Date of the Session:___/___/_____ **Time of the Session:_______to________**

EX – 3 Implementing Programs on Sorting

Prerequisites:

- Basics of Data Structures and C Programming.
- Basic knowledge about Sorting Techniques.

Pre-Lab:

1) Write a Divide and Conquer algorithm to find the minimum distance between the pair of the points given in an array. Find the time complexity of the algorithm.

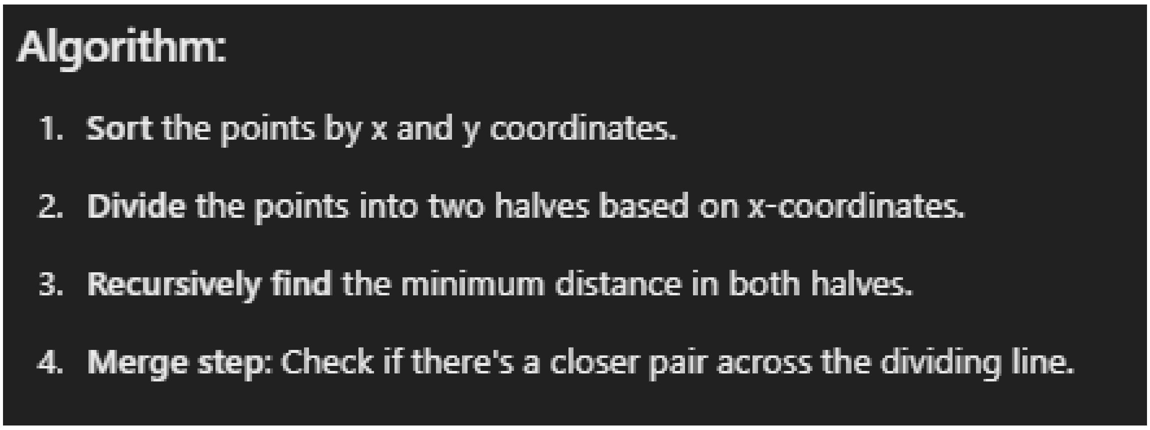
P[] = {{2, 3}, {12, 30}, {40, 50}, {5, 1}, {12, 10}, {3, 4}}

```
#include <stdio.h>

#include <math.h>

#include <stdlib.h>


typedef struct {

   int x, y;

} Point;


double dist(Point p1, Point p2) {

   return sqrt((p1.x - p2.x) * (p1.x - p2.x) + (p1.y - p2.y) * (p1.y - p2.y));

}


double min(double a, double b) {

   return (a < b) ? a : b;

}
```

```
int compareX(const void *a, const void *b) {
    return ((Point*)a)->x - ((Point*)b)->x;
}

int compareY(const void *a, const void *b) {
    return ((Point*)a)->y - ((Point*)b)->y;
}

double stripClosest(Point strip[], int size, double d) {
    double min_dist = d;
    qsort(strip, size, sizeof(Point), compareY);
    for (int i = 0; i < size; i++) {
        for (int j = i + 1; j < size && (strip[j].y - strip[i].y) < min_dist; j++) {
            min_dist = min(min_dist, dist(strip[i], strip[j]));
        }
    }
    return min_dist;
}

double closestUtil(Point Px[], Point Py[], int n) {
    if (n <= 3) {
        double min_dist = INFINITY;
        for (int i = 0; i < n; i++) {
            for (int j = i + 1; j < n; j++) {
                min_dist = min(min_dist, dist(Px[i], Px[j]));
            }
        }
    return min_dist;
}

int mid = n / 2;
Point midPoint = Px[mid];

Point Qx[mid], Rx[n - mid];
Point Qy[n], Ry[n];

int li = 0, ri = 0, liq = 0, riq = 0;
for (int i = 0; i < n; i++) {
    if (Px[i].x <= midPoint.x) {
        Qx[li++] = Px[i];
    } else {
        Rx[ri++] = Px[i];
    }

    if (Px[i].x <= midPoint.x) {
        Qy[liq++] = Py[i];
    } else {
        Ry[riq++] = Py[i];
    }
}

double dl = closestUtil(Qx, Qy, li);
double dr = closestUtil(Rx, Ry, ri);
double d = min(dl, dr);
```

```
    Point strip[n];
    int j = 0;
    for (int i = 0; i < n; i++) {
        if (abs(Py[i].x - midPoint.x) < d) {
            strip[j++] = Py[i];
        }
    }

    return min(d, stripClosest(strip, j, d));
}

double closestPair(Point P[], int n) {
    Point Px[n], Py[n];

    for (int i = 0; i < n; i++) {
        Px[i] = P[i];
        Py[i] = P[i];
    }

    qsort(Px, n, sizeof(Point), compareX);
    qsort(Py, n, sizeof(Point), compareY);

    return closestUtil(Px, Py, n);
}

int main() {
    Point P[] = {{2, 3}, {12, 30}, {40, 50}, {5, 1}, {12, 10}, {3, 4}};
    int n = sizeof(P) / sizeof(P[0]);

    printf("The minimum distance is: %lf\n", closestPair(P, n));

    return 0;
}
```

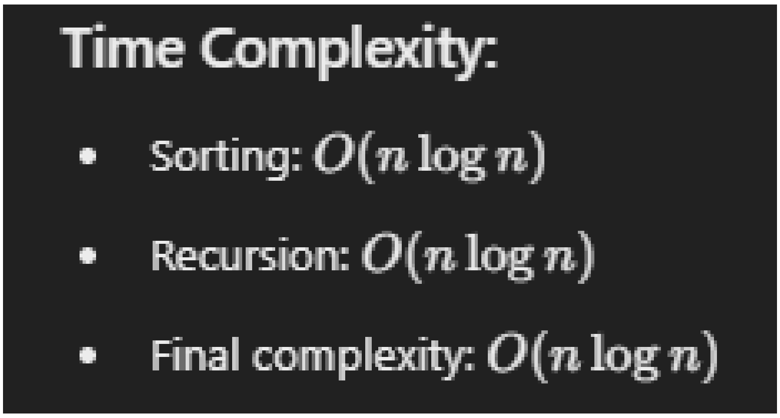
Algorithm



Algorithm:

1. Sort the points by x and y coordinates.
2. Divide the points into two halves based on x-coordinates.
3. Recursively find the minimum distance in both halves.
4. **Merge step:** Check if there's a closer pair across the dividing line.

Time Complexity



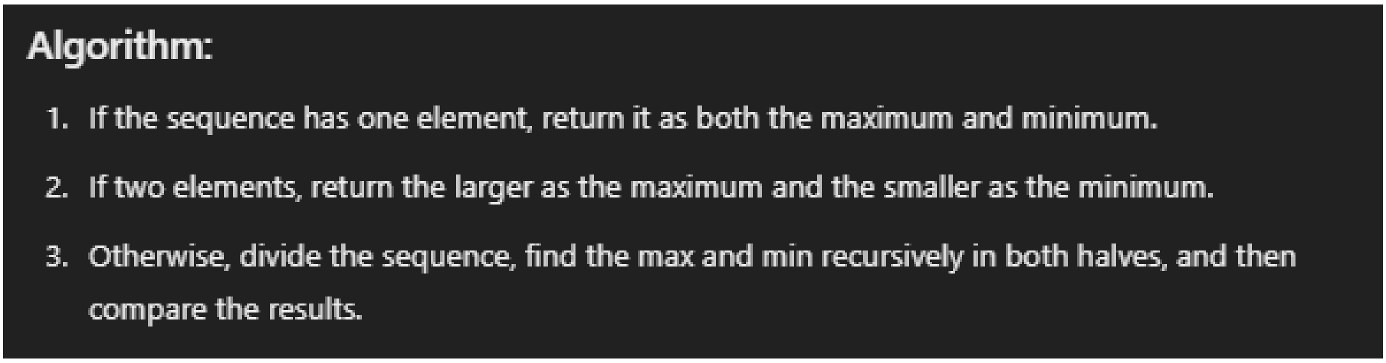
Time Complexity:

- Sorting: $O(n \log n)$
- Recursion: $O(n \log n)$
- Final complexity: $O(n \log n)$

2) Write a divide and conquer algorithm for finding the maximum and minimum in the sequence of numbers. Find the time complexity.

```
#include <stdio.h>

void find_max_min(int arr[], int low, int high, int *max, int *min) {
    if (low == high) {
        *max = arr[low];
        *min = arr[low];
    } else if (high == low + 1) {
        if (arr[low] > arr[high]) {
            *max = arr[low];
            *min = arr[high];
        } else {
            *max = arr[high];
            *min = arr[low];
        }
    } else {
        int mid = (low + high) / 2;
        int max1, min1, max2, min2;

        find_max_min(arr, low, mid, &max1, &min1);
        find_max_min(arr, mid + 1, high, &max2, &min2);

        *max = (max1 > max2) ? max1 : max2;
        *min = (min1 < min2) ? min1 : min2;
    }
}

int main() {
    int arr[] = {12, 5, 7, 9, 15, 3, 11, 10};
    int n = sizeof(arr) / sizeof(arr[0]);
    int max, min;

    find_max_min(arr, 0, n - 1, &max, &min);

    printf("Maximum: %d\n", max);
    printf("Minimum: %d\n", min);
    return 0;
}
```

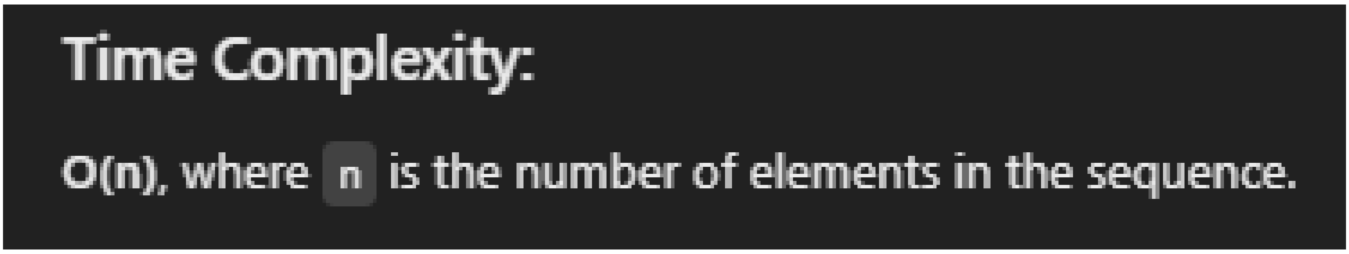
Algorithm



Algorithm:

1. If the sequence has one element, return it as both the maximum and minimum.
2. If two elements, return the larger as the maximum and the smaller as the minimum.
3. Otherwise, divide the sequence, find the max and min recursively in both halves, and then compare the results.

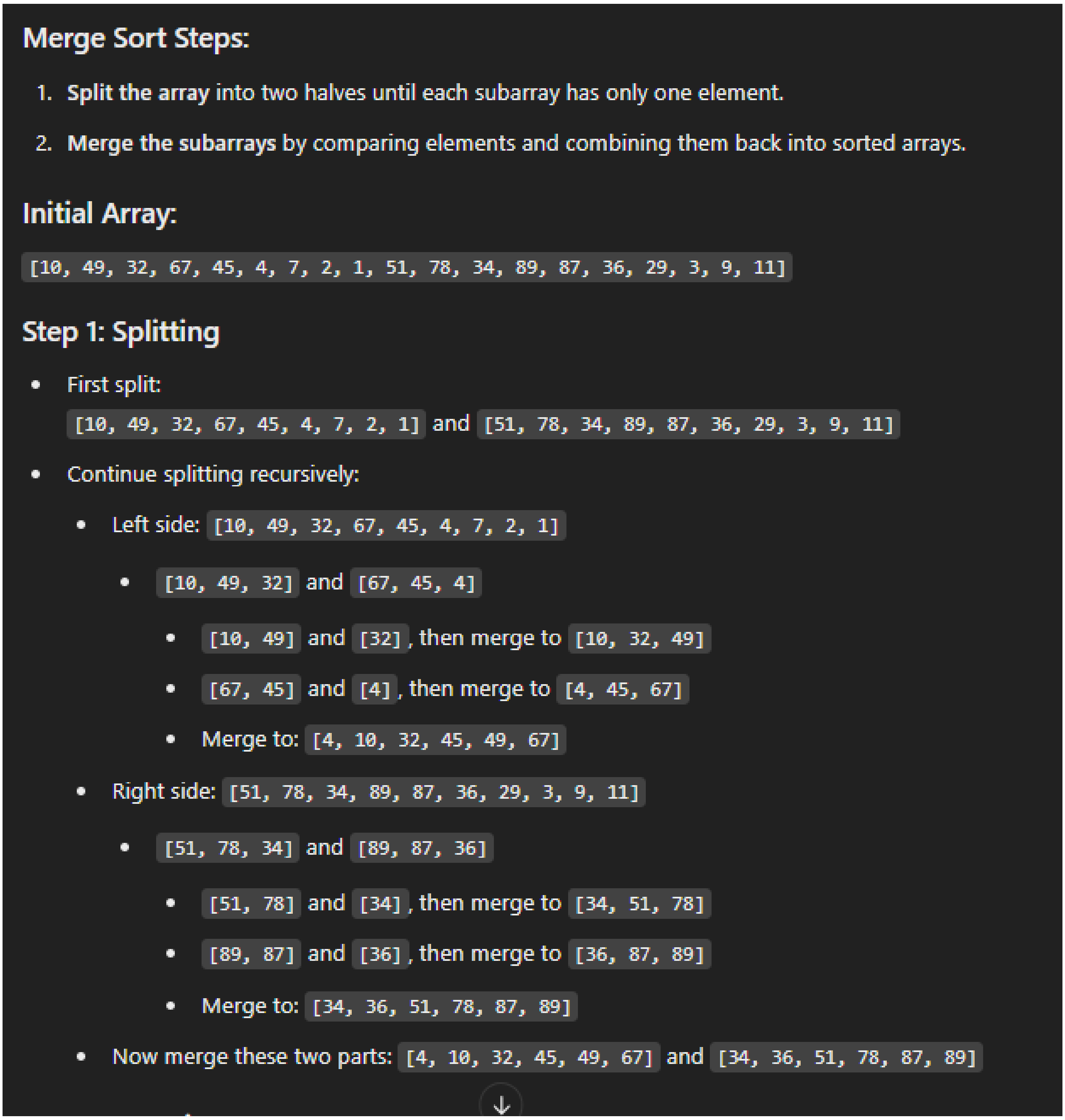
Time Complexity



Time Complexity:

O(n), where `n` is the number of elements in the sequence.

3) Trace out the output of the following using Merge sort.10, 49, 32, 67, 45, 4, 7, 2, 1, 51, 78, 34, 89, 87, 36, 29, 3, 9, 11.



Merge Sort Steps:

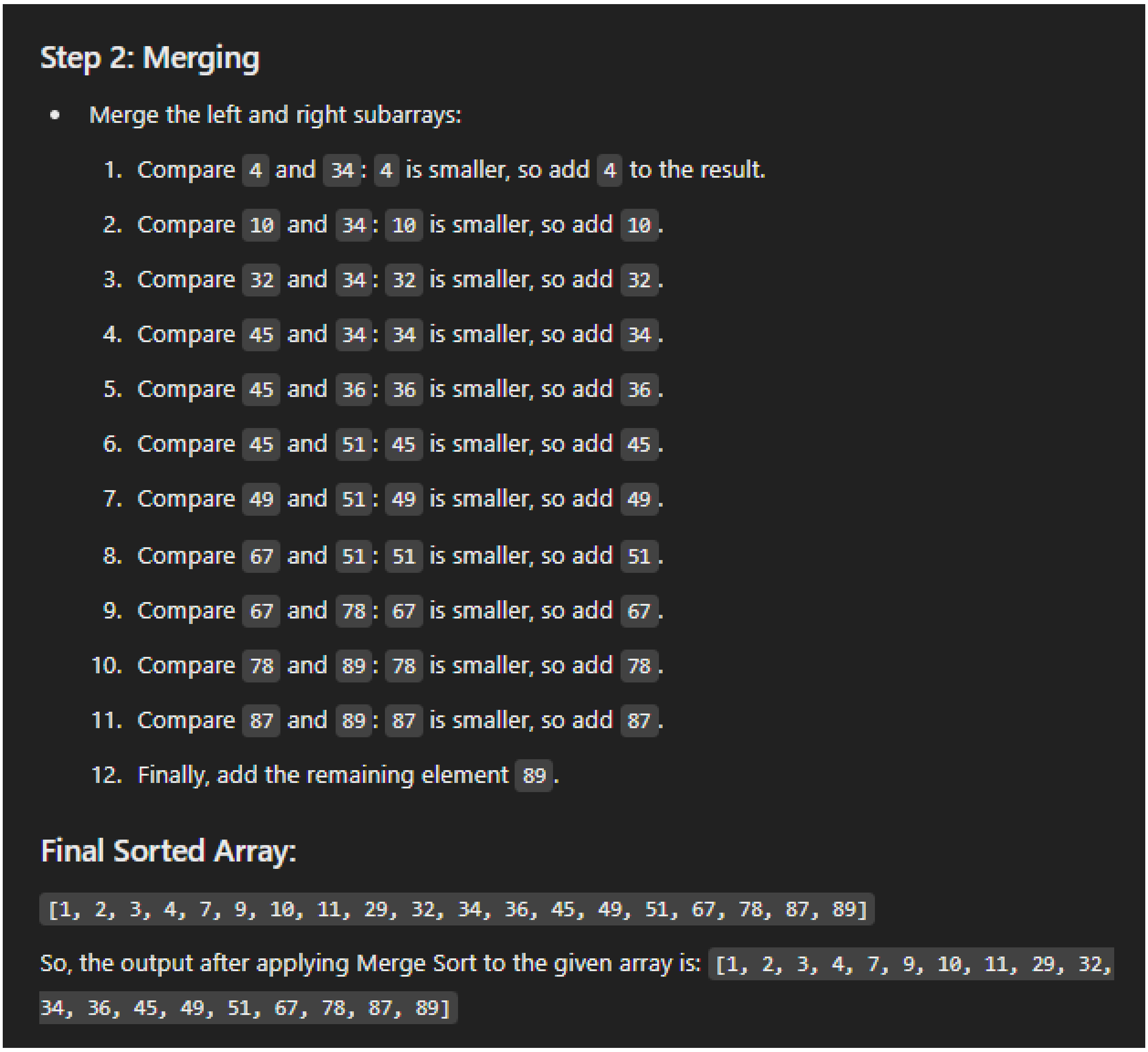
1. **Split the array** into two halves until each subarray has only one element.
2. **Merge the subarrays** by comparing elements and combining them back into sorted arrays.

Initial Array:

`[10, 49, 32, 67, 45, 4, 7, 2, 1, 51, 78, 34, 89, 87, 36, 29, 3, 9, 11]`

Step 1: Splitting

- First split:
  `[10, 49, 32, 67, 45, 4, 7, 2, 1]` and `[51, 78, 34, 89, 87, 36, 29, 3, 9, 11]`
- Continue splitting recursively:
  - Left side: `[10, 49, 32, 67, 45, 4, 7, 2, 1]`
    - `[10, 49, 32]` and `[67, 45, 4]`
      - `[10, 49]` and `[32]`, then merge to `[10, 32, 49]`
      - `[67, 45]` and `[4]`, then merge to `[4, 45, 67]`
      - Merge to: `[4, 10, 32, 45, 49, 67]`
  - Right side: `[51, 78, 34, 89, 87, 36, 29, 3, 9, 11]`
    - `[51, 78, 34]` and `[89, 87, 36]`
      - `[51, 78]` and `[34]`, then merge to `[34, 51, 78]`
      - `[89, 87]` and `[36]`, then merge to `[36, 87, 89]`
      - Merge to: `[34, 36, 51, 78, 87, 89]`
  - Now merge these two parts: `[4, 10, 32, 45, 49, 67]` and `[34, 36, 51, 78, 87, 89]`



Step 2: Merging

- Merge the left and right subarrays:
    1. Compare `4` and `34`: `4` is smaller, so add `4` to the result.
    2. Compare `10` and `34`: `10` is smaller, so add `10`.
    3. Compare `32` and `34`: `32` is smaller, so add `32`.
    4. Compare `45` and `34`: `34` is smaller, so add `34`.
    5. Compare `45` and `36`: `36` is smaller, so add `36`.
    6. Compare `45` and `51`: `45` is smaller, so add `45`.
    7. Compare `49` and `51`: `49` is smaller, so add `49`.
    8. Compare `67` and `51`: `51` is smaller, so add `51`.
    9. Compare `67` and `78`: `67` is smaller, so add `67`.
    10. Compare `78` and `89`: `78` is smaller, so add `78`.
    11. Compare `87` and `89`: `87` is smaller, so add `87`.
    12. Finally, add the remaining element `89`.

Final Sorted Array:

`[1, 2, 3, 4, 7, 9, 10, 11, 29, 32, 34, 36, 45, 49, 51, 67, 78, 87, 89]`

So, the output after applying Merge Sort to the given array is: `[1, 2, 3, 4, 7, 9, 10, 11, 29, 32, 34, 36, 45, 49, 51, 67, 78, 87, 89]`

In-Lab:

1) Harry’s Aunt and family treat him badly and make him work all the time. Dudley, his cousin got homework from school and he as usual handed it over to Harry but Harry has a lot of work and his own homework to do.
The homework is to solve the problems which are numbered in numerical he tries to solve random question after solving random questions he did not put those questions in order Dudley will return in a time of n*logn Harry has to arrange them as soon as possible. Help Harry to solve this problem so that he can go on and do his own homework.

Example

Input

9
15,5,24,8,1,3,16,10,20

Output

1, 3, 5, 8, 10, 15, 16, 20, 24

Source code:

```
#include <stdio.h>

void quicksort(int arr[], int low, int high) {
    if (low < high) {
        int pivot = arr[high];
        int i = (low - 1);

        for (int j = low; j < high; j++) {
            if (arr[j] < pivot) {
                i++;
                int temp = arr[i];
                arr[i] = arr[j];
                arr[j] = temp;
            }
        }
        int temp = arr[i + 1];
        arr[i + 1] = arr[high];
        arr[high] = temp;

        quicksort(arr, low, i);
        quicksort(arr, i + 2, high);
    }
}

int main() {
    int arr[] = {15, 5, 24, 8, 1, 3, 16, 10, 20};
    int n = sizeof(arr) / sizeof(arr[0]);

    quicksort(arr, 0, n - 1);

    for (int i = 0; i < n; i++) {
        if (i != 0) {
            printf(", ");
            }
            printf("%d", arr[i]);
        }
        printf("\n");

        return 0;
    }
```

2) A group of 9 friends are playing a game, rules of the game are as follows: Each member will be assigned with a number and the sequence goes like e.g.: 7,6,10,5,9,2,1,15,7.
Now they will be sorted in ascending order in such a way that tallest one will be sorted first. Now your task is to find the order of indices based on initial position of the given sequence and print the order of indices at the end of the iteration.

Source code:

```
#include <stdio.h>

#include <stdlib.h>


typedef struct {

    int value, index;

} Element;


int compare(const void *a, const void *b) {

    return ((Element*)b)->value - ((Element*)a)->value;

}


int main() {

    int sequence[] = {7, 6, 10, 5, 9, 2, 1, 15, 7};

    int n = sizeof(sequence) / sizeof(sequence[0]);

    Element elements[n];


    for (int i = 0; i < n; i++) {

        elements[i].value = sequence[i];

        elements[i].index = i;

    }


    qsort(elements, n, sizeof(Element), compare);
    for (int i = 0; i < n; i++) {
        printf("%d ", elements[i].index);
    }

    return 0;
}
```

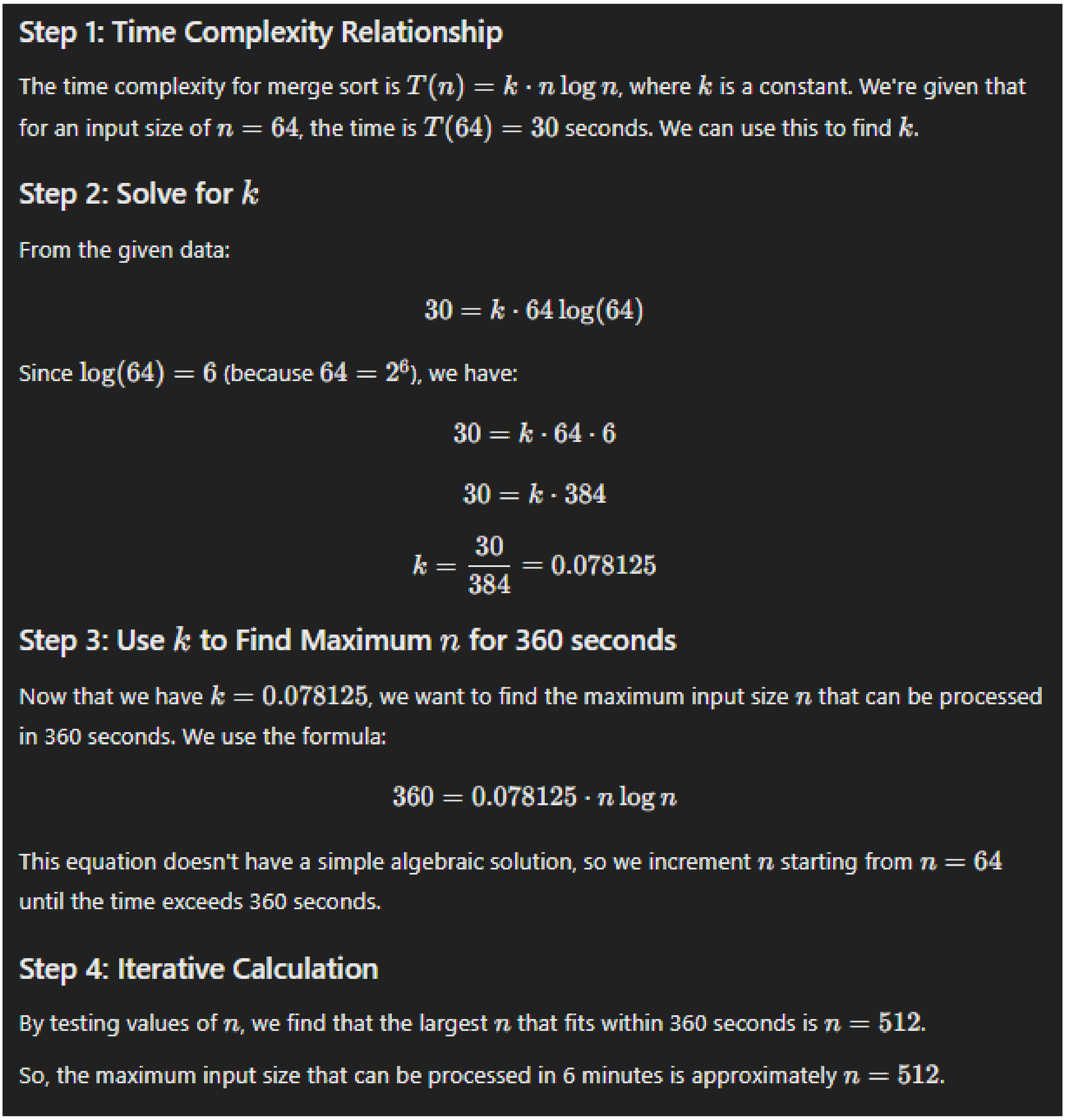
OUTPUT

```
7 2 4 0 8 1 3 5 6
```

Post-Lab:

1) Suppose a merge sort algorithm, for input size 64, takes 30 secs in the worst case. What is the maximum input size that can be calculated in 6 minutes (approximately)?

Source code:



Step 1: Time Complexity Relationship

The time complexity for merge sort is $T(n) = k \cdot n \log n$, where k is a constant. We're given that for an input size of $n = 64$, the time is $T(64) = 30$ seconds. We can use this to find k.

Step 2: Solve for k

From the given data:

$$30 = k \cdot 64 \log(64)$$

Since $\log(64) = 6$ (because $64 = 2^6$), we have:

$$30 = k \cdot 64 \cdot 6$$

$$30 = k \cdot 384$$

$$k = \frac{30}{384} = 0.078125$$

Step 3: Use k to Find Maximum n for 360 seconds

Now that we have $k = 0.078125$, we want to find the maximum input size n that can be processed in 360 seconds. We use the formula:

$$360 = 0.078125 \cdot n \log n$$

This equation doesn't have a simple algebraic solution, so we increment n starting from $n = 64$ until the time exceeds 360 seconds.

Step 4: Iterative Calculation

By testing values of n, we find that the largest n that fits within 360 seconds is $n = 512$.

So, the maximum input size that can be processed in 6 minutes is approximately $n = 512$.

2) Chris and Scarlett were playing a block sorting game where Scarlett challenged Chris that he has to sort the blocks which arranged in random order. And Scarlett puts a restriction that he should not use reference of first, median and last blocks to sort, and after sorting one block with reference to other block, for next iteration he must choose another block as the reference not the same block (random pivot).

Now, Chris wants help from you to sort the blocks. He wanted to sort them in a least time. Help him with the least time complexity sorting algorithm.

Input format

First line of input contains the number of test cases.

Next t lines of input contain

The number of blocks provided by Scarlett.

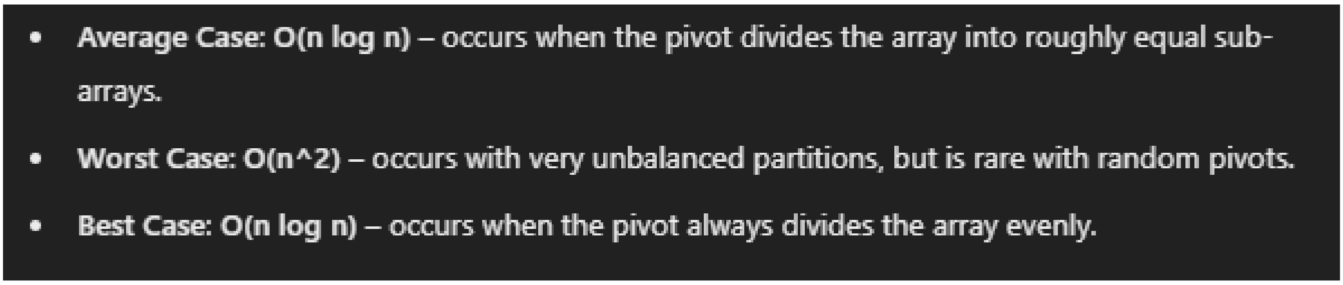
The array of blocks.

Source code:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

void swap(int *a, int *b) {
   int temp = *a;
   *a = *b;
   *b = temp;
}

int partition(int arr[], int low, int high) {
   int pivotIndex = low + rand() % (high - low + 1);
   int pivot = arr[pivotIndex];
   swap(&arr[pivotIndex], &arr[high]);
   int i = low - 1;
   for (int j = low; j < high; j++) {
      if (arr[j] < pivot) {
         i++;
         swap(&arr[i], &arr[j]);
      }
   }
   swap(&arr[i + 1], &arr[high]);
   return i + 1;
}

void randomizedQuickSort(int arr[], int low, int high) {
   if (low < high) {
      int pivotIndex = partition(arr, low, high);
      randomizedQuickSort(arr, low, pivotIndex - 1);
      randomizedQuickSort(arr, pivotIndex + 1, high);
   }
}

int main() {
   srand(time(NULL));
   int t;
   scanf("%d", &t);
```

```
    for (int i = 0; i < t; i++) {
        int n;
        scanf("%d", &n);
        int arr[n];

        for (int j = 0; j < n; j++) {
            scanf("%d", &arr[j]);
        }

        randomizedQuickSort(arr, 0, n - 1);

        for (int j = 0; j < n; j++) {
            printf("%d ", arr[j]);
        }
        printf("\n");
    }

    return 0;
}
```

Time Complexity



- **Average Case: O(n log n)** – occurs when the pivot divides the array into roughly equal sub-arrays.
- **Worst Case: O(n^2)** – occurs with very unbalanced partitions, but is rare with random pivots.
- **Best Case: O(n log n)** – occurs when the pivot always divides the array evenly.

Comments of the Evaluators (if Any)	Evaluator’s Observation
	Marks Secured:________out of [50]. Signature of the Evaluator Date of Evaluation: