Module-2

Mark

s

Bfs of unlinked tree:

<https://www.codingninjas.com/codestudio/library/bfs-in-disconnected-graph>

Questions

1. Explain general method of divide and conquer problem. 8

1. Apply binary search algorithm for following array elements: 14

3,10,15,20,35,40,60. And analysis the same.

1. Construct the algorithm for search and insert an element in a Binary 7

search tree. Tabulate the best, worst, and average-case complexity of the binary search tree for each operation

1. Construct the algorithm for the Breadth-First Search (BFS) 7

traversal of a directed graph. Mention the best, worst, and average- case complexity of the BFS

1. Construct the algorithm for Depth-First Search (DFS) traversal of a 8

directed graph. Mention the best, worst, and average-case complexity of the DFS

1. List Difference between BFS and DFS 7
2. Construct the binary search tree for the following data elements: 7

45, 15, 79, 90, 10, 55, 12, 20, 50

Trace the algorithm which you have constructed in 3a for every insertion and depict the final tree

1. Design an algorithm to sort the n number of elements using the 8

divide and conquer technique and provide the time complexities for best average, and worst case.

1. Sort the following elements using the above-designed divide and 8

conquer algorithm 100, 25, 98, 54, 79, 64, 84, 26, 48 and16. Trace the algorithm for each and every process

1. A man has 10 varieties of fish: 7

fish1, fish2 …… fish10 namely, in his aquarium. The number of fishes from fish1, fish2…. fish10 are as follows:

48, 56, 24, 96, 68, 59, 71, 84, 99, and 35.

Apply the divide and conquer methodology to find the name of the fish which is maximum and minimum in the aquarium

1. Suggest and design the suitable algorithm for the above problem 8

and also provide the time complexity for that algorithm

1. Discuss Quick Sort Algorithm and Explain it with example. Derive 14

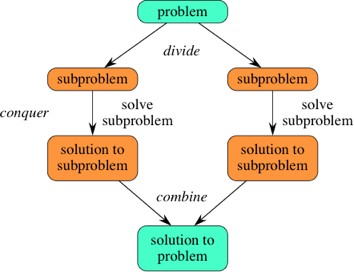
Worst case and Average Case Complexityi.

1. Explain Strassen’s matrix multiplication. Evaluate it’s efficiency. 14

Q1. Explain general method of divide and conquer problem.

**divide-and-conquer**, breaks a problem into subproblems that are similar to the original problem recursively and recursively solves the subproblems, and finally combines the solutions to the subproblems to solve the original problem. Because divide-and-conquer solves subproblems recursively, each subproblem must be smaller than the original problem, and there must be a base case for subproblems. You should think of a divide-and-conquer algorithm as having three parts:

1. **Divide** the original problem into a set of subproblems.
2. **Conquer:** Solve every subproblem individually, recursively.
3. **Combine:** Put together the solutions of the subproblems to get the solution to the whole problem.



Generally, we can follow the **divide-and-conquer** approach in a three-step process.

**Examples:** The specific computer algorithms those based on the Divide & Conquer approach are:

1. Maximum and Minimum Problem
2. Binary Search
3. Sorting (merge sort, quick sort)
4. Tower of Hanoi.0o,

# Q2. Apply binary search algorithm for following array elements 3,10,15,20,35,40,60. And analysis the same.

# Analysis of Binary Search:

Binary Search can be analysed with the best, worst, and average case number of comparisons. Now, let's see the time complexity of Binary search in the best case, average case, and worst case. We will also see the space complexity of Binary search.

### **1. Time Complexity**

|  |  |
| --- | --- |
| **Case** | **Time Complexity** |
| **Best Case** | O(1) |
| **Average Case** | O(logn) |
| **Worst Case** | O(logn) |

* **Best Case Complexity -** In Binary search, best case occurs when the element to search is found in first comparison, i.e., when the first middle element itself is the element to be searched. The best-case time complexity of Binary search is **O(1).**
* **Average Case Complexity -** The average case time complexity of Binary search is **O(logn).**
* **Worst Case Complexity -** In Binary search, the worst case occurs, when we have to keep reducing the search space(array) till it has only one element. The worst-case time complexity of Binary search is **O(logn).**

### **2. Space Complexity**

|  |  |
| --- | --- |
| **Space Complexity** | O(1) |

* The space complexity of binary search is O(1).

Exam point of view ;

[Link:](https://www.javatpoint.com/binary-search)

Algorithm:

|  |
| --- |
| Binary\_Search(a, lower\_bound, upper\_bound, val) // 'a' is the given array, 'lower\_bound' is the index of the first array element, 'upper\_bound' is the index of the last array element, 'val' is the value to search  Step 1: set beg = lower\_bound, end = upper\_bound, pos = - 1  Step 2: repeat steps 3 and 4 while beg **<**=end  Step 3: set mid = (beg + end)/2  Step 4: if a[mid] = val  set pos = mid  print pos  go to step 6  else if a[mid] **>** val  set end = mid - 1  //bcz need to be searched value will be in lower half of the array  else  set beg = mid + 1  [end of if]  [end of loop]  Step 5: if pos = -1  print "value is not present in the array"  [end of if]  Step 6: exit |
| 1. **class** BinarySearch { 2. **static** **int** binarySearch(**int** a[], **int** beg, **int** end, **int** val) 3. { 4. **int** mid; 5. **if**(end >= beg) 6. { 7. mid = (beg + end)/2; 8. **if**(a[mid] == val) 9. { 10. **return** mid+1;  /\* if the item to be searched is present at middle 11. \*/ 12. } 13. /\* if the item to be searched is smaller than middle, then it can only 14. be in left subarray \*/ 15. **else** **if**(a[mid] < val) 16. { 17. **return** binarySearch(a, mid+1, end, val); 18. } 19. /\* if the item to be searched is greater than middle, then it can only be 20. in right subarray \*/ 21. **else** 22. { 23. **return** binarySearch(a, beg, mid-1, val); 24. } 25. } 26. **return** -1; 27. } 28. **public** **static** **void** main(String args[]) { 29. **int** a[] = {8, 10, 22, 27, 37, 44, 49, 55, 69}; // given array 30. **int** val = 37; // value to be searched 31. **int** n = a.length; // size of array 32. **int** res = binarySearch(a, 0, n-1, val); // Store result 33. System.out.print("The elements of the array are: "); 34. **for** (**int** i = 0; i < n; i++) 35. { 36. System.out.print(a[i] + " "); 37. } 38. System.out.println(); 39. System.out.println("Element to be searched is: " + val); 40. **if** (res == -1) 41. System.out.println("Element is not present in the array"); 42. **else** 43. System.out.println("Element is present at " + res + " position of array"); 44. } 45. } |

Q3. Construct the algorithm for search and insert an element in a Binary search tree. Tabulate the best, worst, and average-case complexity of the binary search tree for each operation. 7M

Searching Algorithm Input: key elements

Output: return the elements if found, otherwise returns null

struct node\* search(int data){ struct node \*current = root; printf("Visiting elements: ");

while(current->data != data){ if(current != NULL) {

printf("%d ",current->data);

//go to left tree if(current->data > data){

current = current->leftChild;

} //else go to right tree else {

current = current->rightChild;

}

//not found

if(current == NULL){ return NULL;

}

}

}

return current;

}

Inserting Algorithm

Input: key elements

Output: insert key elements in the existing BST or create a new BST if the tree is empty

void insert(int data) {

struct node \*tempNode = (struct node\*) malloc(sizeof(struct node)); struct node \*current;

struct node \*parent;

tempNode->data = data; tempNode->leftChild = NULL; tempNode->rightChild = NULL;

//if tree is empty if(root == NULL) {

root = tempNode;

} else {

current = root; parent = NULL;

while(1) {

parent = current;

//go to left of the tree if(data < parent->data) {

current = current->leftChild;

//insert to the left

if(current == NULL) {

parent->leftChild = tempNode; return;

}

} //go to right of the tree else {

current = current->rightChild;

//insert to the right if(current == NULL) {

parent->rightChild = tempNode; return;

}

}

}

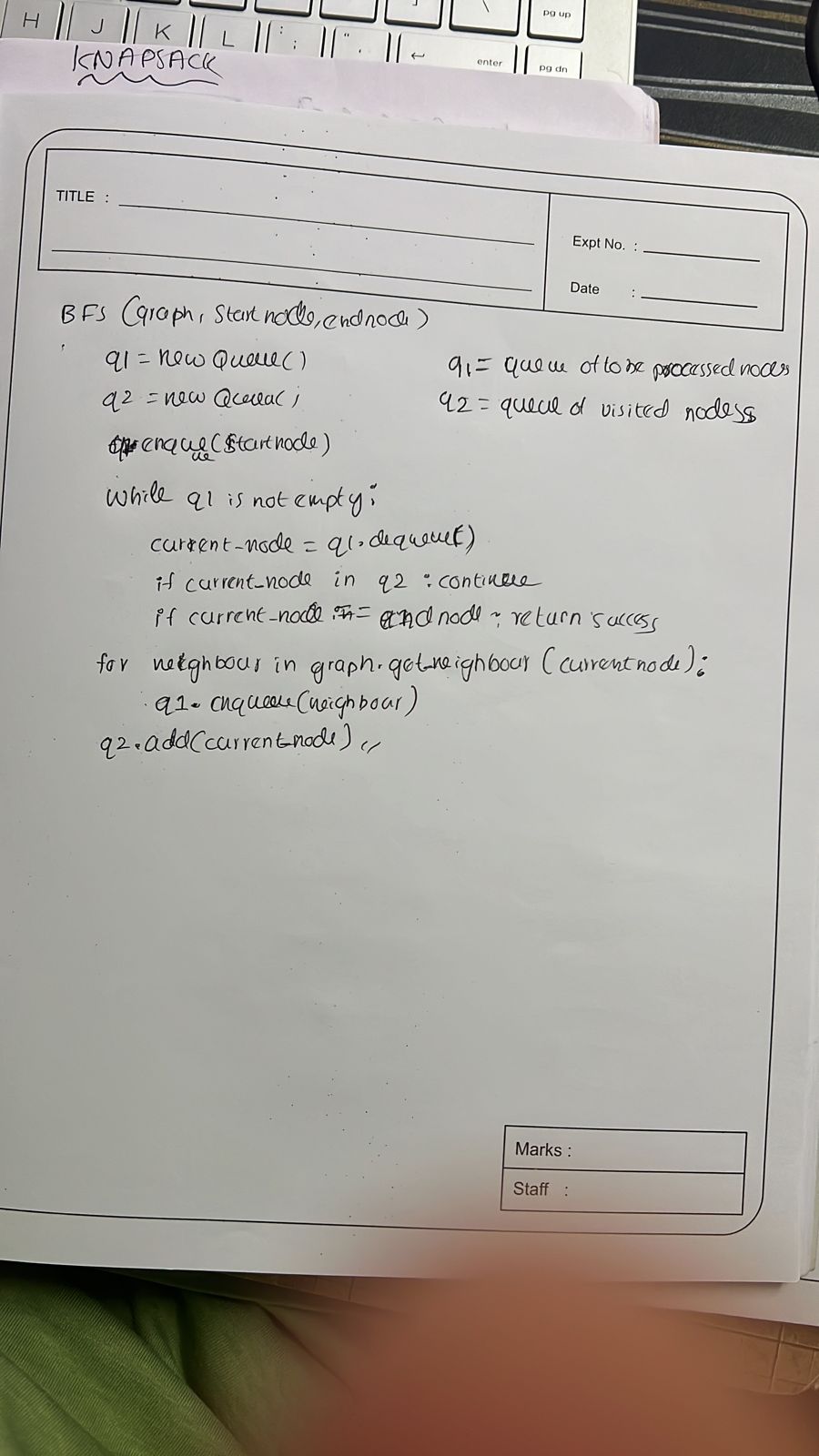
}

}

|  |  |  |  |
| --- | --- | --- | --- |
| **Operatio ns** | **Best case time complexity** | **Average case time complexity** | **Worst case time complexity** |
| **Insertion** | O(log n) | O(log n) | O(n) |
| **Deletion** | O(log n) | O(log n) | O(n) |
| **Search** | O(log n) | O(log n) | O(n) |

Q4. Construct the algorithm for the Breadth-First Search (BFS) traversal of a directed graph. Mention the best, worst, and average-case complexity of the BFS.7M

**Input −** The list of vertices, and the start vertex.

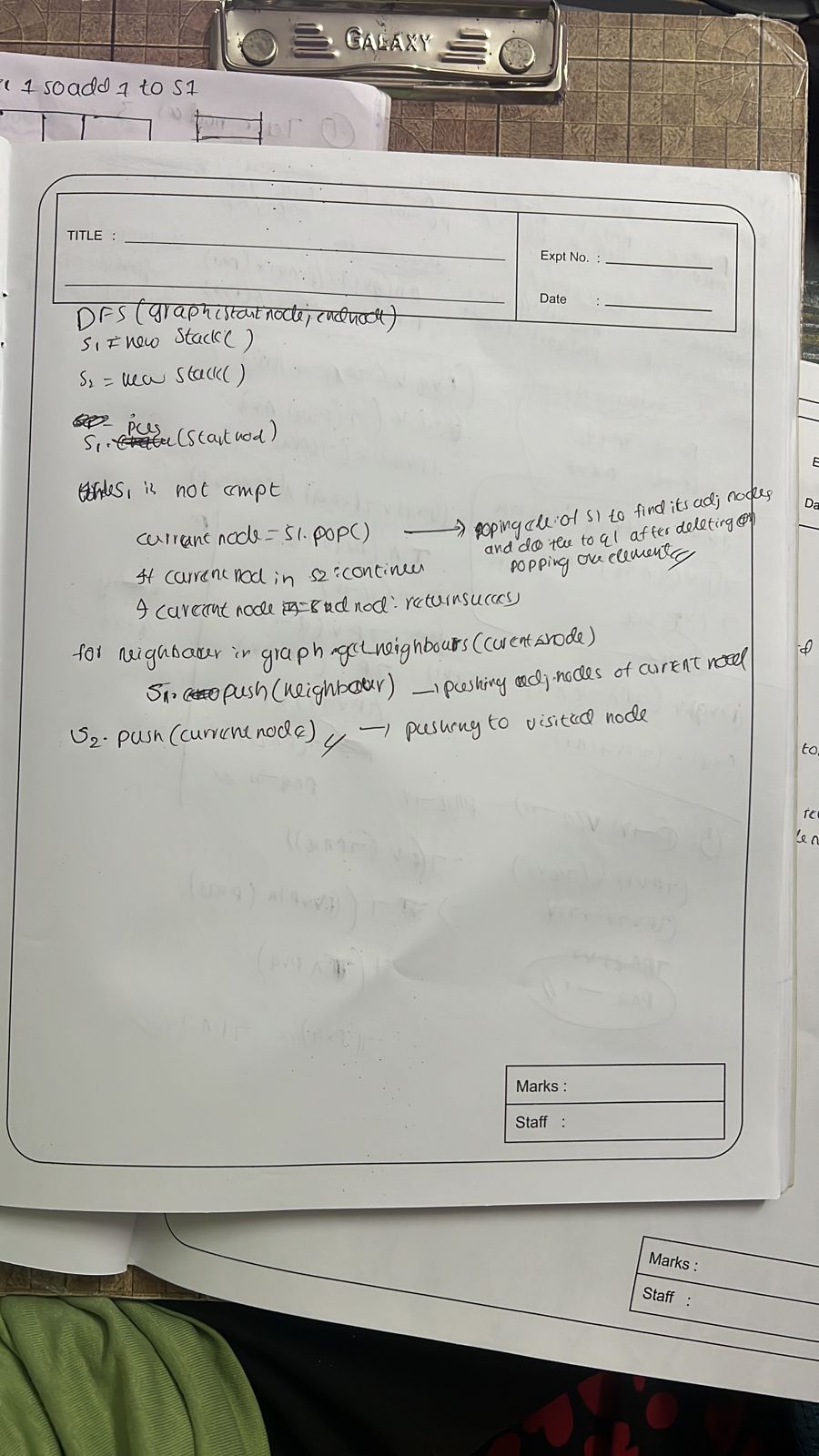
**Output −** Traverse all of the nodes, if the graph is connected. 

Time complexity will be O(V+E) for all the cases.

Q5. Construct the algorithm for Depth-First Search (DFS) traversal of a directed graph. Mention the best, worst, and average-case complexity of the DFS.8M

**Input −** The list of vertices, and the start vertex.

**Output −** Traverse all of the nodes, if the graph is connected



Time complexity will be O(V+E) for all the cases.

Q6. List Difference between BFS and DFS. 7M

|  |  |
| --- | --- |
| **BFS** | **DFS** |
| BFS uses Queue to find the shortest path. | DFS uses Stack to find the shortest path. |
| BFS is better when target is closer to Source. | DFS is better when target is far from source. |
| As BFS considers all neighbour so it is not suitable for decision tree used in puzzle games. | DFS is more suitable for decision tree. As with one decision, we need to traverse  further to augment the decision. If we reach the conclusion, we won. |
| BFS is slower than DFS. | DFS is faster than BFS. |
| Time Complexity of BFS = O(V+E) where V is vertices and E is edges. | Time Complexity of DFS is also O(V+E) where V is vertices and E is edges. |

Q7. Construct the binary search tree for the following data elements: 45, 15, 79, 90, 10, 55, 12, 20, 50

Trace the algorithm which you have constructed in 3a for every insertion and depict the final

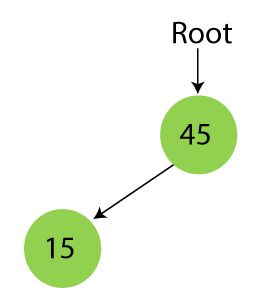
tree.7M

# Step 1 - Insert 45.



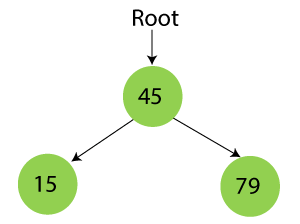
**Step 2 - Insert 15.**

As 15 is smaller than 45, so insert it as the root node of the left subtree.



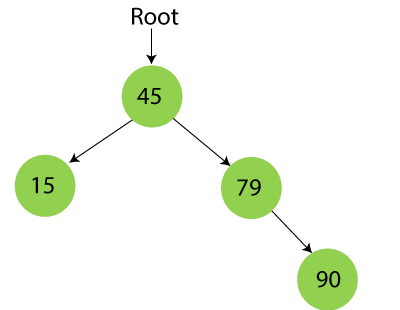
# Step 3 - Insert 79.

As 79 is greater than 45, so insert it as the root node of the right subtree.



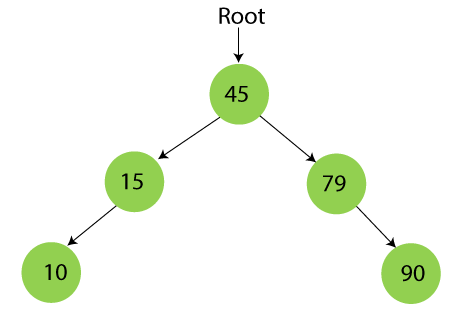
# Step 4 - Insert 90.

90 is greater than 45 and 79, so it will be inserted as the right subtree of 79.



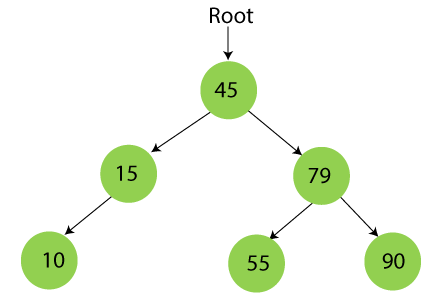
# Step 5 - Insert 10.

10 is smaller than 45 and 15, so it will be inserted as a left subtree of 15.



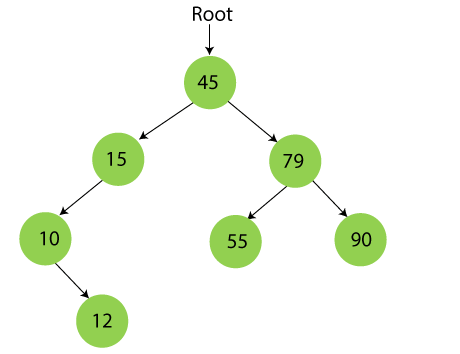
# Step 6 - Insert 55.

55 is larger than 45 and smaller than 79, so it will be inserted as the left subtree of 79.



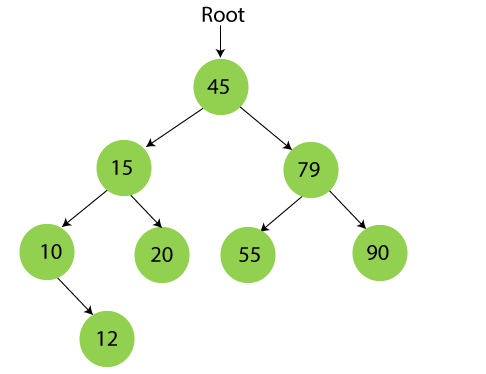
# Step 7 - Insert 12.

12 is smaller than 45 and 15 but greater than 10, so it will be inserted as the right subtree of 10.



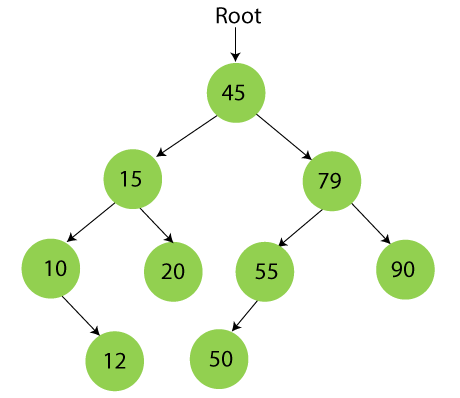
# Step 8 - Insert 20.

20 is smaller than 45 but greater than 15, so it will be inserted as the right subtree of 15.



# Step 9 - Insert 50.

50 is greater than 45 but smaller than 79 and 55. So, it will be inserted as a left subtree of 55.



Now, the creation of binary search tree is completed.

Q8. Design an algorithm to sort the n number of elements using the divide and conquer technique and provide the time complexities for best average, and worst case.

Design an algorithm to sort the n number of elements using the divide and conquer technique and provide the time complexities for best average, and worst case. 8M

Either we can use Merge sort or Quick sort Input: unsorted array of elements

Output: sorted array of elements

Time complexity of merge sort will be O(n log n) for all the cases.

Q6. Sort the following elements using the above-designed divide and conquer algorithm 100, 25, 98, 54, 79, 64, 84, 26, 48 and16. Trace the algorithm for each and every process.7M

ALGORITHM:

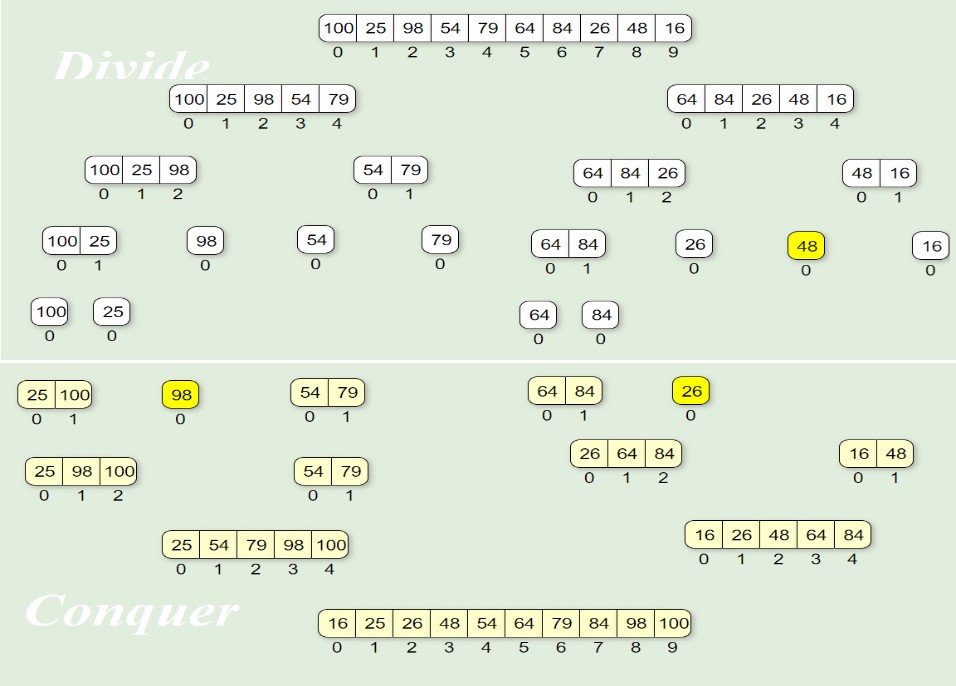
|  |
| --- |
| 1. **void** merge(**int** a[], **int** beg, **int** mid, **int** end) 2. { 3. **int** i, j, k; 4. **int** n1 = mid - beg + 1; 5. **int** n2 = end - mid; 7. /\* temporary Arrays \*/ 8. **int** LeftArray[] = **new** **int**[n1]; 9. **int** RightArray[] = **new** **int**[n2]; 11. /\* copy data to temp arrays \*/ 12. **for** (i = 0; i < n1; i++)    // i in for loop will be upto for loop itself if it is declared in for loop li(for int i=0) 13. LeftArray[i] = a[beg + i];   //but here it is not so I will of where it is declared is changed 14. **for** (j = 0; j < n2; j++) 15. RightArray[j] = a[mid + 1 + j]; 17. i = 0; /\* initial index of first sub-array \*/  //here we are making 0 bcz I value will be n1 bcz it will 18. j = 0; /\* initial index of second sub-array \*/   //be modified in for loop above 19. k = beg;  /\* initial index of merged sub-array \*/ 21. **while** (i < n1 && j < n2) 22. { 23. **if**(LeftArray[i] <= RightArray[j]) 24. { 25. a[k] = LeftArray[i]; 26. i++; 27. } 28. **else** 29. { 30. a[k] = RightArray[j]; 31. j++; 32. } 33. k++; 34. } 36. /\* Copy remaining elements of L[] if any \*/ 37. **while** (i<n1) 38. { 39. a[k] = LeftArray[i]; 40. i++;     //k , i value will be continuation of I in while loop 41. k++;     //they are declared outside of while loop.so if there are extra elements i 42. }     //in LA[] then as condition is (i < n1 && j < n2)    so both wont satisfy at a size 43. //greater than RA[] so the extra elements from the remaining index which are 44. **while** (j<n2)     //<n1 are copied and merged. 45. { 46. a[k] = RightArray[j]; 47. j++; 48. k++; 49. } 50. } 51. **void** mergeSort(**int** a[], **int** beg, **int** end) 52. { 53. **if** (beg < end)   //to check if more than 1 ele or there or not if yes to divide 54. { 55. **int** mid = (beg + end) / 2; 56. mergeSort(a, beg, mid); 57. mergeSort(a, mid + 1, end); 58. merge(a, beg, mid, end); 59. } 60. } |
|  |

|  |
| --- |
| 1. **class** Merge { 3. /\* Function to merge the subarrays of a[] \*/ 4. **void** merge(**int** a[], **int** beg, **int** mid, **int** end) 5. { 6. **int** i, j, k; 7. **int** n1 = mid - beg + 1; 8. **int** n2 = end - mid; 10. /\* temporary Arrays \*/ 11. **int** LeftArray[] = **new** **int**[n1]; 12. **int** RightArray[] = **new** **int**[n2]; 14. /\* copy data to temp arrays \*/ 15. **for** (i = 0; i < n1; i++) 16. LeftArray[i] = a[beg + i]; 17. **for** (j = 0; j < n2; j++) 18. RightArray[j] = a[mid + 1 + j]; 20. i = 0; /\* initial index of first sub-array \*/ 21. j = 0; /\* initial index of second sub-array \*/ 22. k = beg;  /\* initial index of merged sub-array \*/ 24. **while** (i < n1 && j < n2) 25. { 26. **if**(LeftArray[i] <= RightArray[j]) 27. { 28. a[k] = LeftArray[i]; 29. i++; 30. } 31. **else** 32. { 33. a[k] = RightArray[j]; 34. j++; 35. } 36. k++; 37. } 38. /\* Copy remaining elements of L[] if any \*/ 39. **while** (i<n1) 40. { 41. a[k] = LeftArray[i]; 42. i++;     //k,i value will be continuation of I in while loop 43. k++;     //they are declared outside of while loop.so if there are extra elements i 44. }    //in LA[] then as condition is (i < n1 && j < n2)    so both wont satisfy at a size 45. //greater than RA[] so the extra elements from the remaining index which are 46. **while** (j<n2)     //<n1 are copied and merged. 47. { 48. a[k] = RightArray[j]; 49. j++; 50. k++; 51. } 52. } 54. **void** mergeSort(**int** a[], **int** beg, **int** end) 55. { 56. **if** (beg < end) 57. { 58. **int** mid = (beg + end) / 2; 59. mergeSort(a, beg, mid); 60. mergeSort(a, mid + 1, end); 61. merge(a, beg, mid, end); 62. } 63. } 65. /\* Function to print the array \*/ 66. **void** printArray(**int** a[], **int** n) 67. { 68. **int** i; 69. **for** (i = 0; i < n; i++) 70. System.out.print(a[i] + " "); 71. } 73. **public** **static** **void** main(String args[]) 74. { 75. **int** a[] = { 11, 30, 24, 7, 31, 16, 39, 41 }; 76. **int** n = a.length; 77. Merge m1 = **new** Merge(); 78. System.out.println("\nBefore sorting array elements are - "); 79. m1.printArray(a, n); 80. m1.mergeSort(a, 0, n - 1); 81. System.out.println("\nAfter sorting array elements are - "); 82. m1.printArray(a, n); 83. System.out.println(""); 84. } 86. } |

### **1. Time Complexity**

|  |  |
| --- | --- |
| **Case** | **Time Complexity** |
| **Best Case** | O(n\*logn) |
| **Average Case** | O(n\*logn) |
| **Worst Case** | O(n\*logn) |

* **Best Case Complexity -** It occurs when there is no sorting required, i.e. the array is already sorted. The best-case time complexity of merge sort is **O(n\*logn)**.
* **Average Case Complexity -** It occurs when the array elements are in jumbled order that is not properly ascending and not properly descending. The average case time complexity of merge sort is **O(n\*logn)**.
* **Worst Case Complexity -** It occurs when the array elements are required to be sorted in reverse order. That means suppose you have to sort the array elements in ascending order, but its elements are in descending order. The worst-case time complexity of merge sort is **O(n\*logn)**.



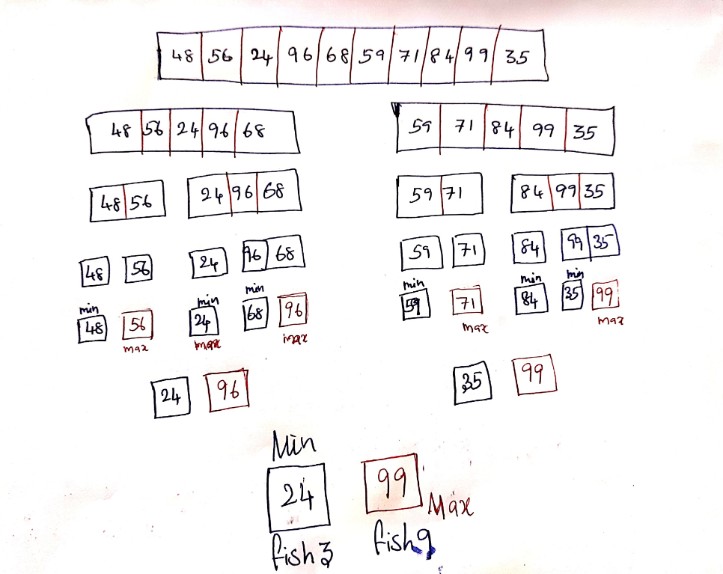
Q9. A man has 10 varieties of fish:

fish1, fish2 …… fish10 namely, in his aquarium. The number of fishes from fish1, fish2…. fish10 are as follows:

48, 56, 24, 96, 68, 59, 71, 84, 99, and 35.

Apply the divide and conquer methodology to find the name of the fish which is maximum and minimum in the aquarium.8M

We can apply the Maxmin algorithm to find the minimum and the maximum number of fish in the aquarium in minimum time complexity using the divide and conquer technique.



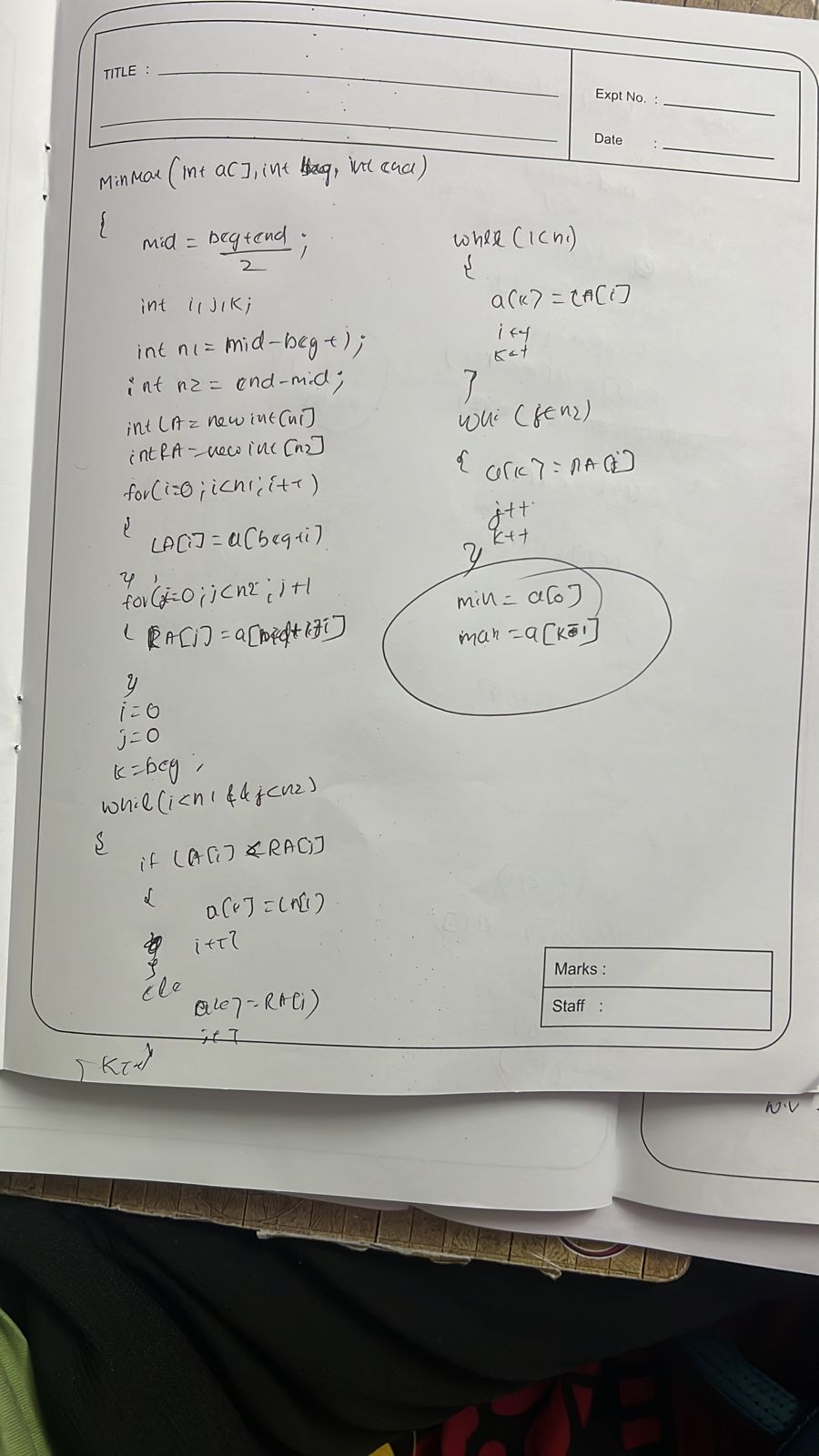
Q10. Suggest and design the suitable algorithm for the above problem and also provide the time complexity for that algorithm. 7M

Algorithm MAXMIN (A, low, high)

// Description : Find minimum and maximum element from array using divide and conquer approach

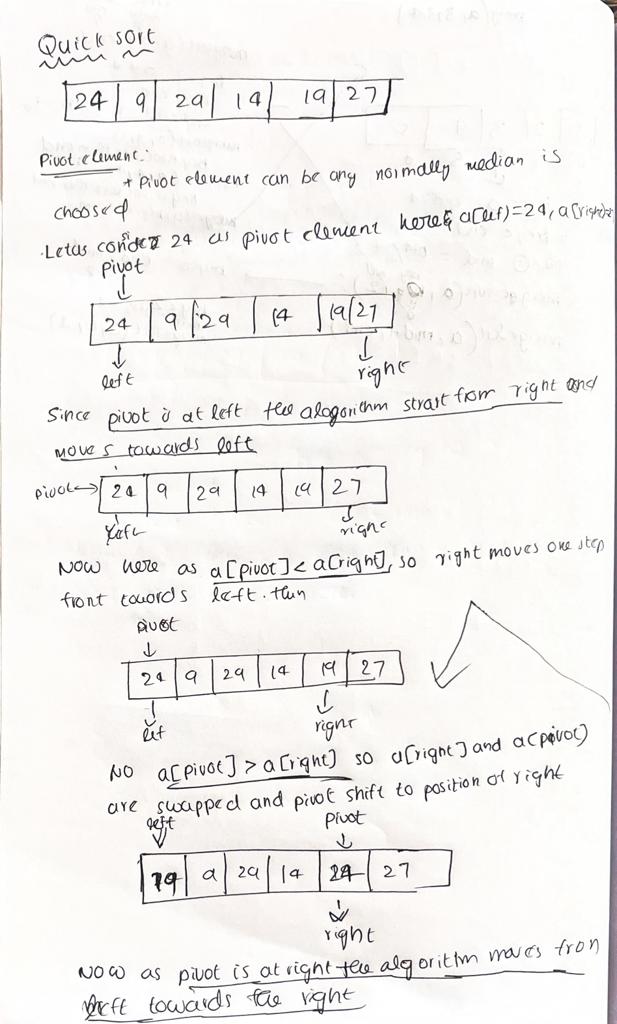
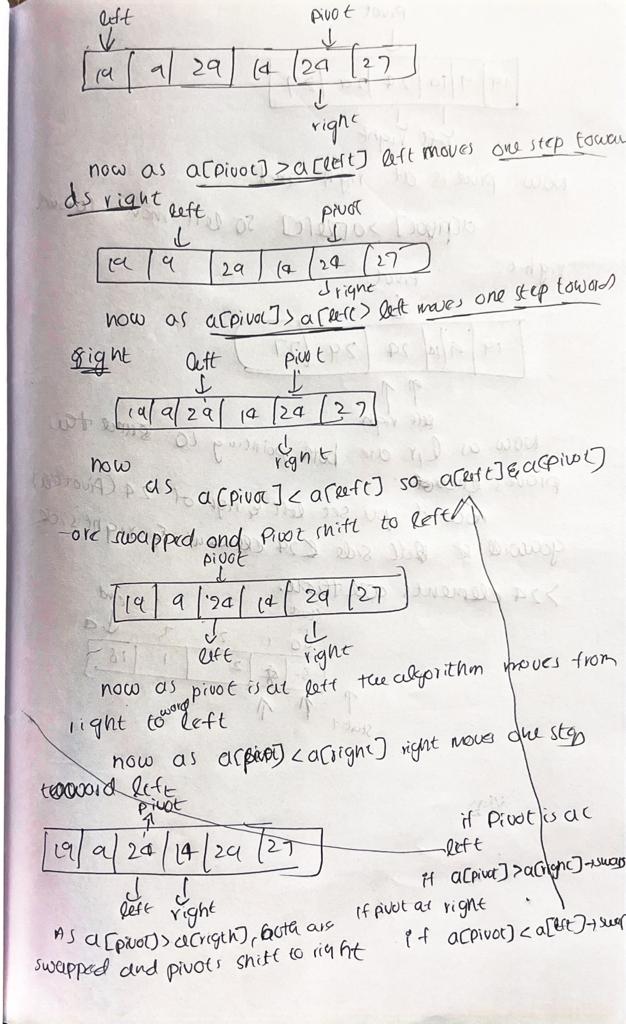
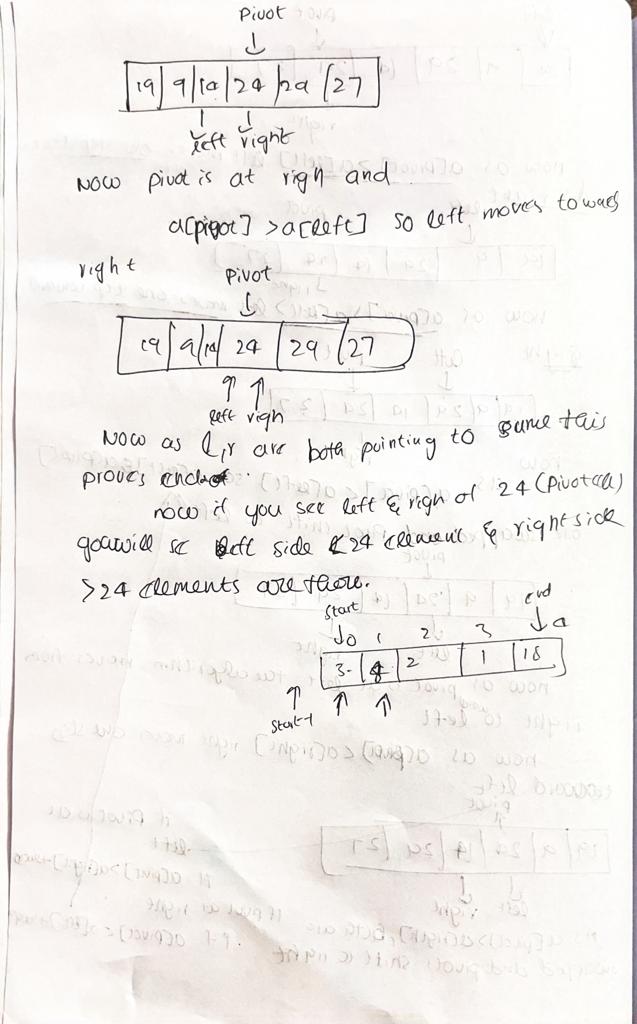
// Input : Array A of length n, and indices low = 0 and high = n - 1

// Output : (min, max) variables holding minimum and maximum element of array



Q11. Discuss Quick Sort Algorithm and Explain it with example. Derive Worst case and Average Case Complexity.

[limk](https://www.javatpoint.com/quick-sort)

|  |
| --- |
| 2. /\* function to implement quick sort \*/ 3. **void** quick(**int** a[], **int** start, **int** end) /\* a[] = array to be sorted, start = Starting index, end = Ending index \*/ 4. { 5. **if** (start < end) 6. { 7. **int** p = partition(a, start, end);  //p is partitioning index 8. quick(a, start, p - 1); 9. quick(a, p + 1, end); 10. } 11. } 12. /\* function that consider last element as pivot, 13. place the pivot at its exact position, and place 14. smaller elements to left of pivot and greater 15. elements to right of pivot.  \*/ 16. **int** partition (**int** a[], **int** start, **int** end) 17. { 18. **int** pivot = a[end]; // pivot element 19. **int** i = (start - 1); 21. **for** (**int** j = start; j <= end - 1; j++) 22. { 23. // If current element is greater than the pivot 24. **if** (a[j] < pivot) 25. { 26. i++; // increment index of smaller element 27. **int** t = a[i]; 28. a[i] = a[j]; 29. a[j] = t; 30. } 31. } 32. **int** t = a[i+1]; 33. a[i+1] = a[end]; 34. a[end] = t; 35. **return** (i + 1); 36. } |

# Q12. Explain Strassen’s matrix multiplication. Evaluate its efficiency.

1. Let A and B be two nn matrices, that is, each having n rows and n columns. If C=AB, then the product matricx C will also have n rows and n columns.
2. An element C[i, j] can now be found using the formula

C[i,j]=∑k=0n−1A[i,k]∗B[k,j]C[i,j]=∑k=0n−1A[i,k]∗B[k,j]

1. The method of multiplying two matrices is known as the classic matrix multiplication method. Using the above formula, we may write the following statements:

for( i=0; i < n; i++) for( j=0; j<n; j++)

{

C[i] [j] =0;

for( k=0;k<n;k++) C[i][j]=c[i][j]+a[i][k]\*b[k][j];

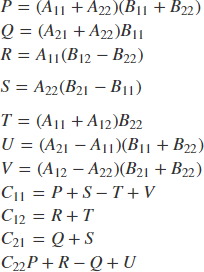
}

It can be easily verified that the complexity of the method is O(n3).

Strassen’s matrix multiplication can be used only when n is a power of 2i.e. n=2k. If n is not a power of 2 then enough rows and columns of zeros can be added to both A and B so that the resulting dimension are a power of 2.

In this method matrix A and B are splitted into 4 squares sub-matrices where each sub- matrices has dimension of n/2. Now the product is found as shown:





It is observed that all c [i, j] can be calculated using a total of 7 multiplications and 18 additions or subtractions.

It is not necessary that n should be always 2. If n>2 then the same formula can be used but now it will be done recursively. The same formula will be continuously applied on smaller sized matrices till n gets reduced to 2. When n=2, this will be terminating condition of recursion and now multiplication is done directly.

# Analysis:

Recurrence relation for Strassen’s Algorithm is:

T(n) = 7T(n/2) + Θ(n2)

Applying the Master Theorem to T(n) = a T(n/b) + f(n) with a=7, b=2, and f(n)=Θ(n2).

Since f(n) = O(nlogb(a)-ε) = O(nlog2(7)-ε), case a) applies and we get T(n)= Θ(nlogb(a)) = Θ(nlog2(7)) = O(n2.81).

Example:

A= [1,3] B=[6,8]

[7,5] [4,2]

// Step 1: Split A and B into half-sized matrices of size 1x1 (scalars).

a11=1, a12=3, a21=7, a22=5, b11=6, b12=8, b21=4, b22=2

// Define the "S" matrix. s1=b12-b22 // 6

s2=a11+a12 // 4 s3= a21+a22 // 12 s4= b21-b11 // -2 s5=a11+a22 // 6 s6=b11+b22 //8 s7=a12-a22 //-2 s8=b21+b22 //6 s9=a11-a21 // -6 s10=b11 + b12 // 14

// Define the "P" matrix.

p1=a11\*s1 // 6 p2=s2\*b22 //8 p3=s3\*b11 //72 p4=a22\*s4//-10 p5=s5\*s6 // 48 p6=s7\*s8 //-12 p7=s9\*s10 // -84

// Fill in the resultant "C" matrix.

c11 = p5 + p4 - p2 + p6 // 18 c12 = p1 + p2 // 14

c21 = p3 + p4 // 62

c22 = p5 + p1 - p3 - p7 // 66

C=[18,14]