**Lab Report**

Analysis and Design of Algorithms



Submitted by: to:

Mukul Bhardwaj Dr. Mukesh Kumar  
UE233066 UIET, Panjab University  
BE(CSE)-4th semester

**Index**

|  |  |  |
| --- | --- | --- |
| Lab no. | Title | Date |
|  | 1. 1D peak 2. 2D peak 3. Magic square | 23 January 2025 |
|  | 1. Bubble sort 2. Heap sort 3. Insertion sort 4. Merge sort 5. Quick sort 6. Selection sort | 30 January 2025 |
|  | Knapsack problem by sorting on basis of:   1. Profit 2. Weight 3. Profit/Weight | 6 February 2025 |
|  | 1. Check whether a point is inside a triangle or not 2. Convex hull naive approach 3. find max min from array by Div and Conq 4. Generate all possible triangles out of n elements 5. iterative quick sort 6. matrix multiplication (conventional by D and C) | 13 February |
|  | 1. Activity selection problem 2. Djkstra’s algorithm 3. Strassen’s multiplication | 27 February |
|  | 1. Kruskal’s algorithm | 20 March |

Lab 1

Problem 1: Draw a 1D peak

Solution:

1. Start
2. Read n from user
3. Insert an array int Mountains[n]
4. Make another array peaks[n]:

Int \*peaks = new int[n](); //initialized with 0

1. for i -> 1 to n-1:

if Mountains[i]>Mountains[i+1] && Mountains[i]>Mountains[i-1]:

peaks[i]=1;

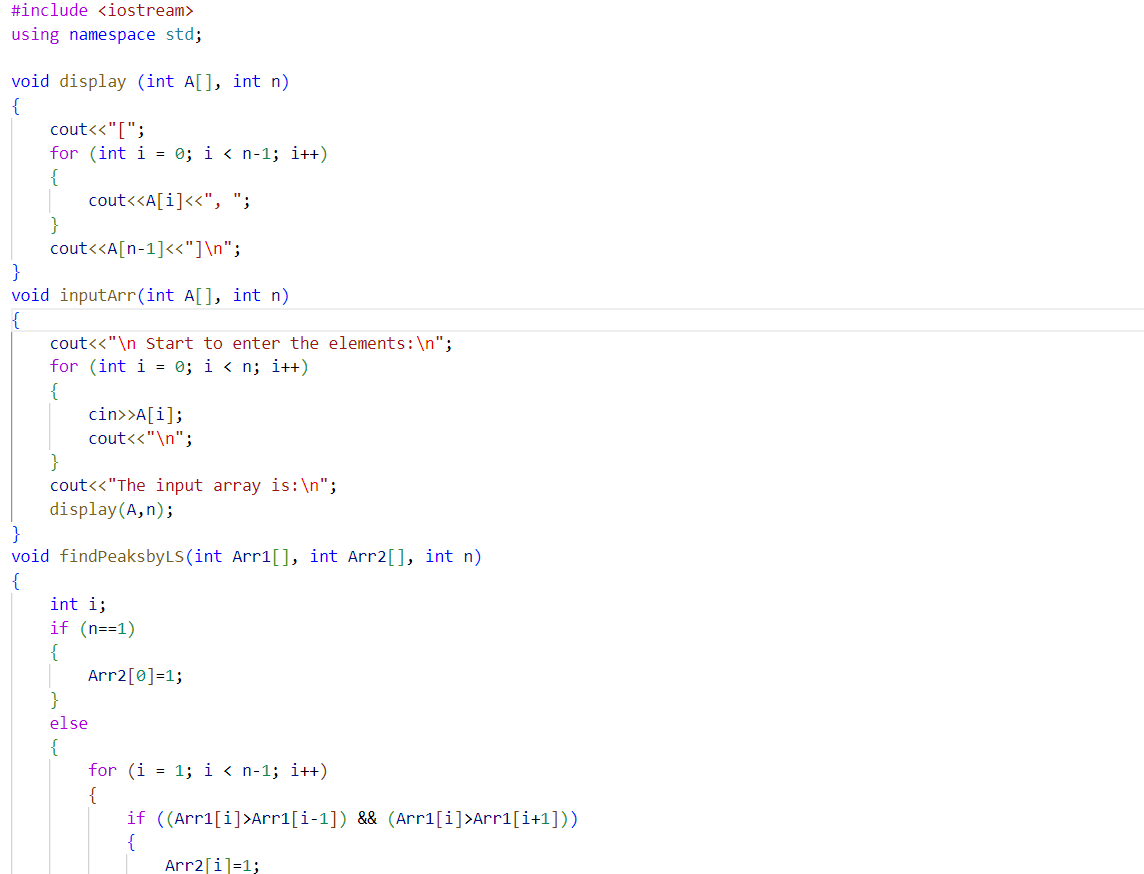
1. if Mountains[0]>Mountains[1]:

peaks[0]=1;

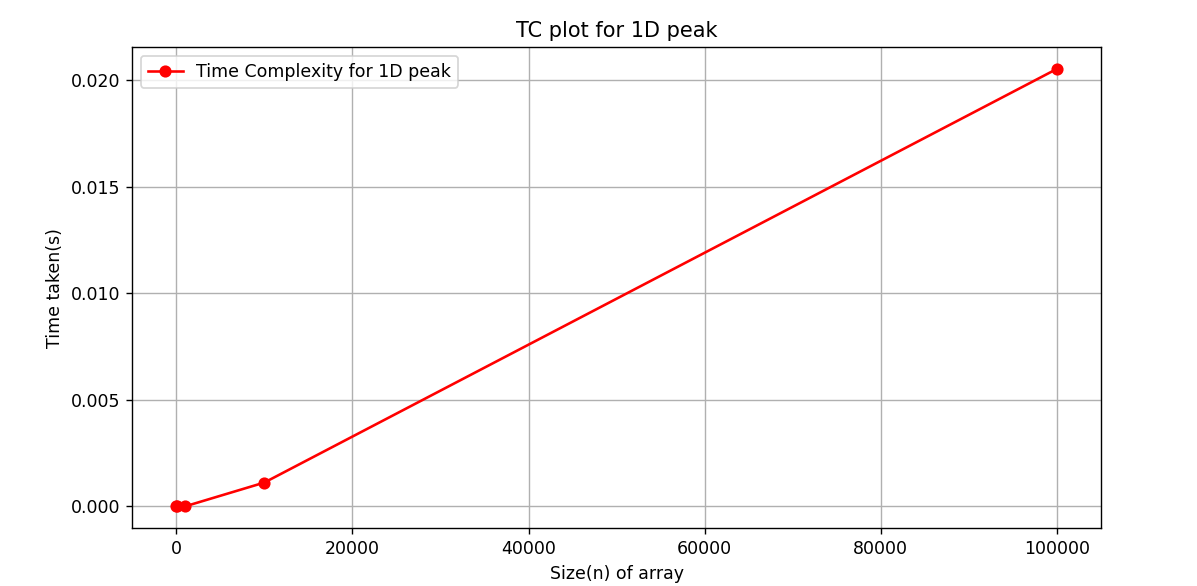
1. if Mountains[n-1]>Mountains[n-2]:

peaks[n-1]=1;

1. Print array peaks;
2. End







Problem 2: Draw a 2D peak

Solution:

1. Start
2. Read int m,n;
3. Insert values into a 2D array created by:

int \*\*Mountains = new int \*[m];

for int i ->0 to n-1:

Mountains[m]=new int[n];

1. Create another 2D array int peaks[m][n] with all entries 0s
2. Check whether an element in Mountains is such that it is greater than all its neighbours (left, right, up and down):

If yes: peaks[i][j]=1;

1. Print the 2D array peaks
2. Free allocated memory by:

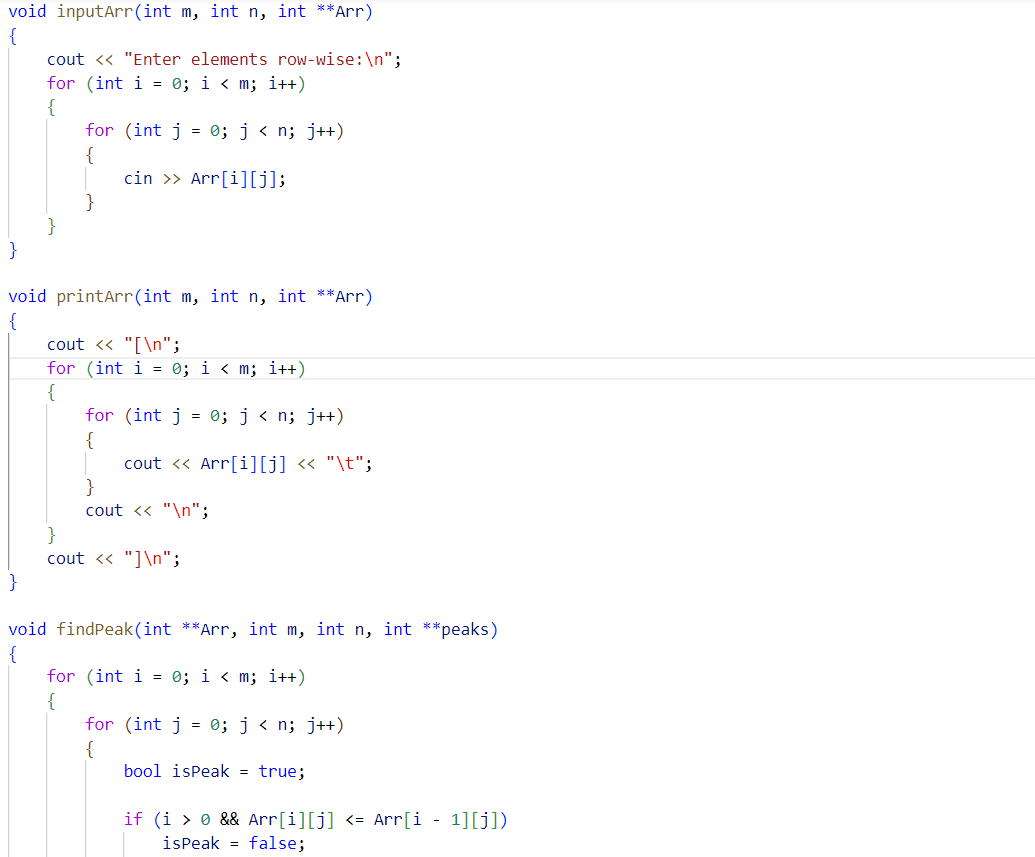
for int i=0 to m-1:

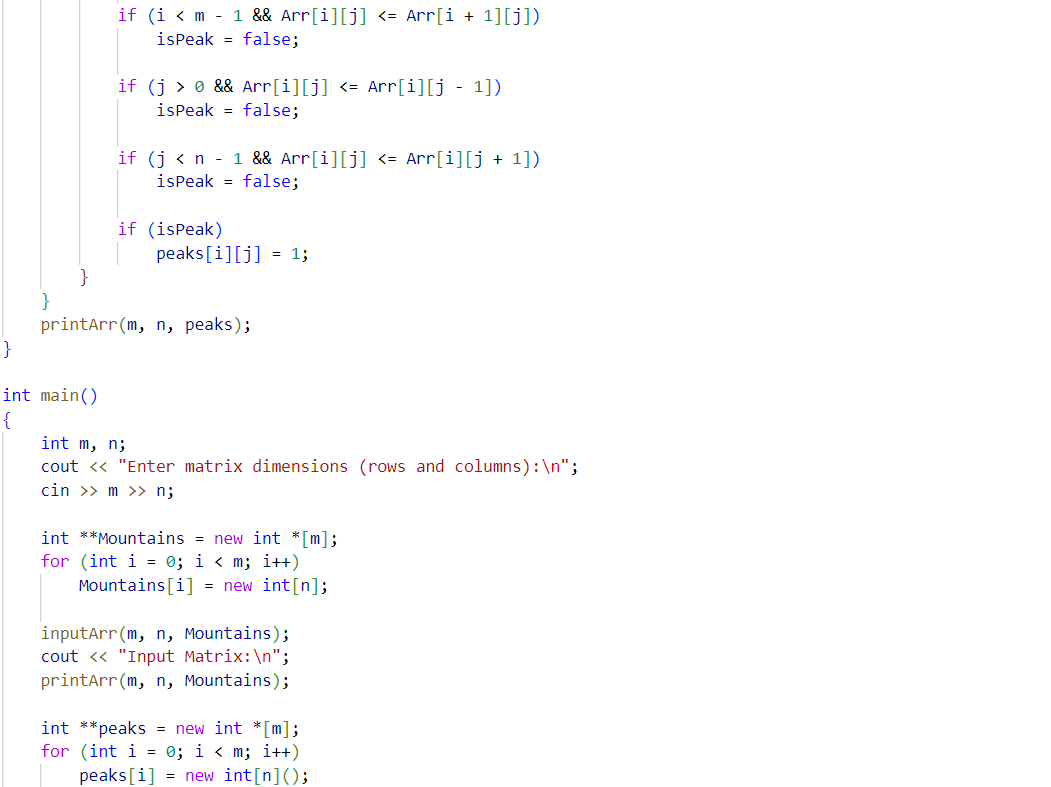
delete[] Mountains[i];

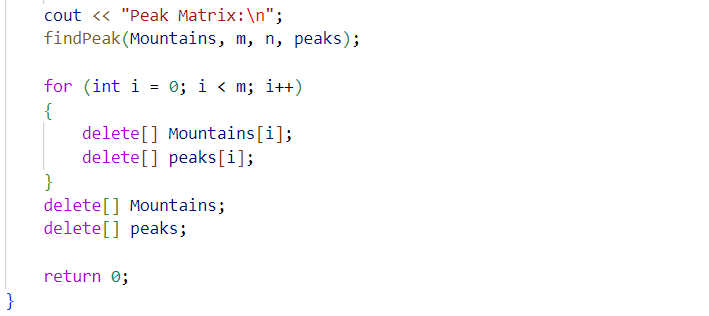
delete[] Mountains;

1. End









Problem 3: Draw a magic square

Solution:

1. Start
2. Read an odd int n;
3. Create an array int msquare[n][n] with all entries 0s
4. num=1;
5. Start with i=0 and j=n/2:

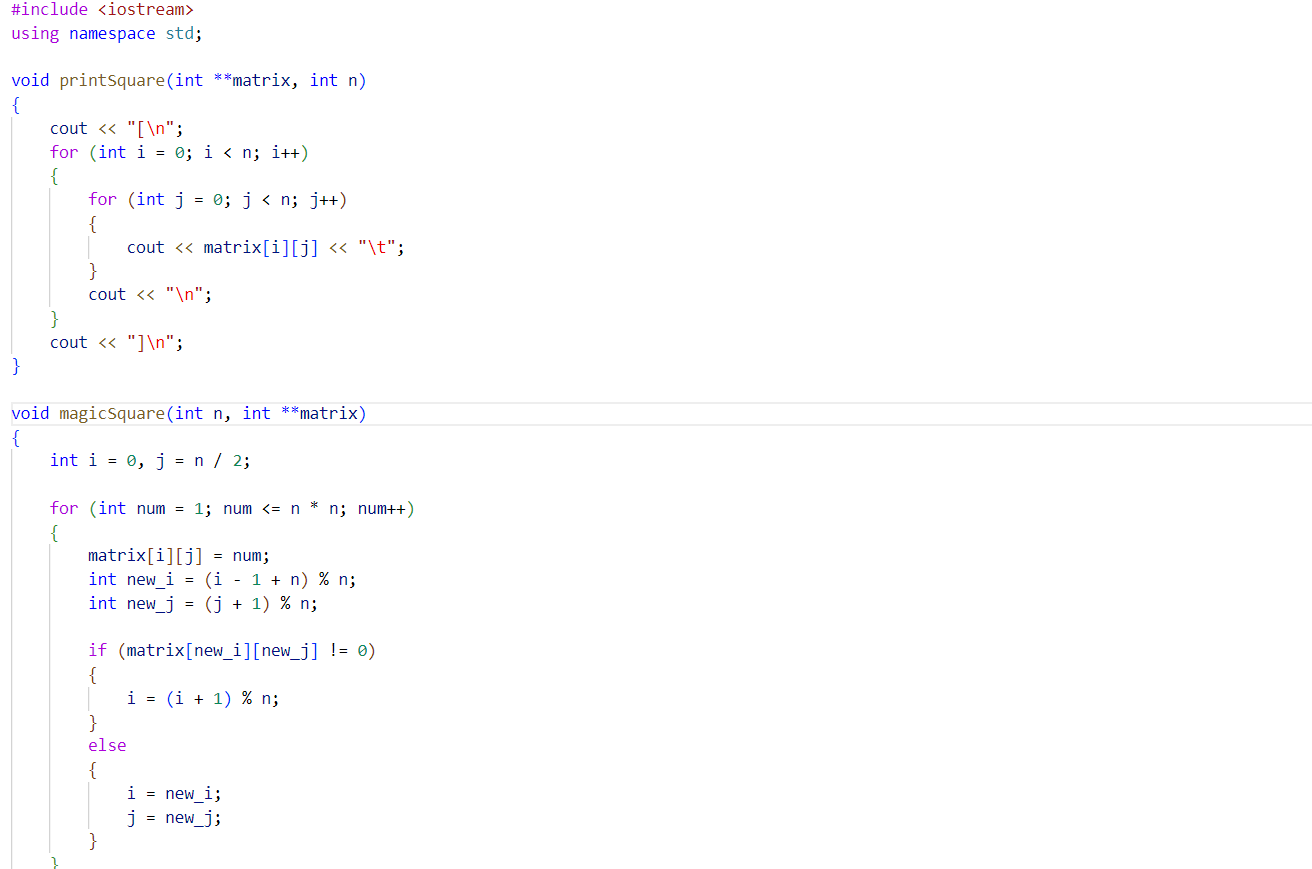
msquare[i][j]= num;

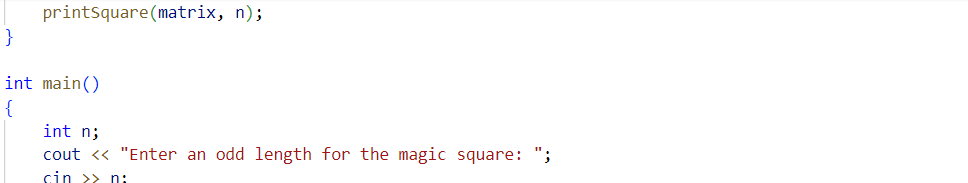
1. while num<=n\*n:

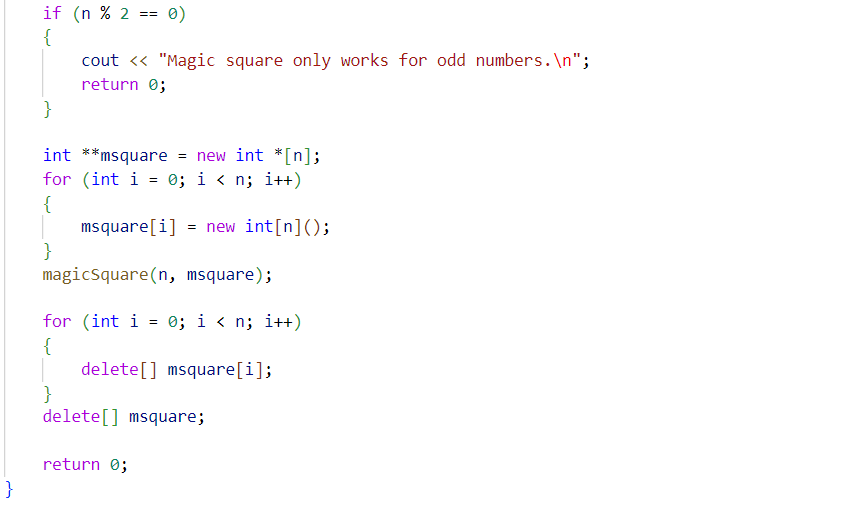
check for [i-1][j-1]th blank space for num and fill it

num=num+1;

1. print the matrix msquare
2. delete allocated memory
3. end







Lab 2

Problem 1: Bubble sort

Solution:

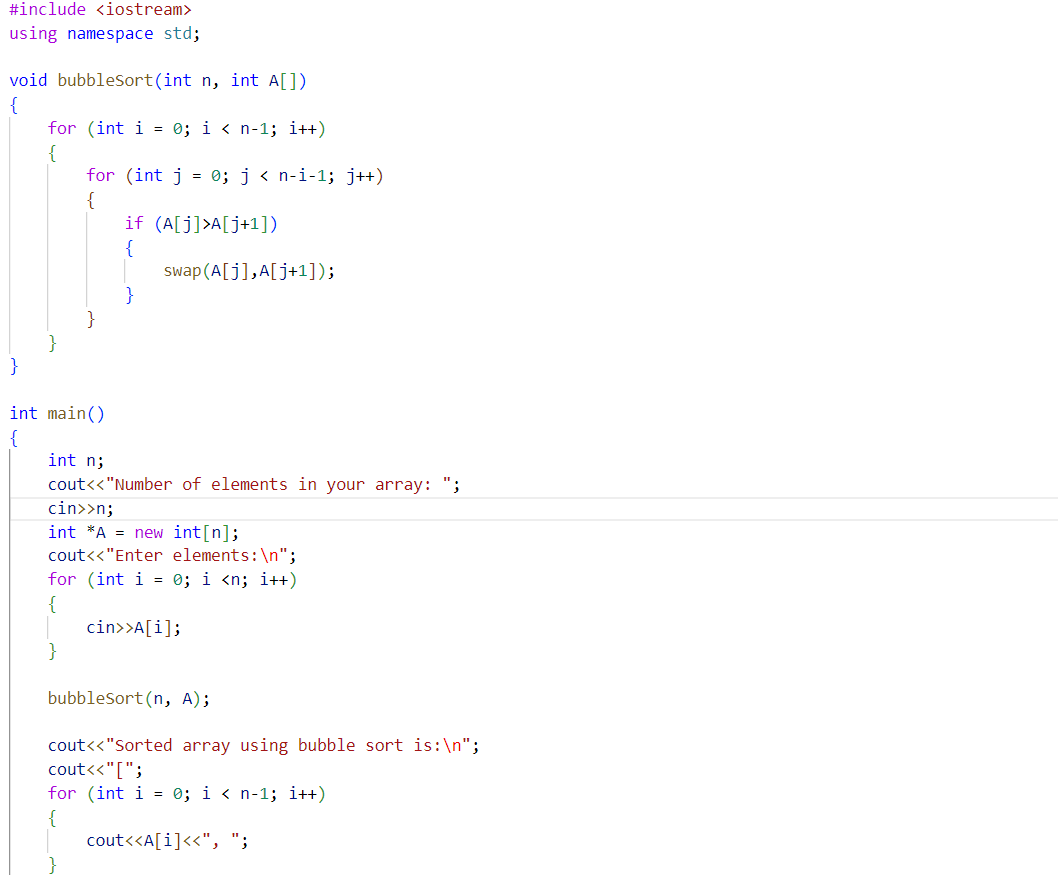
1. Start
2. Read n from user
3. Make an array int arr[n]
4. Insert values into this array
5. Now, for int i->0 to n-2:

for int j->0 to n-i-2:

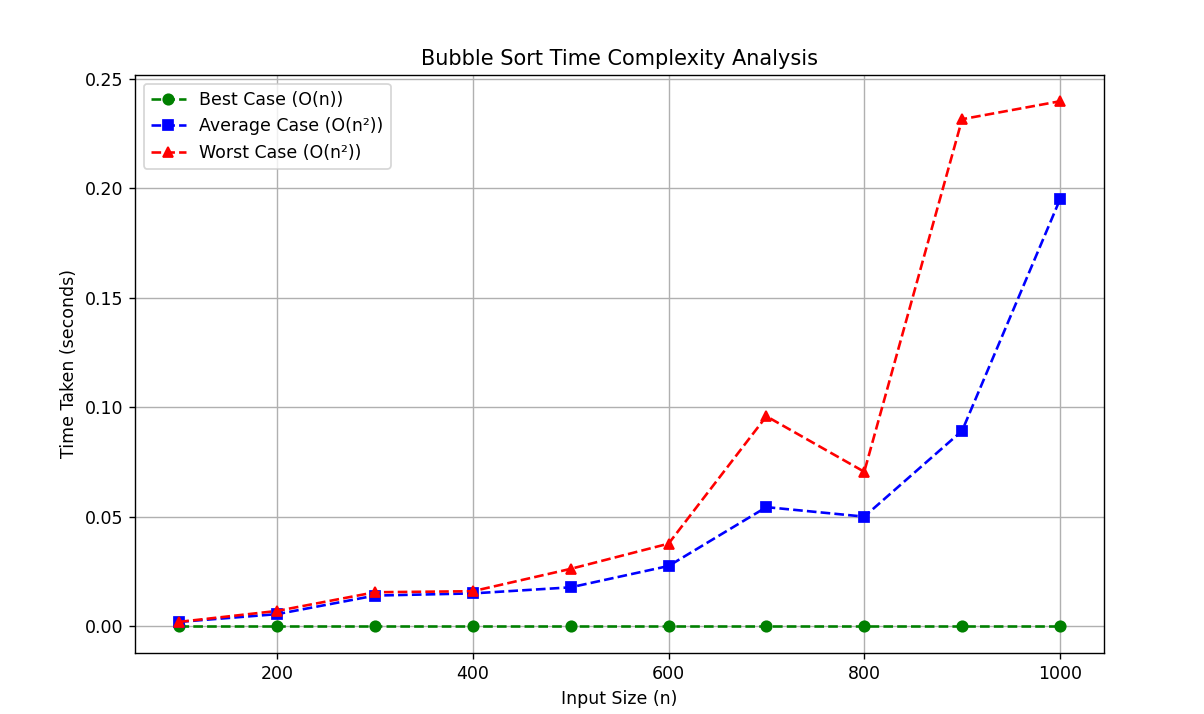
if arr[j]>arr[j+1]:

swap(arr[j],arr[j+1]);

1. Print sorted array arr[n]
2. End







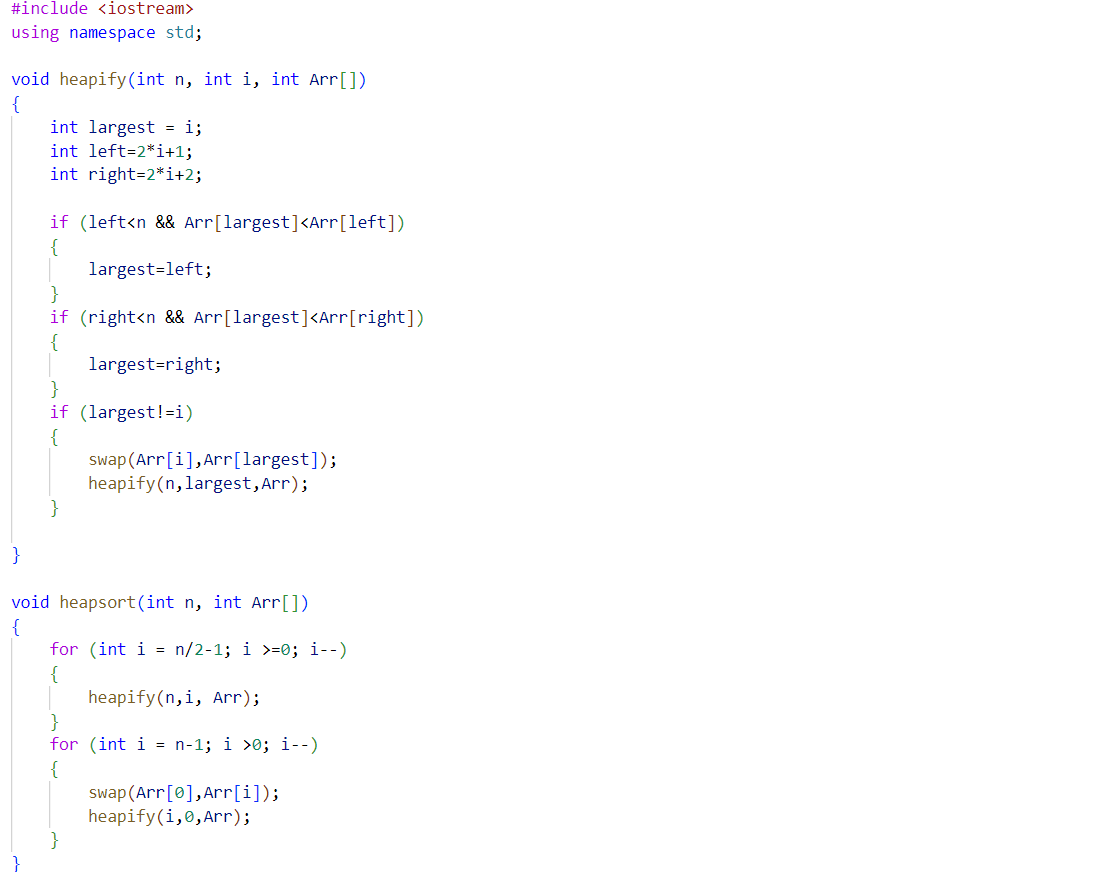
Problem 2: Heap sort

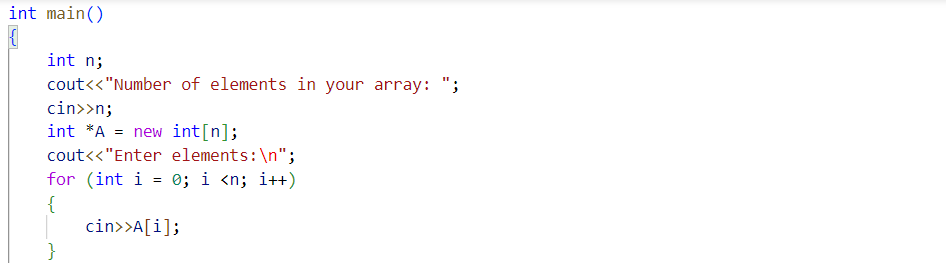
Solution:

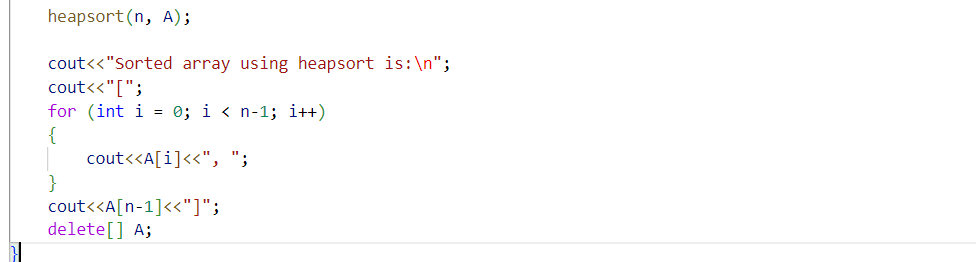
1. Start
2. Create function: void heapify(int Arr[], int i, int n)
3. Read n
4. Insert elements into an array arr[n] from the user
5. Call function: void heapsort(int Arr[], int n)

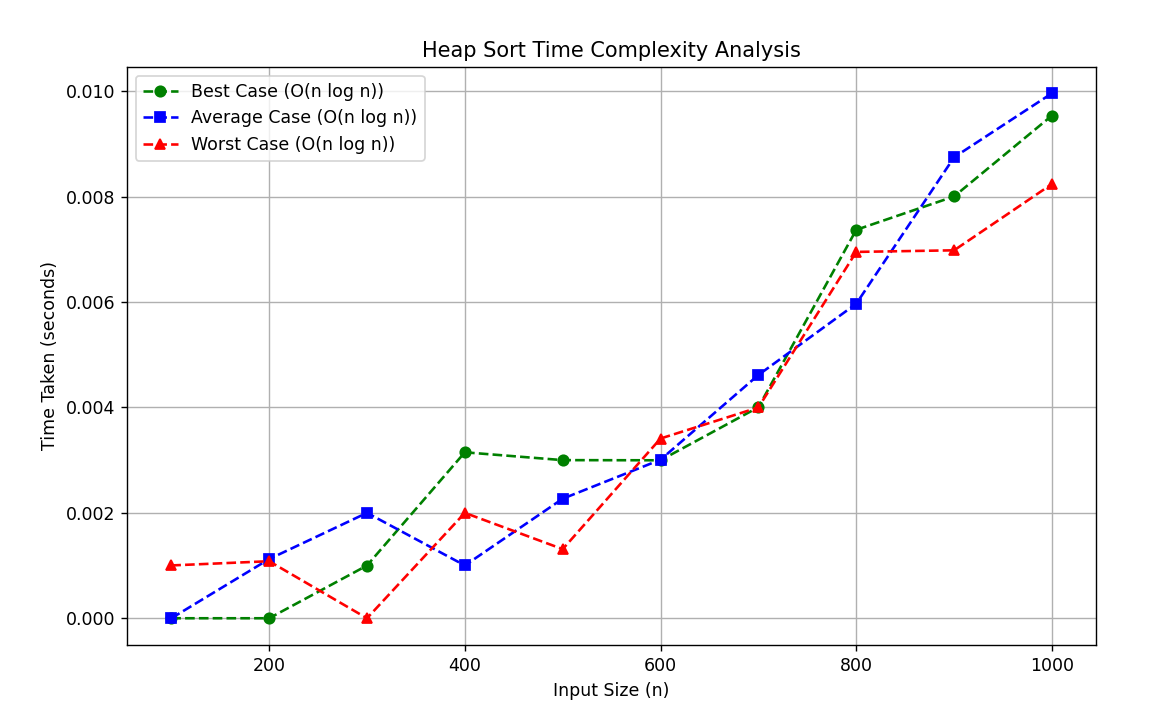
Now re-heapify the array within heapsort function after once converting it into a max heap and then swapping first and last elements until it is sorted

1. Print sorted array
2. End









Problem 3: Insertion sort

Solution:

1. Start
2. Read n from user
3. Make an array int arr[n]
4. Insert values into this array
5. for i->1 to n-1:

set key = arr[i];

set j = i – 1

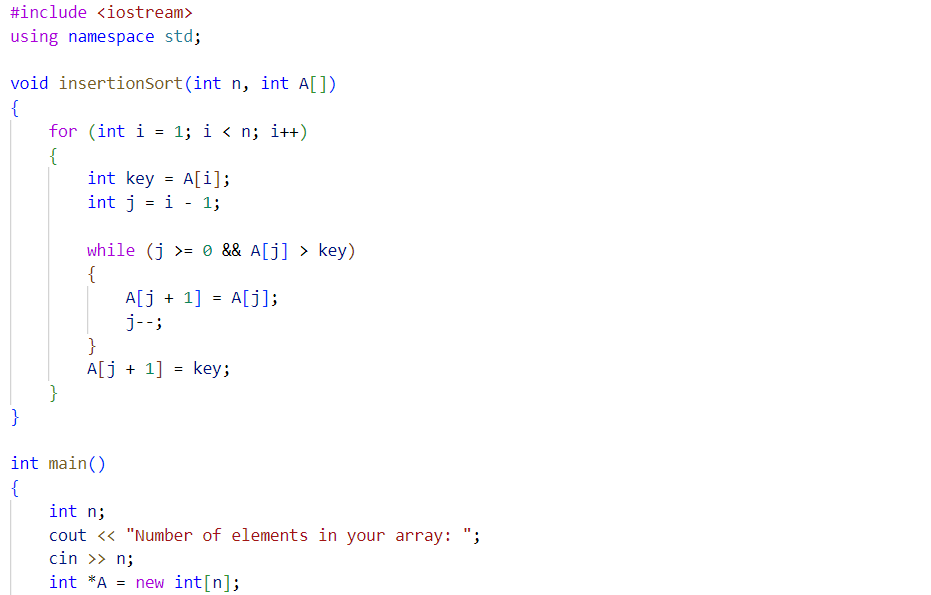
while j>=0 and arr[j] > key:

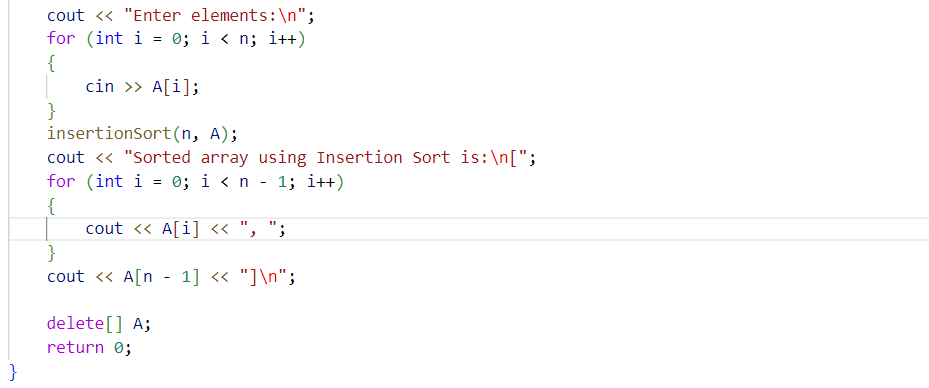
arr[j+1] = arr[j]

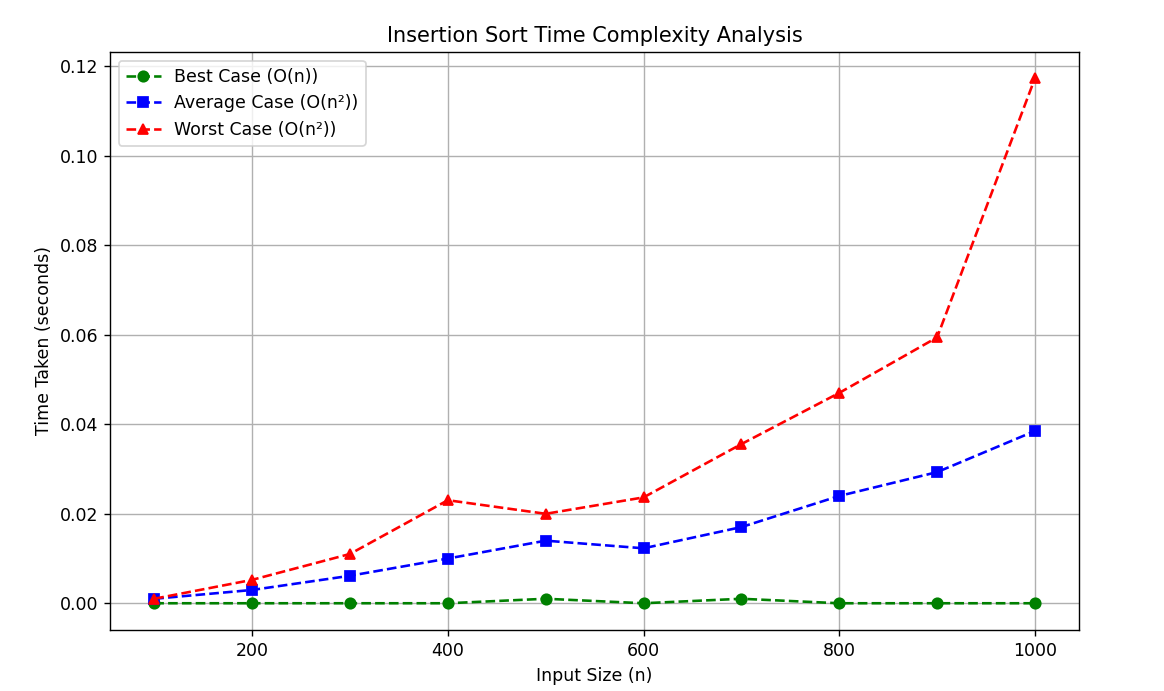
j = j – 1

insert key at arr[j+1]

1. print sorted array arr[n]
2. End







Problem 4: Merge sort

Solution:

1. Start
2. Read n from user
3. Make an array int arr[n]
4. Insert values into this array
5. Define function mergesort(arr, left, right):

If left < right:

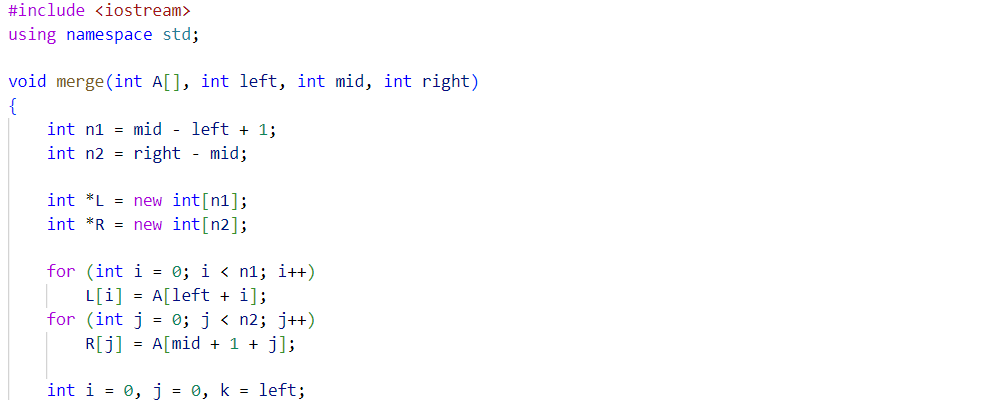
Find mid = (left + right)/2

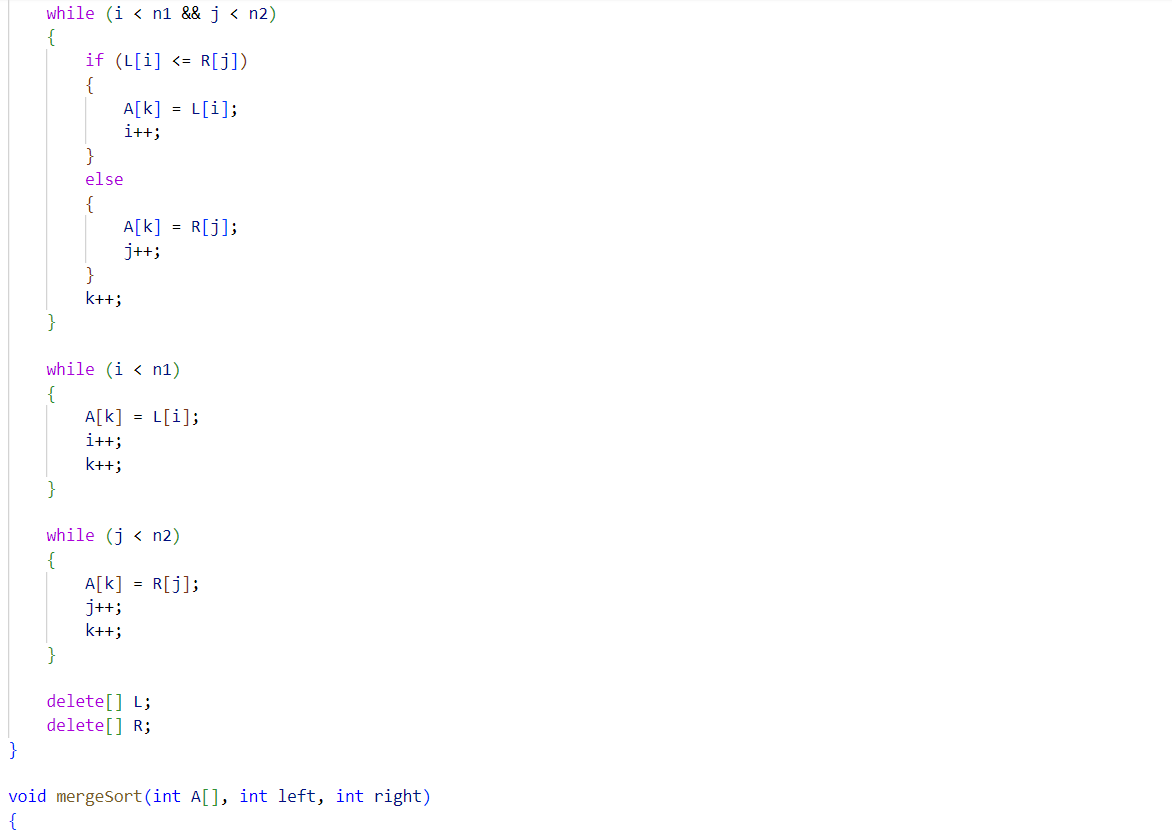
Call mergesort(arr, left, mid)

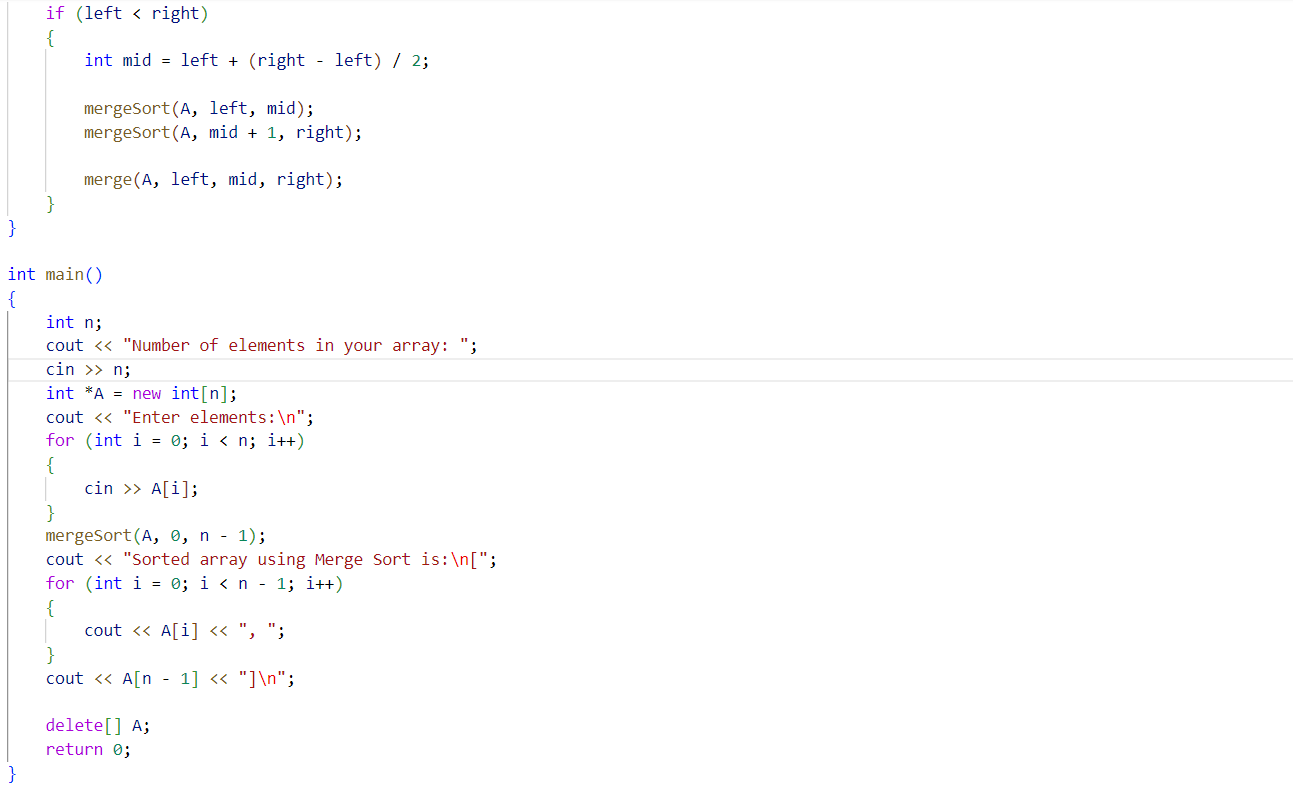
Call mergesort(arr, mid + 1, right)

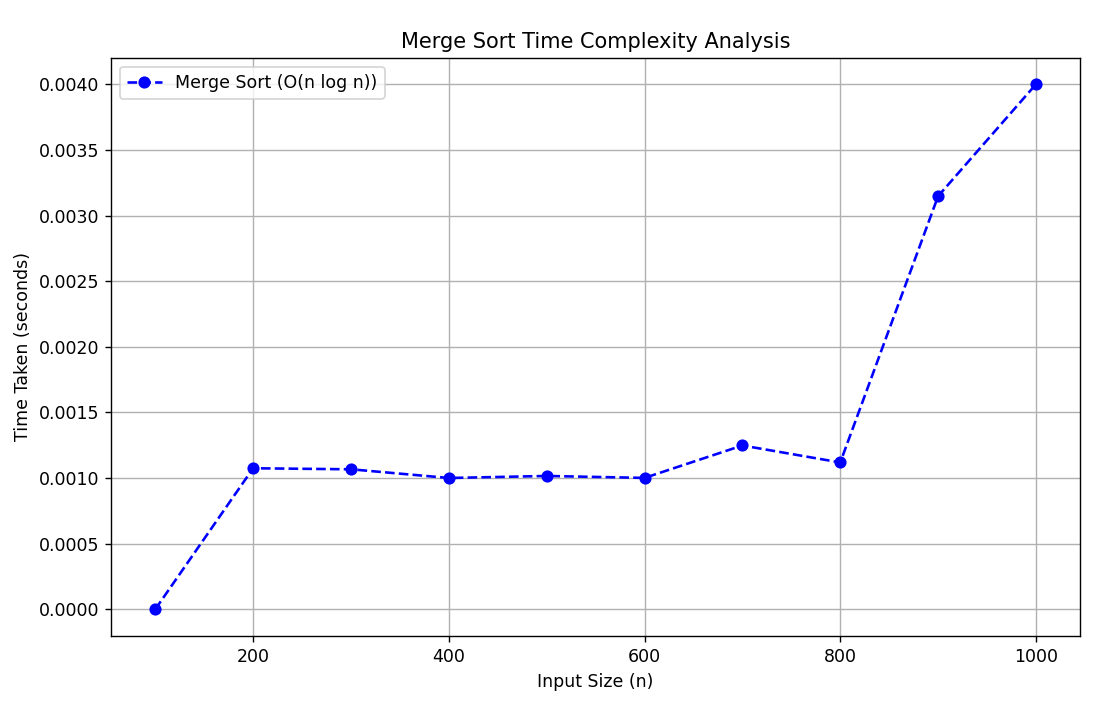
Merge both halves

1. Call mergesort(arr, 0, n-1)
2. Print sorted array arr[n]
3. End









Problem 5: Quick sort

Solution:

1. Start
2. Read n from user
3. Make an array int arr[n]
4. Insert values into this array
5. Define function quicksort(arr, low, high):

If low < high:

Set pivot = arr[high], i = low – 1

for j = low to high-1:

if arr[j] < pivot:

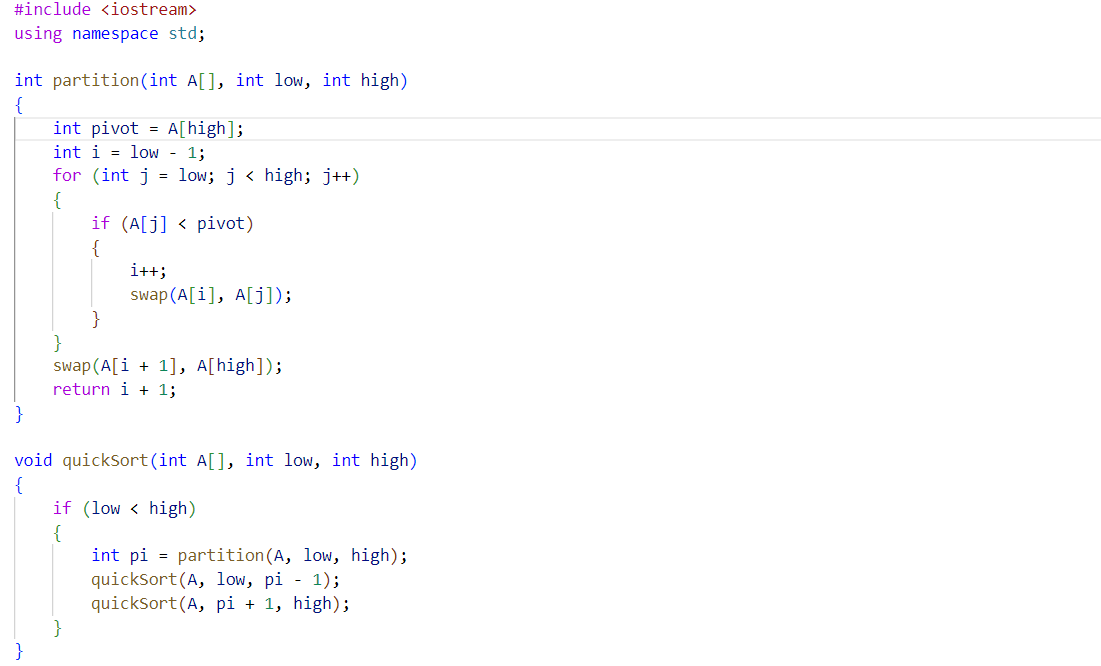
swap(arr[++i], arr[j])

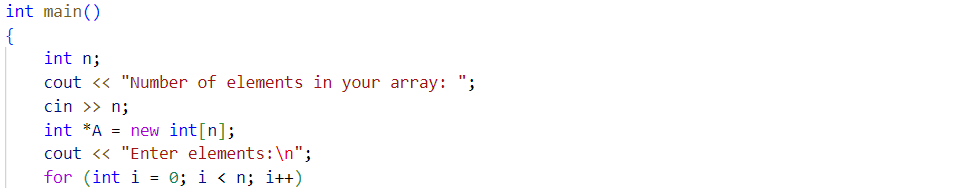
swap(arr[i+1], arr[high])

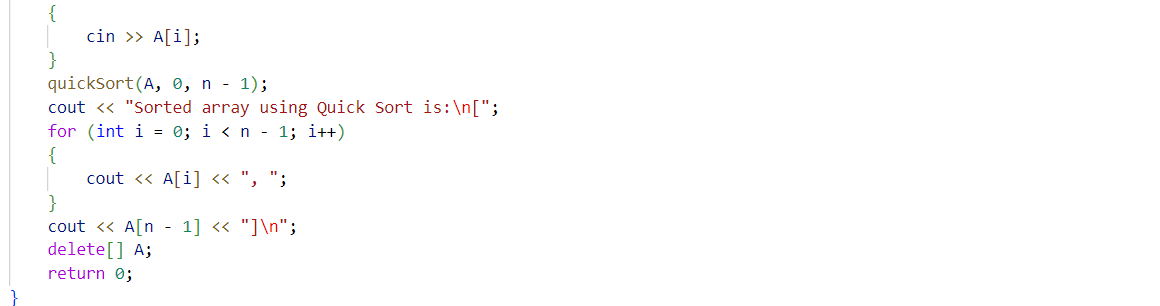
call quicksort(arr, low, partitionIndex – 1)

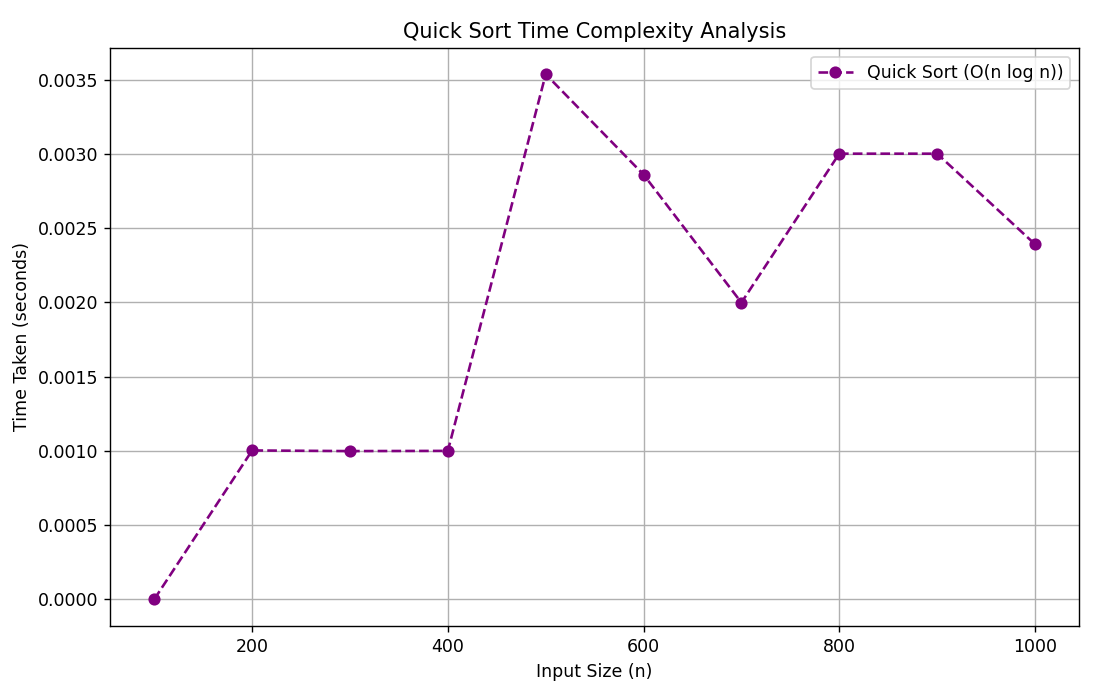
call quicksort(arr, partitionIndex + 1, high)

1. Call quicksort(arr, 0, n-1)
2. Print sorted array arr[n]
3. End









Problem 6: Selection sort

Solution:

1. Start
2. Make array int arr[n] and insert values in it
3. for i-> 0 to n-2:

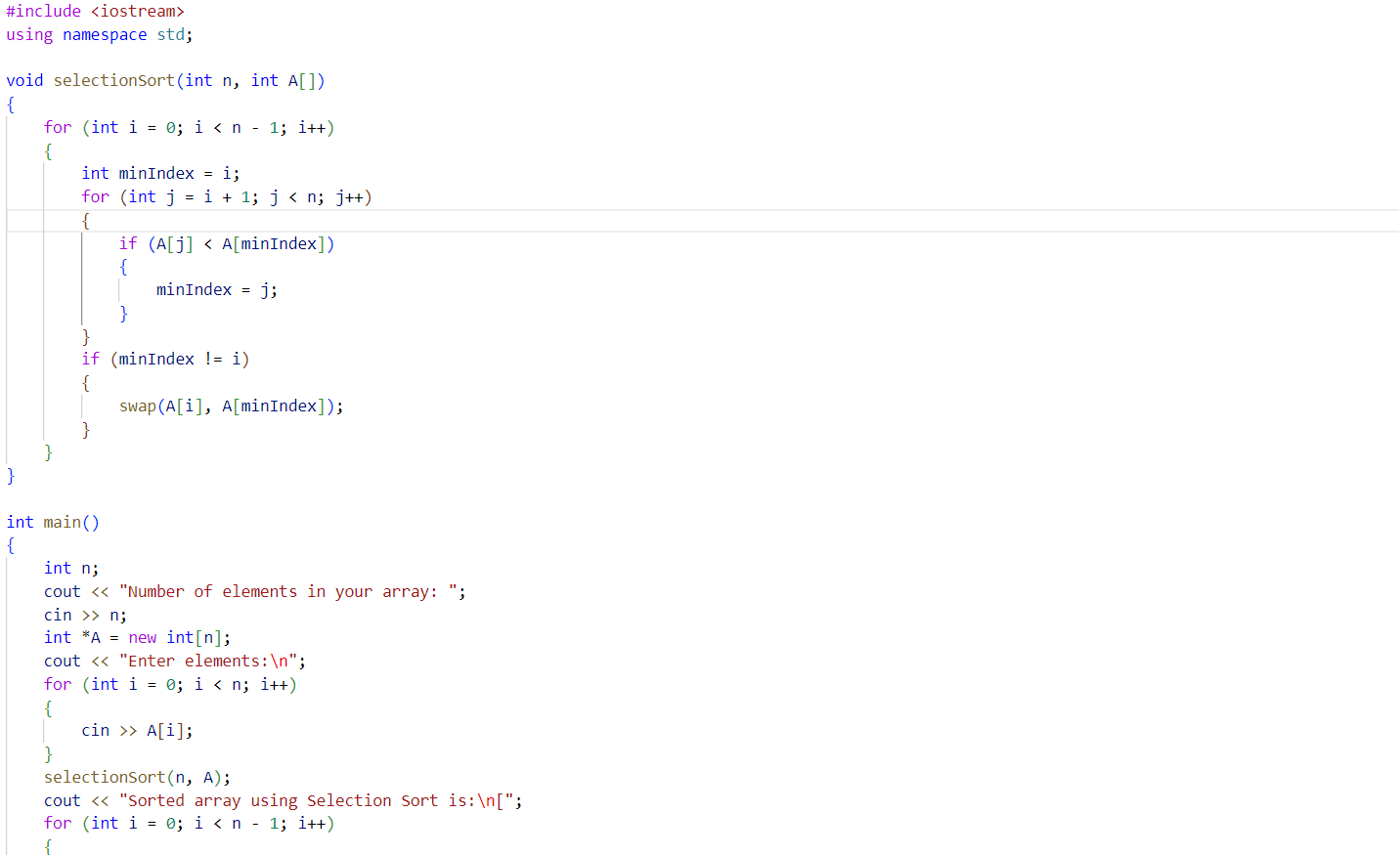
set minIndex = i

for j = i+1 to n-1:

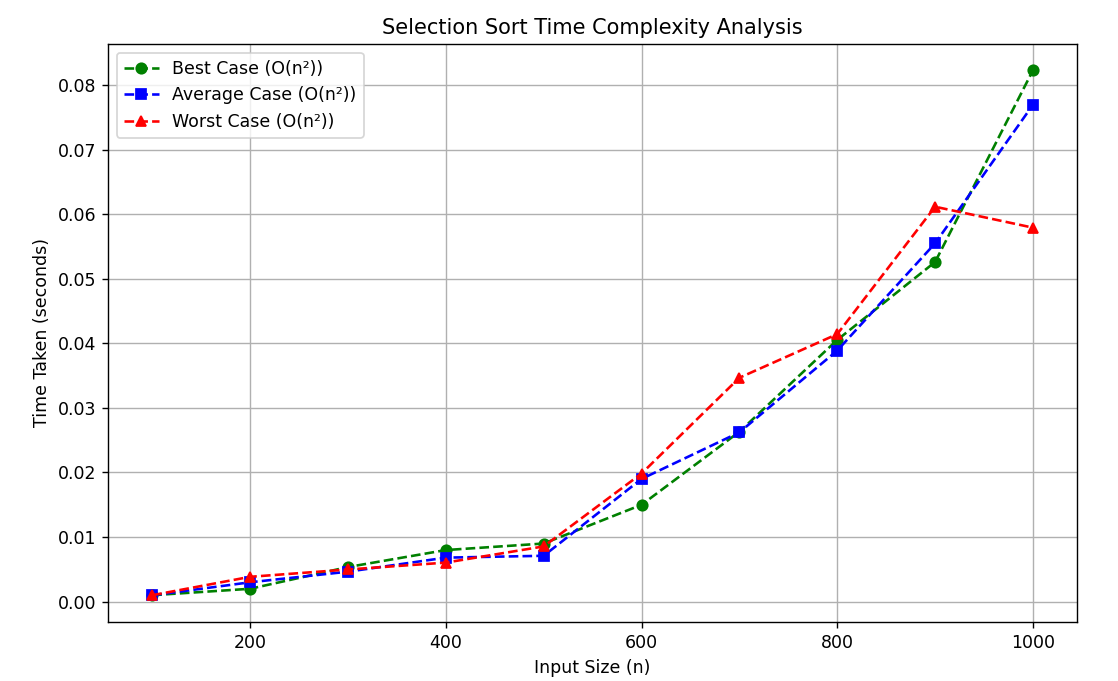
if arr[j] < arr[minIndex], update minIndex = j

swap(arr[i], arr[minIndex])

1. print sorted array arr[n]
2. End







Lab 3

Problem: Knapsack problem:

1. Sorting by profit

Solution:

1. Start
2. Read n(number of items) and W(knapsack capacity) from user
3. Create an array of items containing:

Profit[i], weight[i] 🡪 profit and weight of item i

1. Insert values into the arrays
2. Sort the items in non-increasing order of profit
3. Initialize total\_profit=0 and W\_bought = 0
4. For each item in sorted order:

If W\_bought + weight[i] <=W:

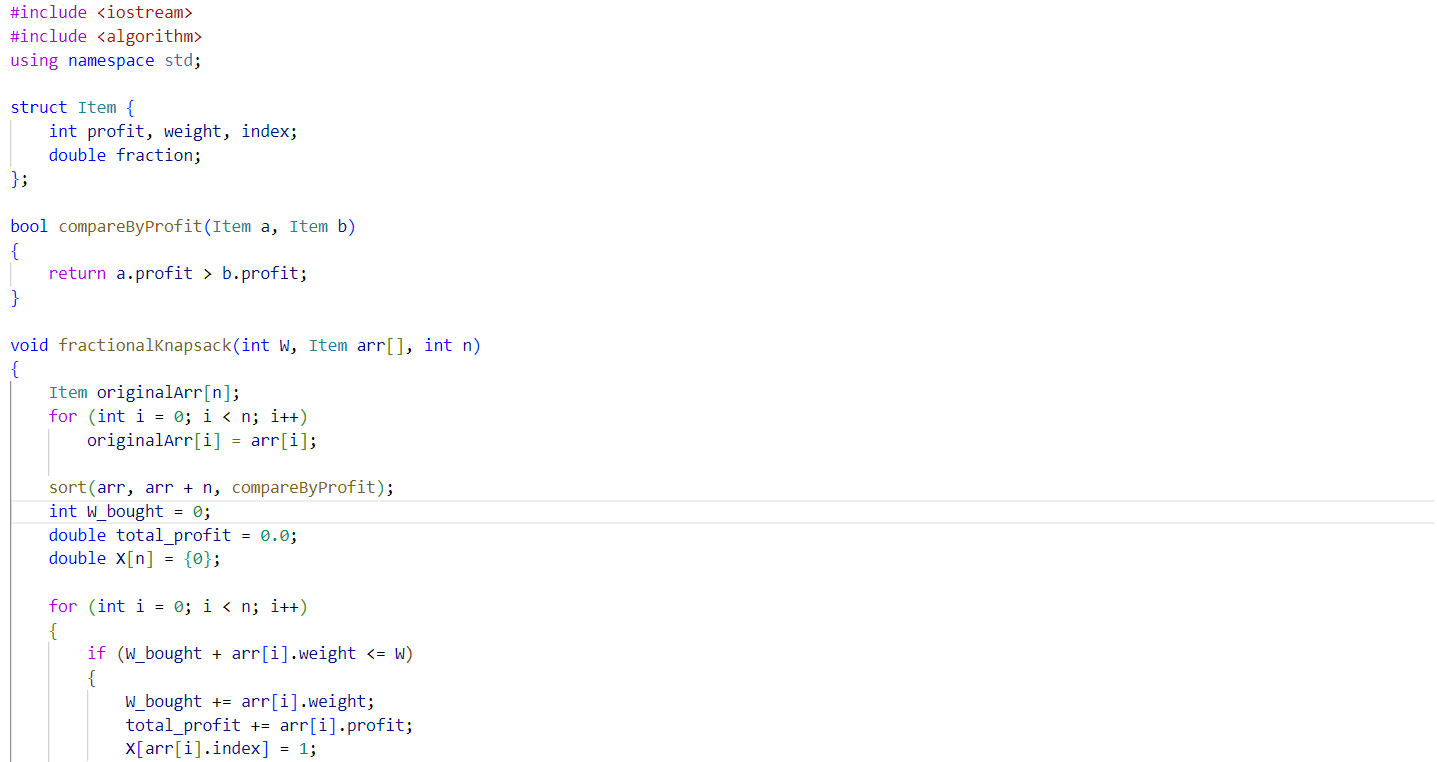
Take the full item, update total\_profit and W\_bought

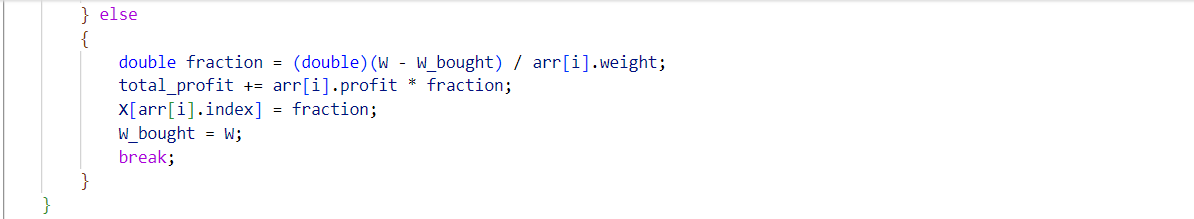
Else:

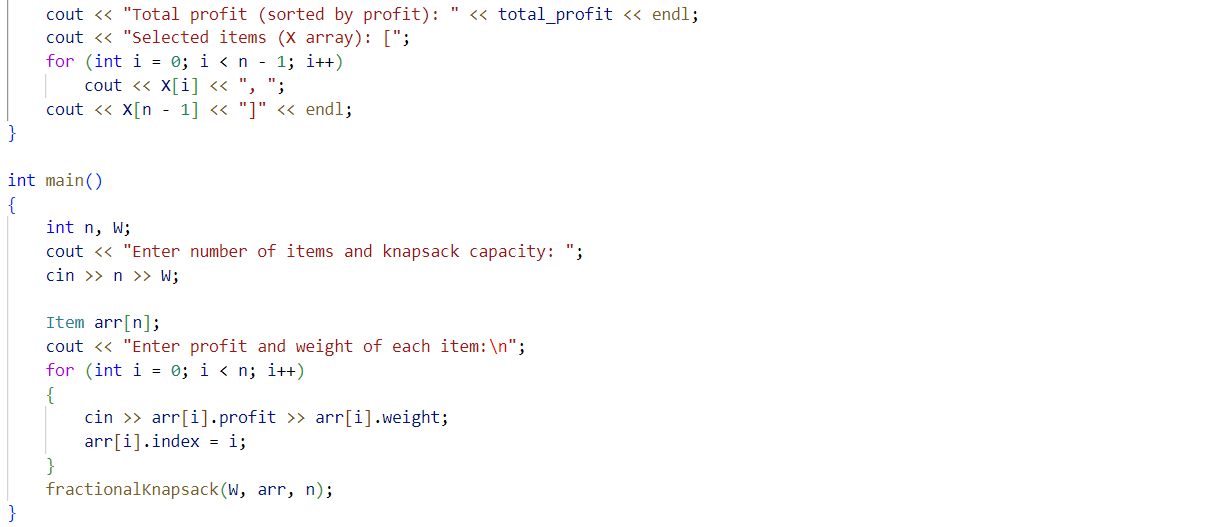
Take a fraction of the item to fill the remaining capacity

Update total\_profit and break the loop

1. Print total\_profit and array X (which indicates selected items)
2. End







1. Sorting by weight

Solution:

1. Start
2. Read n(number of items) and W(knapsack capacity) from user
3. Create an array of items containing:

Profit[i], weight[i] 🡪 profit and weight of item i

1. Insert values into the arrays
2. Sort the items in non-decreasing order of weight
3. Initialize total\_profit=0 and W\_bought = 0
4. For each item in sorted order:

If W\_bought + weight[i] <=W:

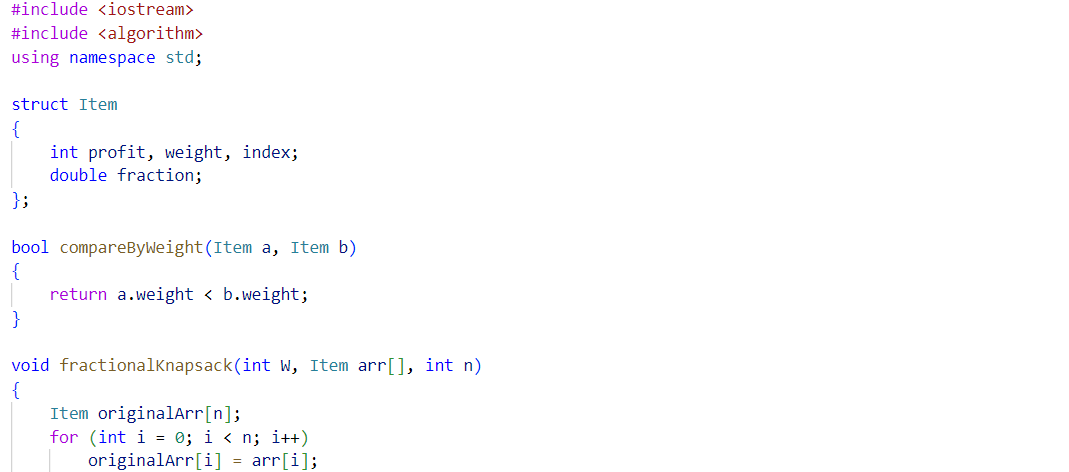
Take the full item, update total\_profit and W\_bought

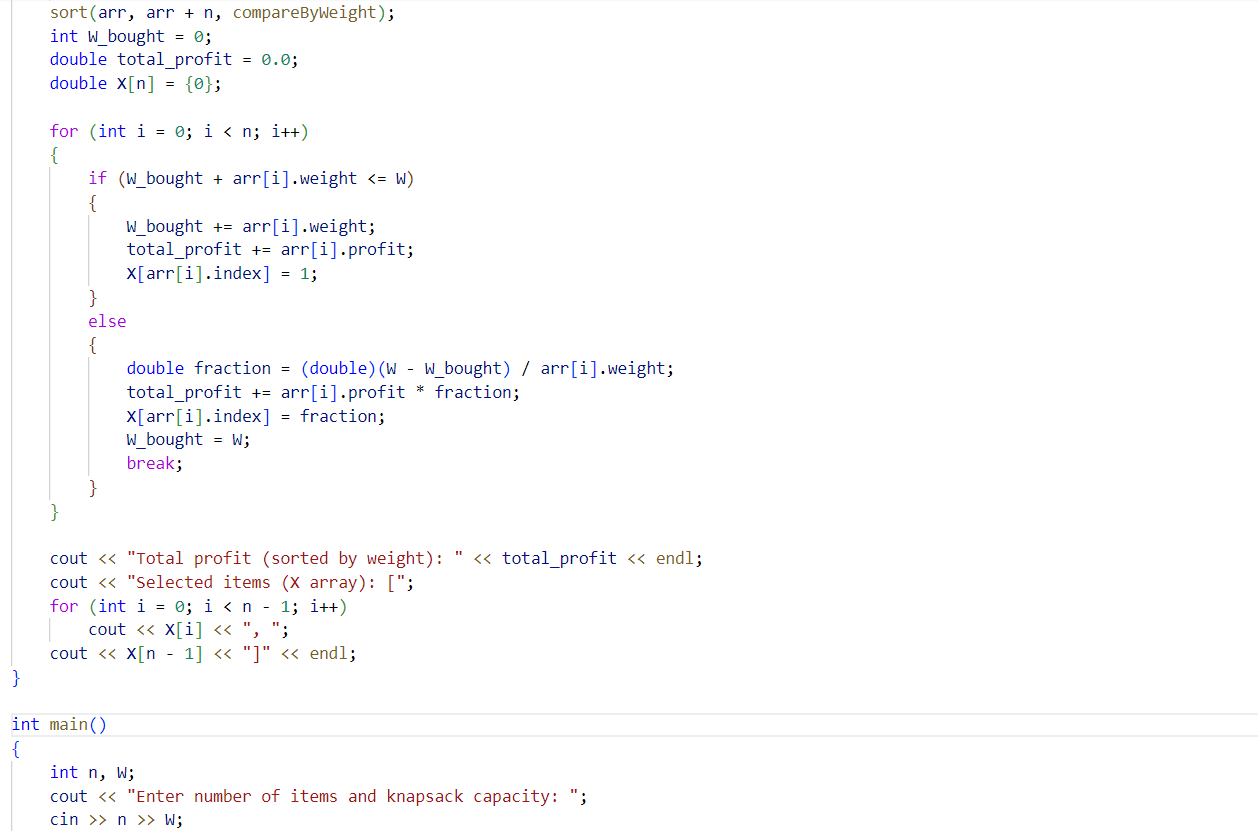
Else:

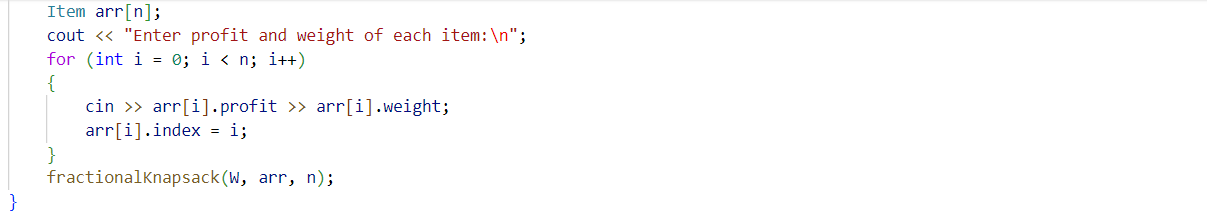
Take a fraction of the item to fill the remaining capacity

Update total\_profit and break the loop

1. Print total\_profit and array X (which indicates selected items)
2. End







1. Sorting on Profit/weight

Solution:

1. Start
2. Read n(number of items) and W(knapsack capacity) from user
3. Create an array of items containing:

Profit[i], weight[i] 🡪 profit and weight of item i

1. Insert values into the arrays
2. Compute profit/weight ratio of items
3. Sort the items in non-increasing order of profit/weight
4. Initialize total\_profit=0 and W\_bought = 0
5. For each item in sorted order:

If W\_bought + weight[i] <=W:

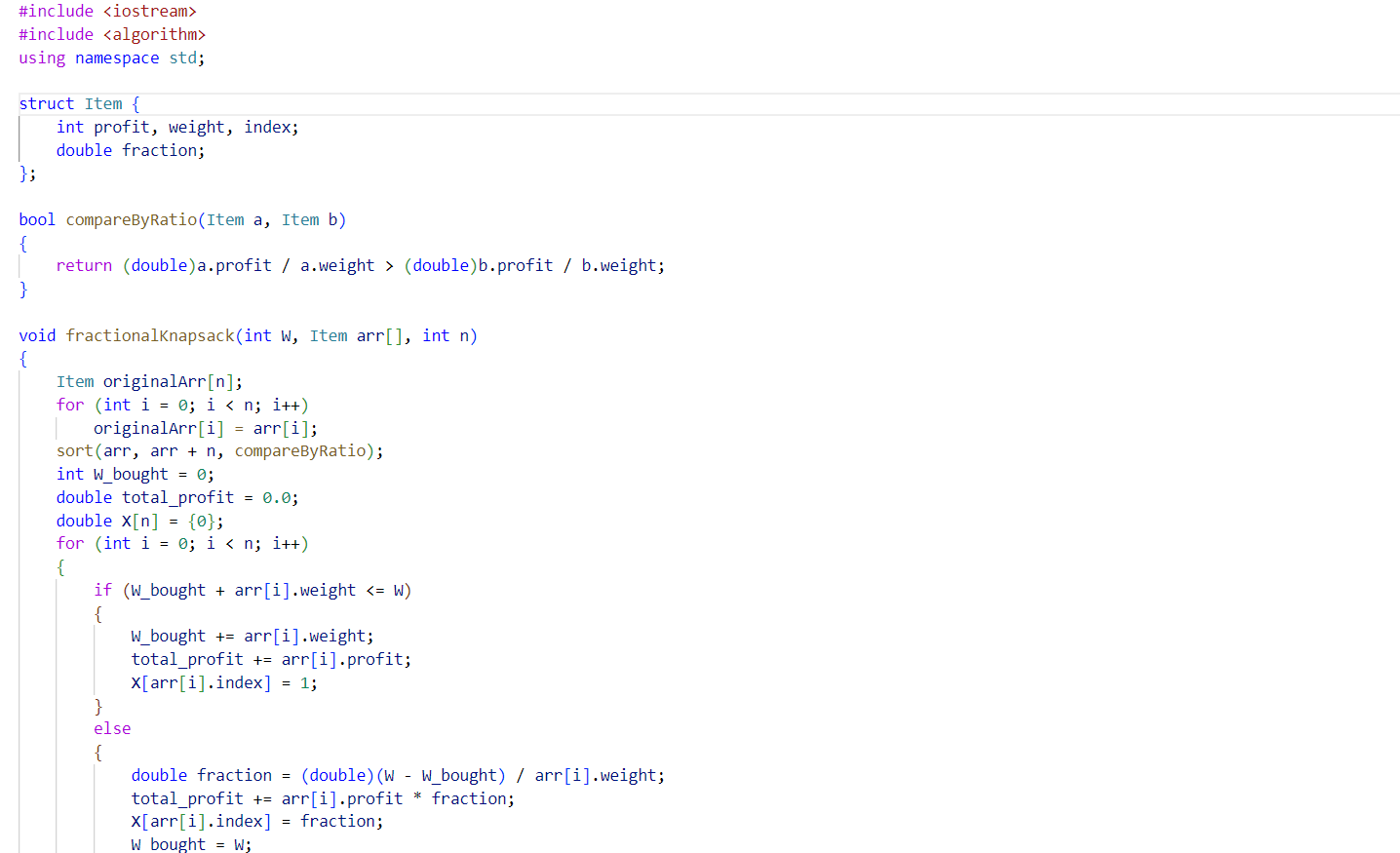
Take the full item, update total\_profit and W\_bought

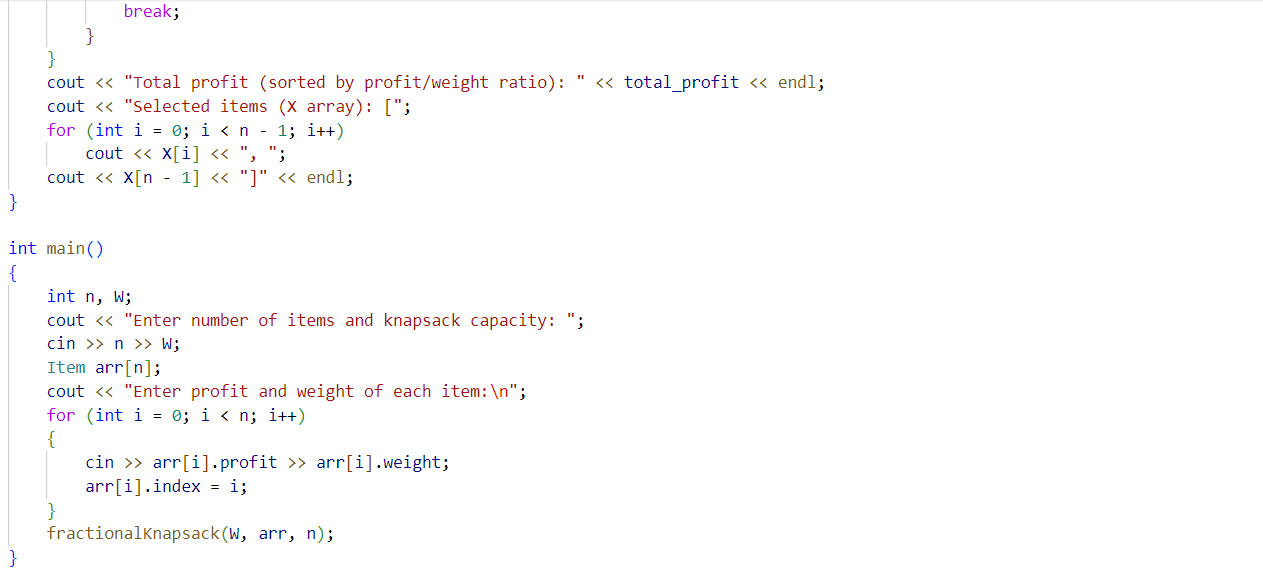
Else:

Take a fraction of the item to fill the remaining capacity

Update total\_profit and break the loop

1. Print total\_profit and array X (which indicates selected items)
2. End





Lab 5

Problem 1: Activity selection problem

Solution:

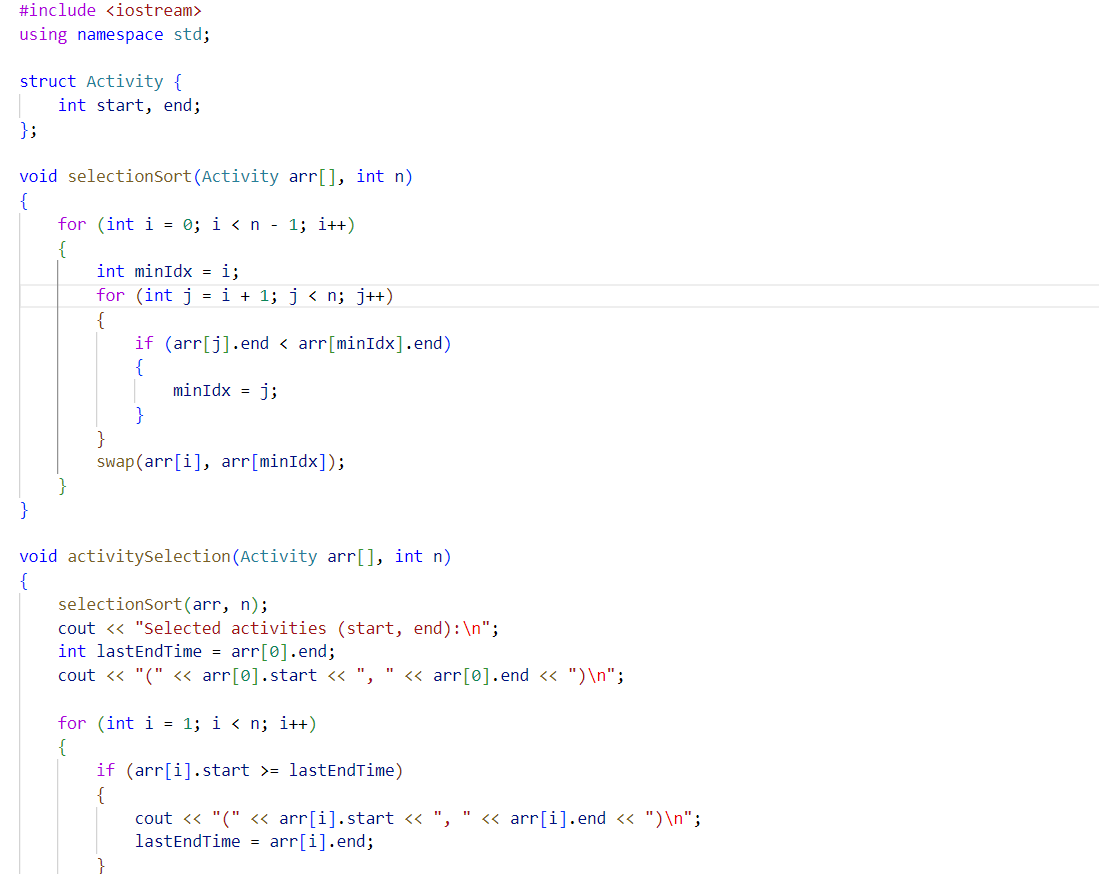
1. Start
2. Read n (number of activities) from the user
3. Create two arrays:

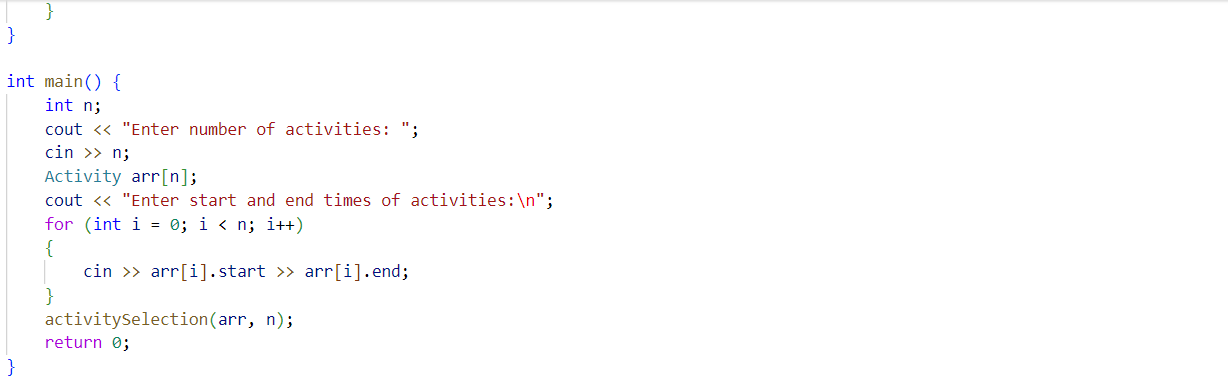
Start[i], finish[i] 🡪 start and finish times of activity i

1. Insert values into these arrays
2. Sort the activities in non-decreasing order of finish time
3. Select the first activity and store it in the result
4. For each remaining activity:

If the start time of the current activity is greater than or equal to the finish time of the previously selected activity, select it

1. Print the maximum number of activities selected
2. End





Problem 2: Djkstra’s algorithm

Solution:

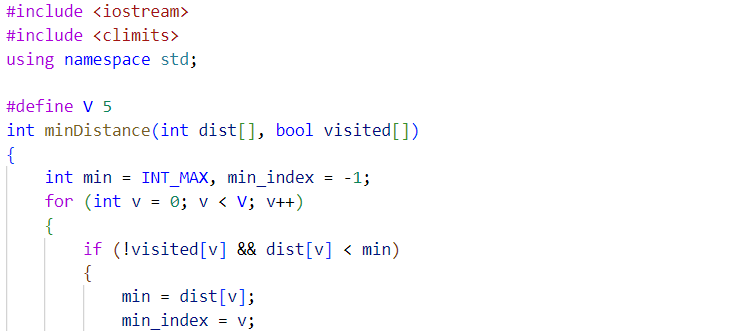
1. Start
2. Read n(number of vertices) and m(number of edges) from the user
3. Create an adjacency matrix or adjacency list representation of the graph
4. Read the m edges, each containing:

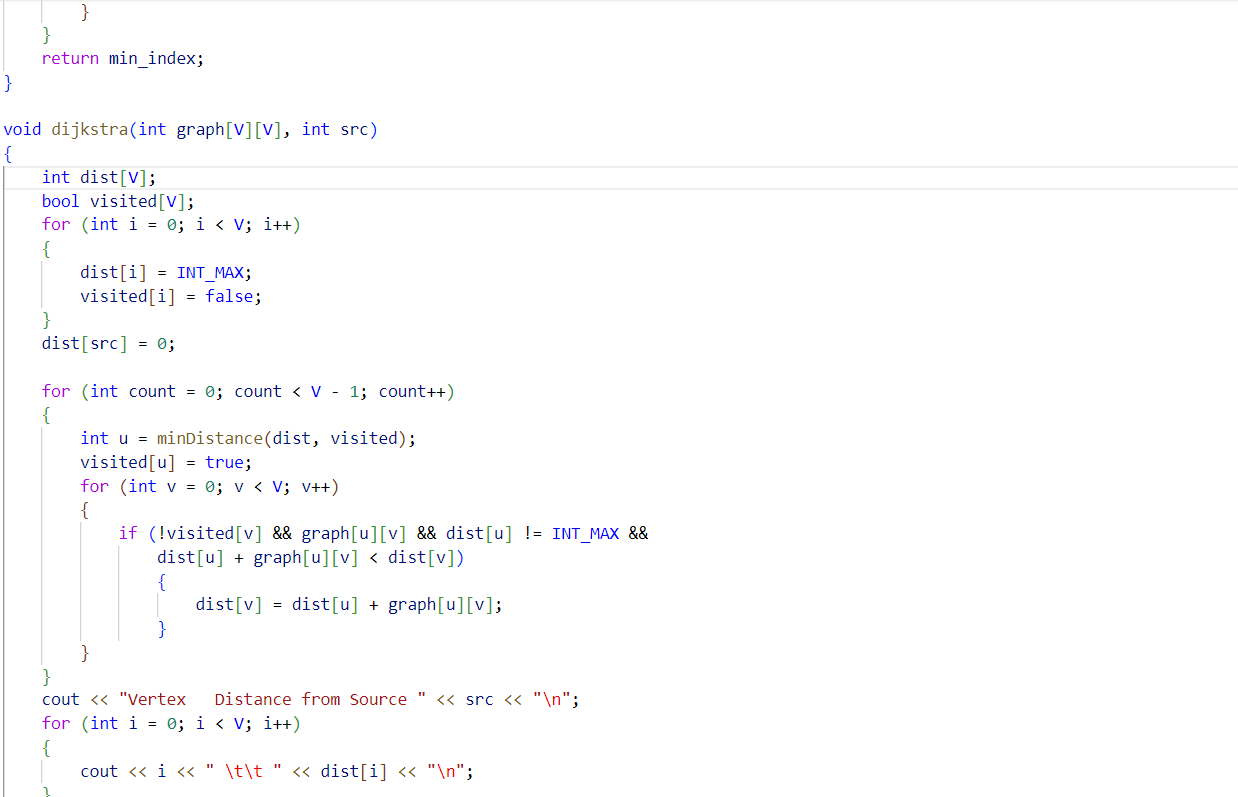
u 🡪 start vertex

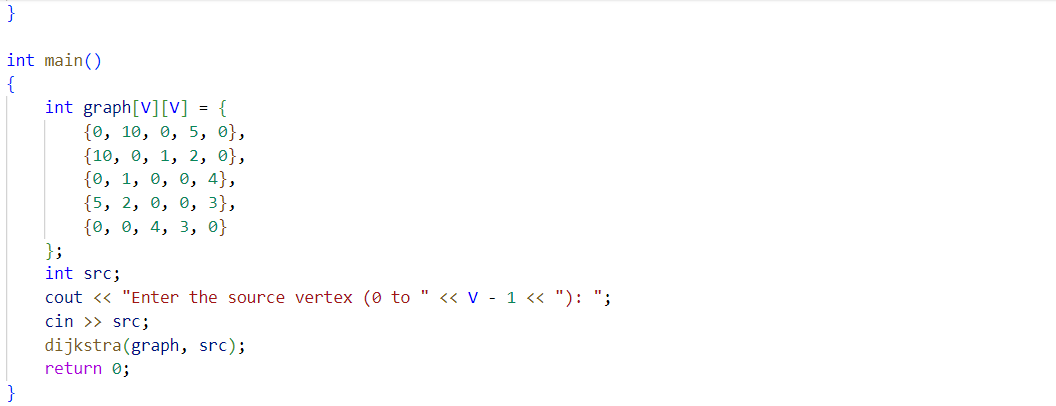
v 🡪 end vertex

w 🡪 weight of the edge (u,v)

1. Read source(starting vertex)
2. Initialize:
   1. Distance array dist[] with infinity for all vertices except source, which is set to 0
   2. A Boolean visited[] array to track processed nodes
3. Use a priority queue (min-heap) to always process the node with the smallest distance
4. While the queue is not empty:
   1. Extract the node with the minimum distance
   2. Update distances of all its adjacent vertices if a shorter path is found
5. Print the shortest distance from source to all other nodes
6. End



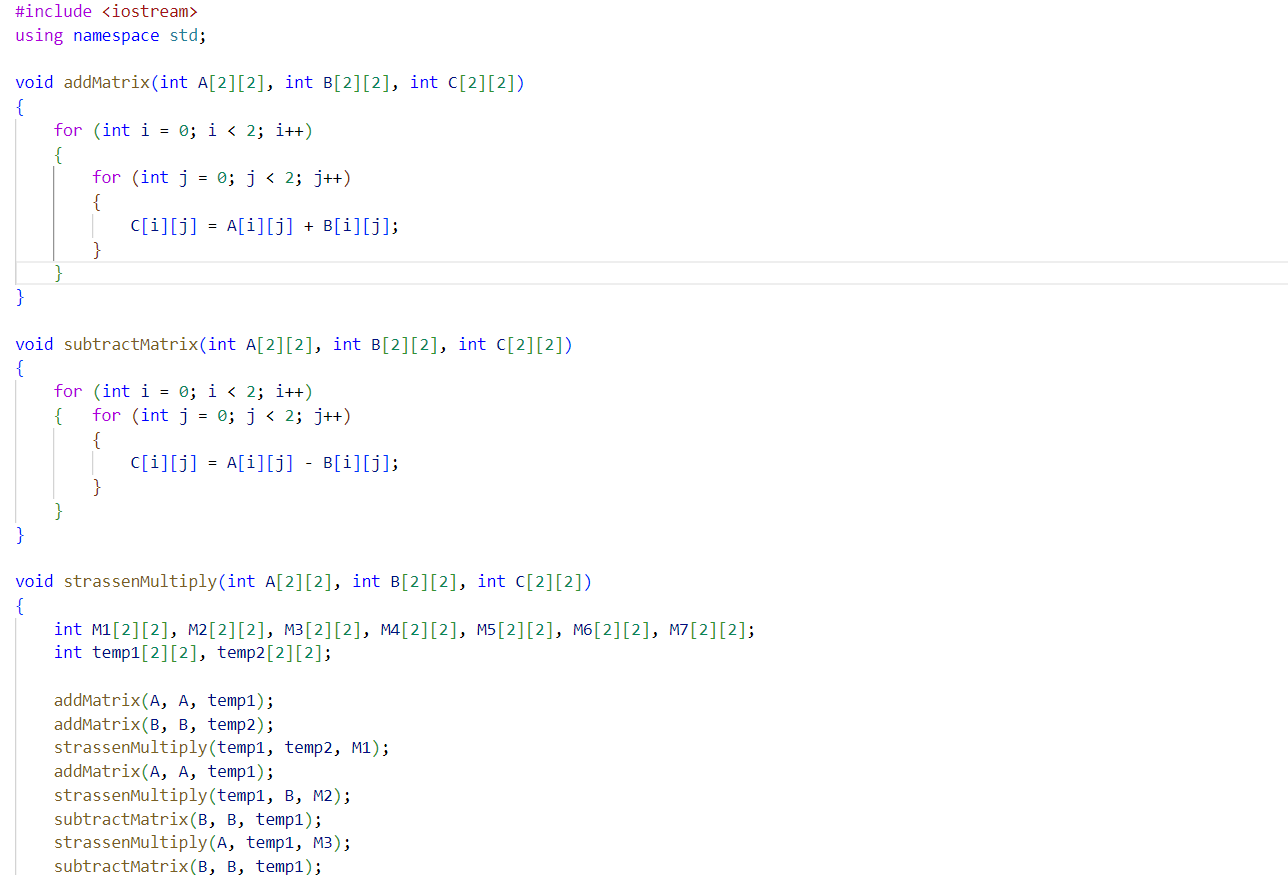


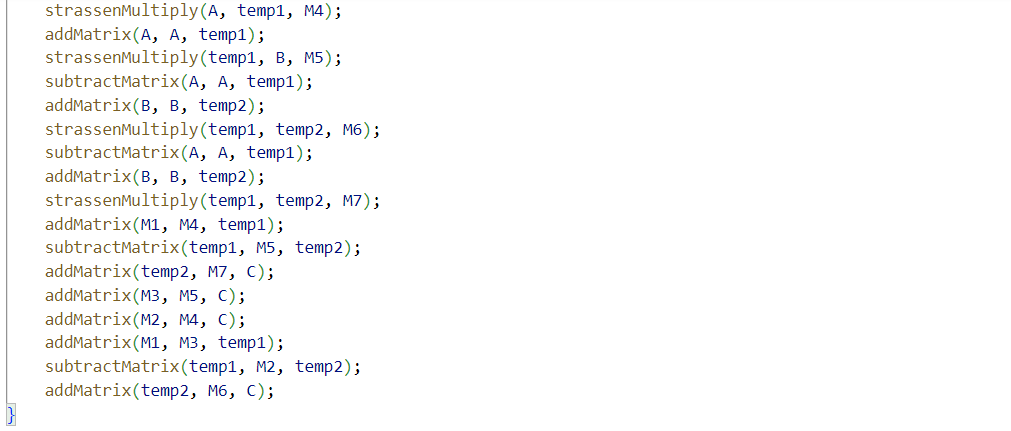


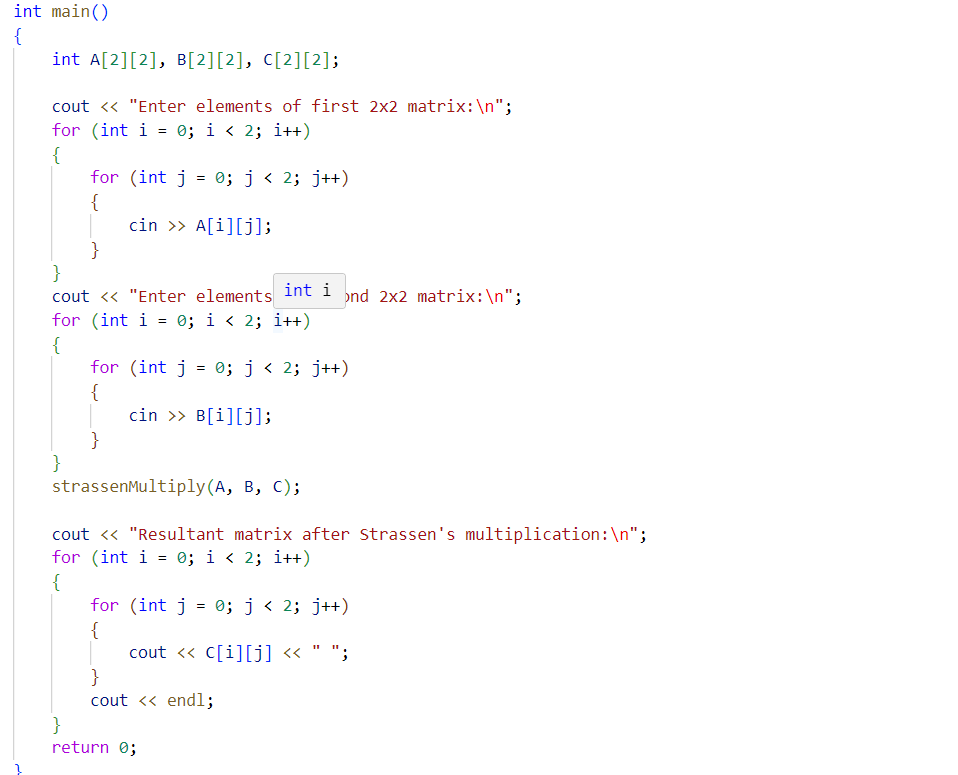
Problem 3: Strassen’s multiplication

Solution:

1. Start
2. Read n (size of square matrices, assumed to be of the type 2k)
3. Read two matrices A and B of sizes n\*n
4. If n = 1, return A\*B (base case)
5. Divide A and B into four submatrices each
6. Compute the 7 strassen products using recursive calls
7. Compute the final submatrices of the result matrix
8. Combine these submatrices to form the result matrix C
9. Print the resultant matrix C
10. End







Lab 6

Problem: Kruskal’s algorithm

Solution:

1. Start
2. Make an array of edge weights
3. Make min-heap
4. Delete the top edge weight which is the least
5. Re-heapify the remaining array
6. While i <=n-1 && heap is not empty

k=find(u) and l=find(v)

From array of edgeweights, use find function on (u,v)

If ((find(u)!=(find(v)))

Use Union function on (u,v)

T[i,1] = k and T[i,2]=l

1. The mst is generated in T
2. End

