = n/2; i <= n; i++)  for( j = 1; j+n/2 <= n; j++)  for( k = 1; k <= n; k=k\*2)  Print(k)  Test (n/2)  }  } | | Test(n)  {      if(n = 0)  return  else  {  **for** (i = 1; i <= n; i++)      {                // Some O(1) task      }    Test(n/3)  Test(n/3)  Test(n/3)  Test(n/3)  }  } | Test(n)  {      if(n = 0)  return  else  {  **for** (i = 1; i <= n; i++)       {                // Some O(1) task       }    Test(n-1)  Test(n/2)  }  } | | Test(n)  {      if(n = 0)  return  else  {  **O(1) Task**  Test(n-1)  Test(n-1)    }  } | Test(n)  {      if(n = 0)  return  else  {  **for** (i = 1; i <= n; i++)       {                // Some O(1) task       }    Test(n/3)  Test(2n/3)  }  } | |
| **Question # 13** |
| Consider the following algorithm.   1. Set up a recurrence relation for the number of times the algorithm’s basic operation is executed. 2. Solve the recurrence relation that you obtained in part (i) using:    1. Iteration Method    2. Recursion Tree Method    3. Substitution Method   **ALGORITHM** Search(A[*l*….*r*], *l*, *r*, *key*)  // A recursive search function.  // Input: An Array A[*l*..*r*]  // Output: It returns location of key in given array A, otherwise -1  {  **if** (*r* ≥ *l*) then     {  *m*1 = *l* + (*r* - *l*)/3;  *m*2 = m1 + (*r* - *l*)/3;  *// If key is present at the m1*  **if** (A[*m*1] == *key*)  **then**  **return** m1;  *// If key is present at the m2*  **if** (A[*m*2] == key)  then  **return** m2;  *// If key is present in left one-third*  **if** (A[*m*1] > *key*) **then**  **return** Search (A, *l*, m1-1, *key*);  *// If key is present in right one-third*          if (A[*m*2] < *key*) **then**  **return** Search (A, m2+1, *r*, *key*);  *// If key is present in middle one-third*  **return** Search (A, *m*2+1, *m*2-1, *key*);     }  *// We reach here when element is not present in array*  **return** -1;  } |
| **Question # 14** |
| Consider the following algorithm.   1. Set up a recurrence relation for the number of times the algorithm’s basic operation is executed. 2. Solve the recurrence relation that you obtained in part (i) using    1. Iteration Method    2. Recursion Tree Method    3. Maser Theorem |
| **Question # 15** |
| 1. Give asymptotic upper and lower bounds for T (n) of the following recurrence relation.      1. Use a recursion tree to give an asymptotically tight solution to the recurrence 2. The recurrence *T*(*n*) = 7*T* (*n*/2)+*n*2 describes the running time of an algorithm *A*. A competing algorithm *A*′ has a running time of *T’*(*n*) = *a T’*(*n*/4) + *n*2. What is the largest integer value for *a* such that *A*′ is asymptotically faster than *A*? 3. Can the master method be applied to the recurrence T (n) = 4T(n/2) + n2 lg n? Why or why not? Give an asymptotic upper bound for this recurrence. |
| **Question # 16** |
| State the general formula for the following recurrence of the form:   |  |  | | --- | --- | | 1 |  | | 2 |  | | 3 |  | |